Virtual Memory: Concepts
虚拟内存:概念

100076202： 计算机系统导论

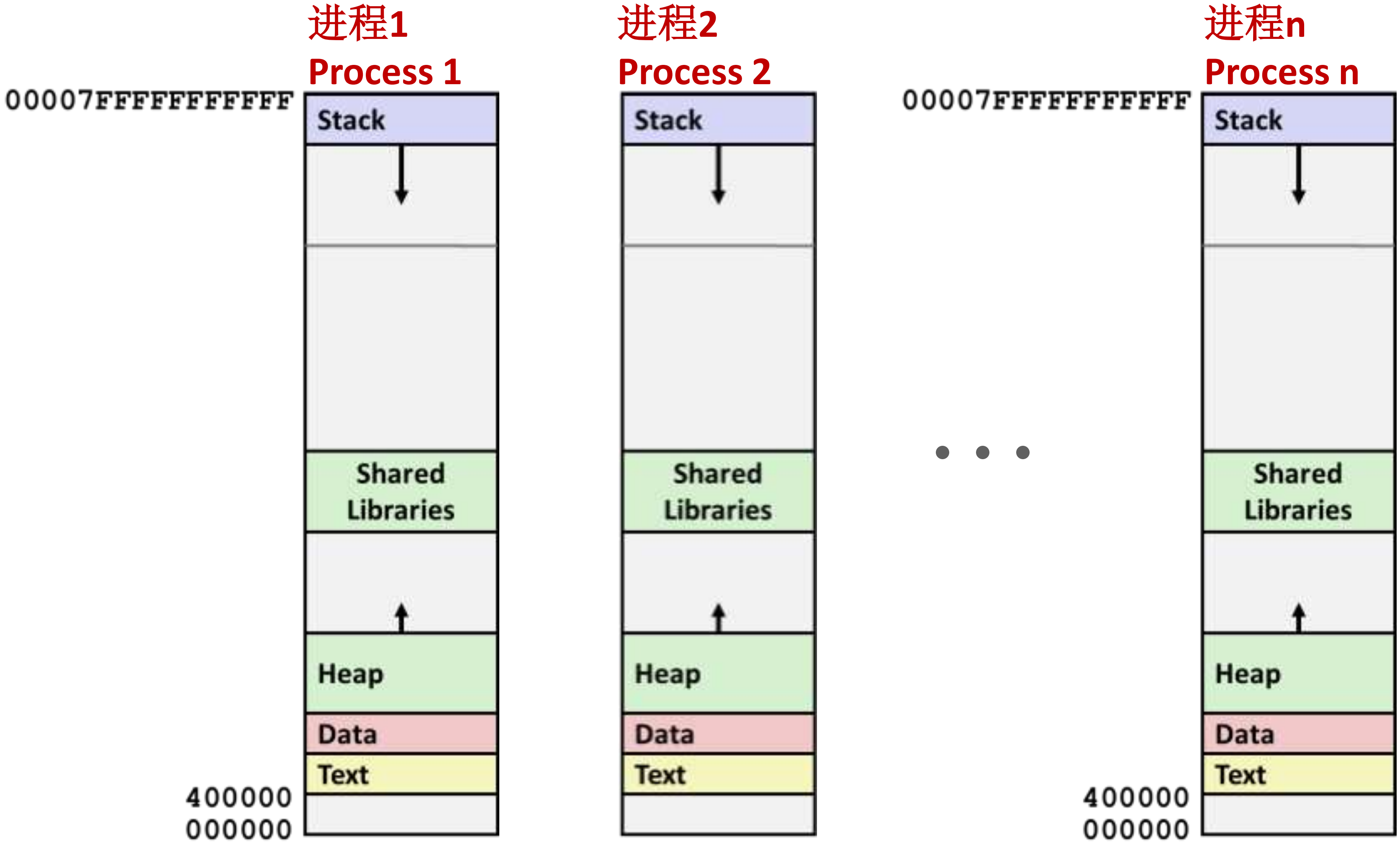
北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY
1940

任课教师:
宿红毅　　张艳　　　　黎有琦　　　颜珂

Carnegie
Mellon
University

原作者:
Randal E. Bryant and David R. O’Hallaron

嗯，这是怎么工作的？！
Hmmm, How Does This Work?!



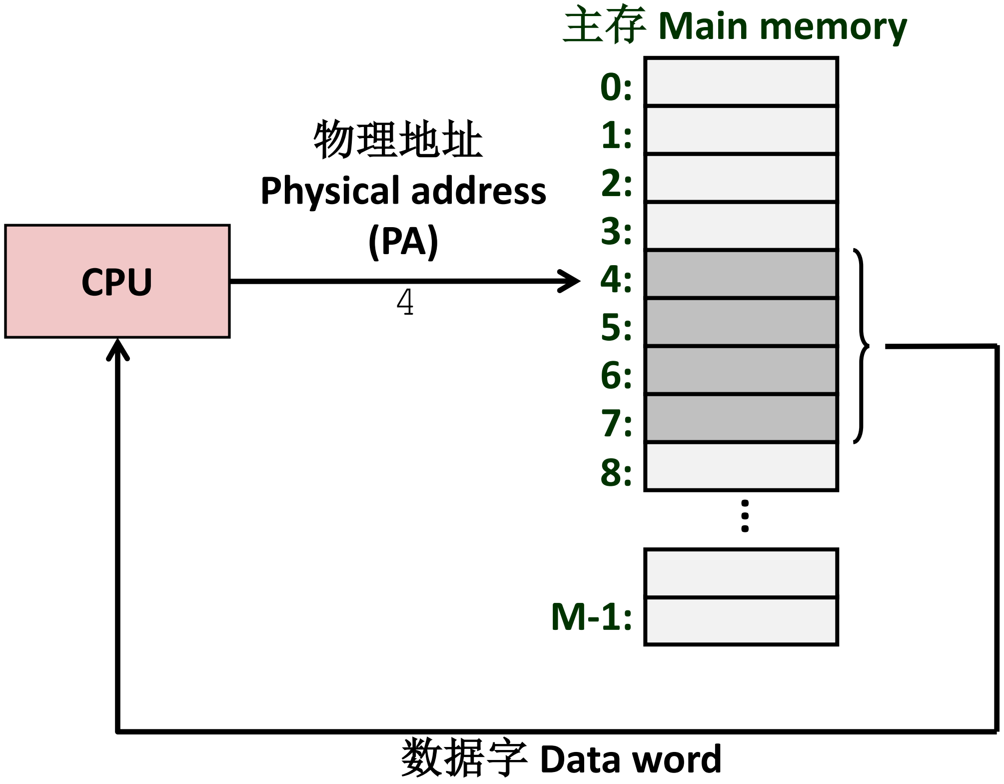
解决方案：虚拟内存（本次和下次课）

Solution: Virtual Memory (today and next lecture)

内容提纲 Today

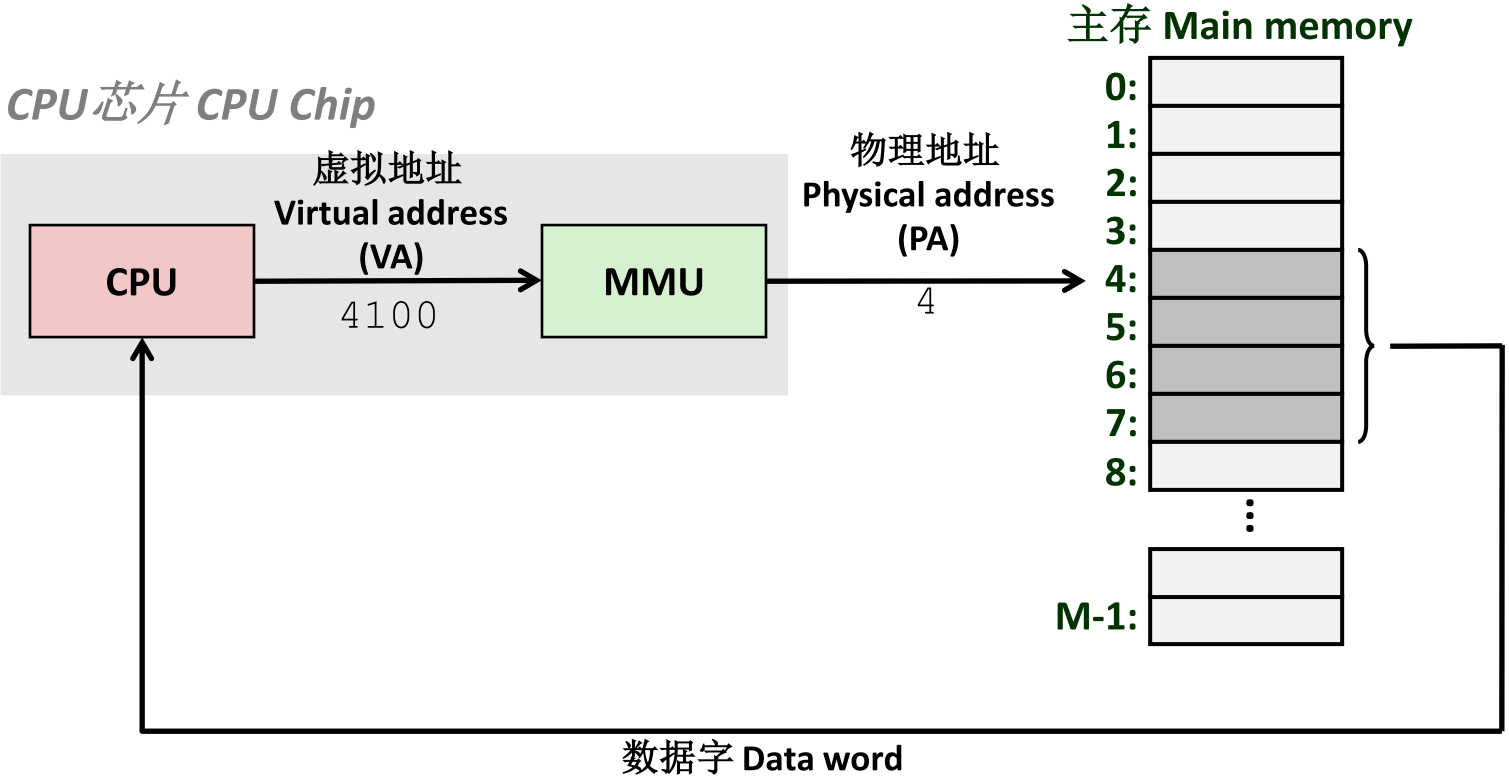
- **地址空间 Address spaces** CSAPP 9.1-9.2
- 基于虚拟内存的缓存机制 VM as a tool for caching CSAPP 9.3
- 基于虚拟内存的内存管理机制 VM as a tool for memory management CSAPP 9.4
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection CSAPP 9.5
- 地址翻译 Address translation CSAPP 9.6

使用物理寻址的系统
A System Using Physical Addressing



- 通常在车、电梯、数字相框等设备中简单系统的嵌入式微控制器使用 **Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames**

使用虚拟寻址的系统
A System Using Virtual Addressing



- 在所有现代服务器、笔记本和智能手机中使用 **Used in all modern servers, laptops, and smart phones**
- 计算机科学的伟大思想之一 **One of the great ideas in computer science**

地址空间 Address Spaces

- **线性地址空间：**连续非负整型地址的有序集合 **Linear address space:** Ordered set of contiguous non-negative integer addresses:

  {0, 1, 2, 3 … }

- **虚拟地址空间：** $N = 2^n$ 虚拟地址集合 **Virtual address space:** Set of $N = 2^n$ virtual addresses

  {0, 1, 2, 3, …, N-1}

- **物理地址空间：** $M = 2^m$ 物理地址集合 **Physical address space:** Set of $M = 2^m$ physical addresses

  {0, 1, 2, 3, …, M-1}

为什么需要虚拟内存(VM)?
Why Virtual Memory (VM)?

- 更高效地使用主存 **Uses main memory efficiently**
  - 使用DRAM作为一部分虚拟地址空间的缓存 Use DRAM as a cache for parts of a virtual address space

- 简化内存管理 **Simplifies memory management**
  - 每个进程都用同样的统一线性地址空间 Each process gets the same uniform linear address space

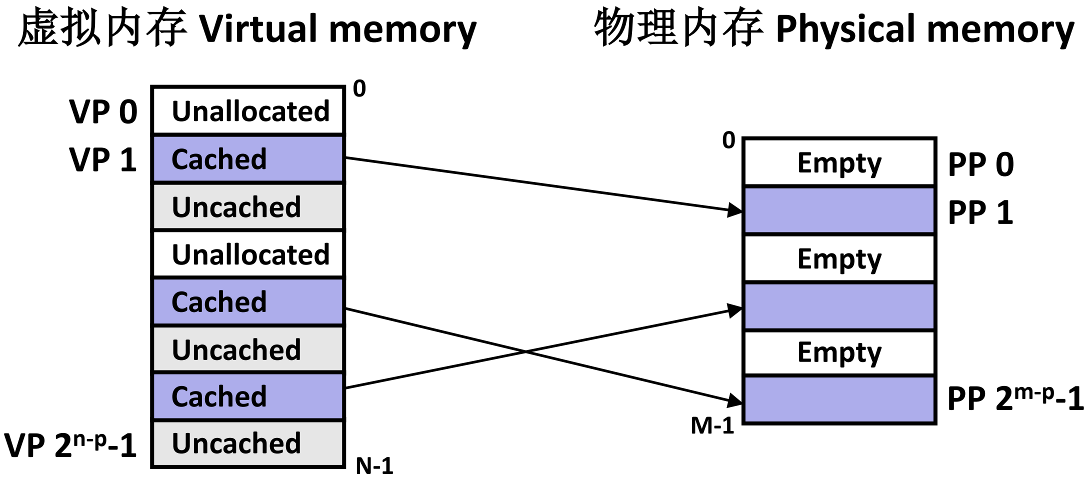
- 隔离的地址空间 **Isolates address spaces**
  - 一个进程不会干扰另一个进程的内存 One process can’t interfere with another’s memory
  - 用户程序不能访问特权内核信息和代码 User program cannot access privileged kernel information and code

内容提纲/Today

- 地址空间 Address spaces
- **基于虚拟内存的缓存机制 VM as a tool for caching**
- 基于虚拟内存的内存管理机制 VM as a tool for memory management
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection
- 地址翻译 Address translation

基于虚拟内存的缓存机制
VM as a Tool for Caching

- 概念上来讲，虚拟内存就是N个连续地存储在磁盘上的字节数组 **Conceptually, *virtual memory* is an array of N contiguous bytes stored on disk.**
- 磁盘上的数组的内容是缓存在物理内存中的（**DRAM**缓存）**The contents of the array on disk are cached in *physical memory* (*DRAM cache*)**
  - 这些cache块称为页（大小为P=2^p字节）These cache blocks are called *pages* (size is P = 2^p bytes)



虚拟页(VP)存储在磁盘上
Virtual pages (VPs) stored on disk

物理页(PP)缓存在DRAM中
Physical pages (PPs) cached in DRAM

DRAM缓存组织 DRAM Cache Organization

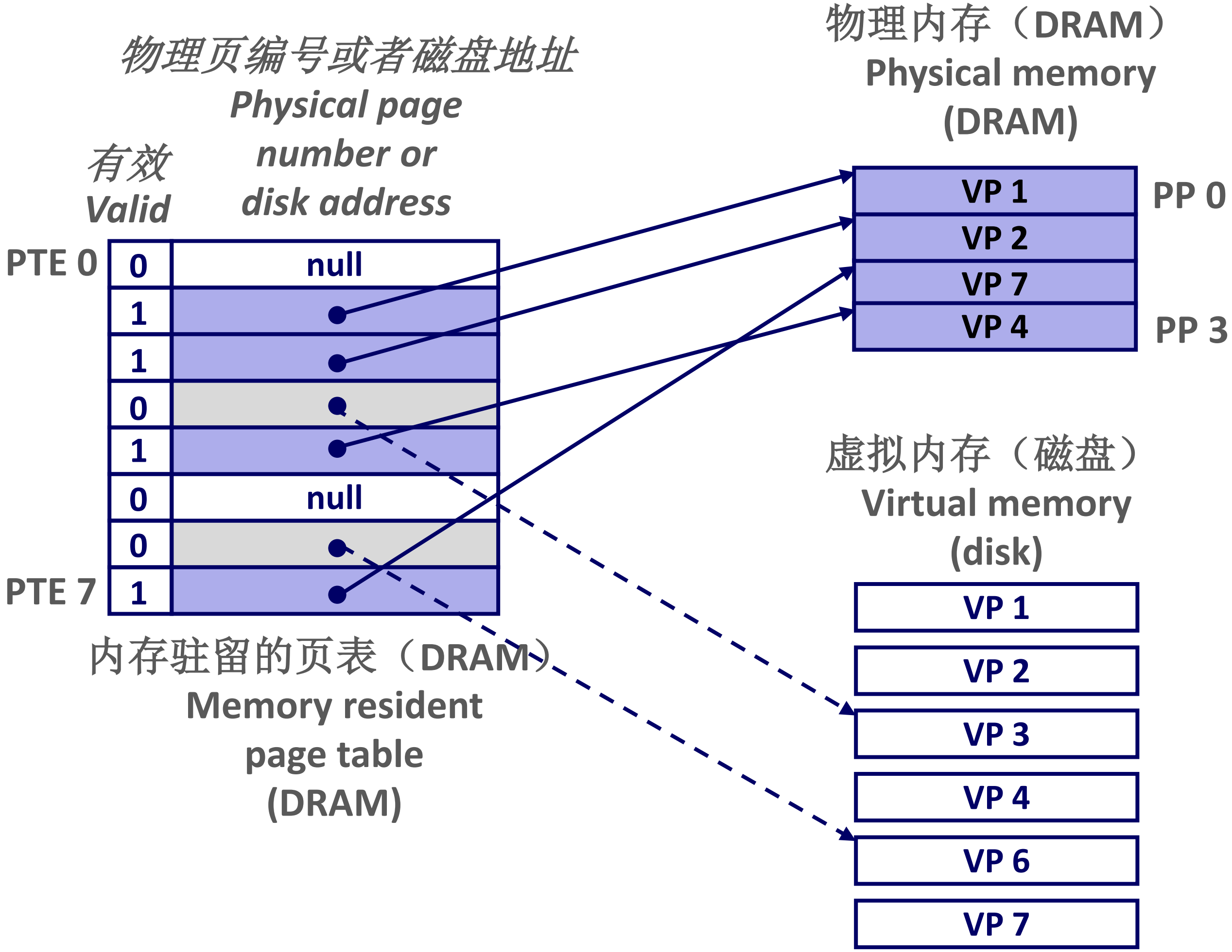
- **DRAM缓存组织是受不命中后惩罚会很高这一因素影响的 DRAM cache organization driven by the enormous miss penalty**
  - DRAM大概比SRAM慢***10***倍左右 DRAM is about ***10x*** slower than SRAM
  - 磁盘大概比DRAM慢***10000***倍 Disk is about ***10,000x*** slower than DRAM
  - 从磁盘装入块的时间大于1ms（超过一百万个时钟周期）Time to load block from disk > 1ms (> 1 million clock cycles)
    - 在此期间CPU能够做很多计算 CPU can do a lot of computation during that time

- 因此 **Consequences**
  - 比较大的页（块）：通常4 KB Large page (block) size: typically 4 KB
    - Linux的“巨大页”可以2MB（默认）到1GB Linux “huge pages” are 2 MB (default) to 1 GB
  - 全相联 Fully associative
    - 任意的虚拟页可以放在任意的物理页中 Any VP can be placed in any PP
    - 与Cache内存不同，需要一个更灵活的映射函数 Requires a “large” mapping function – different from cache memories
  - 高度复杂，替换算法开销比较大 Highly sophisticated, expensive replacement algorithms
    - 由于过于复杂和不确定性，无法在硬件中实现 Too complicated and open-ended to be implemented in hardware
  - 采用写回机制而不是写直达机制 Write-back rather than write-through

使能数据结构：页表

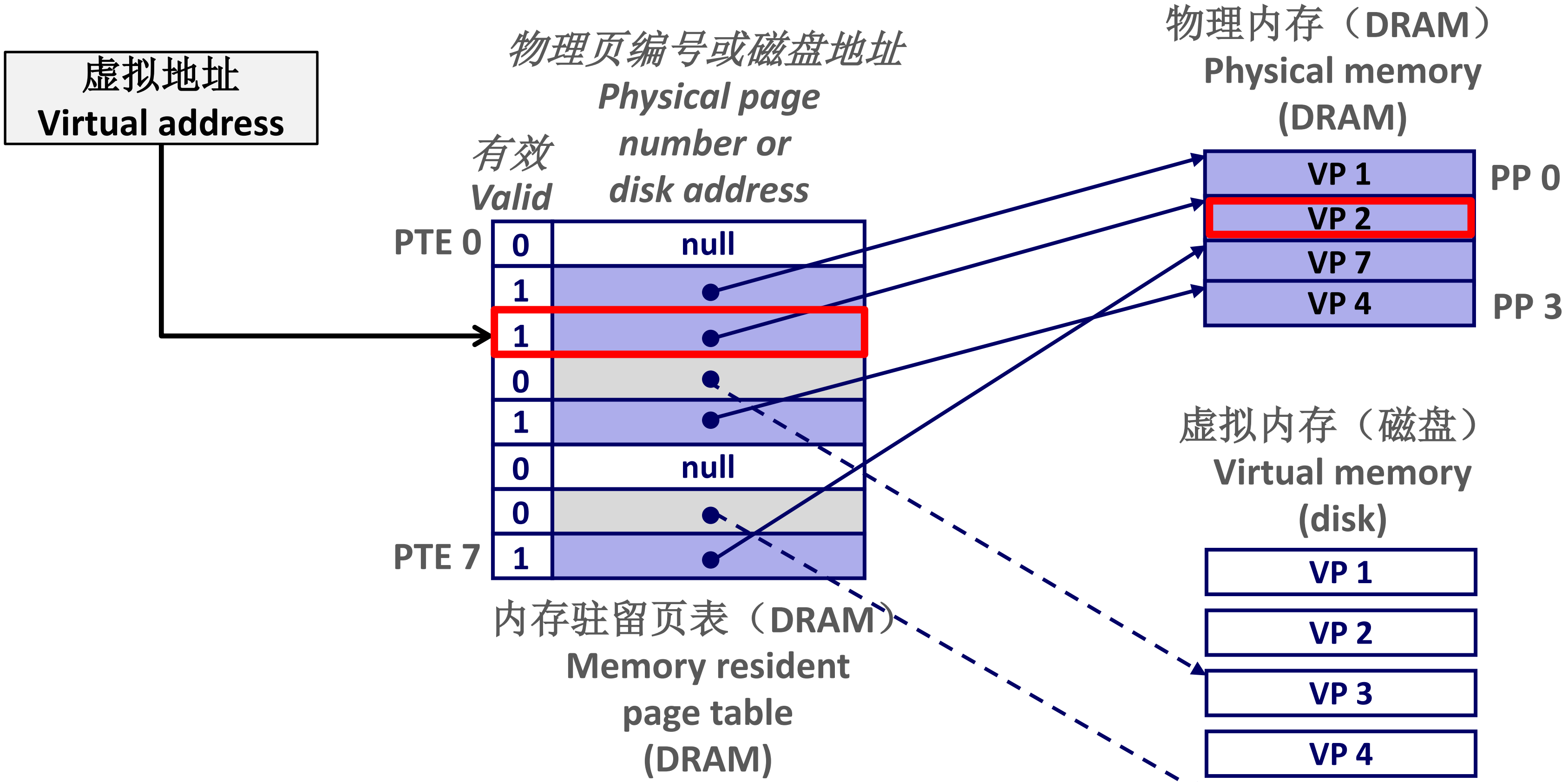
Enabling Data Structure: Page Table

- 一个页表实际上是将虚拟页映射物理页的页表条目（PTE）构成的数组 A ***page table*** is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - 每个进程在DRAM中的核心数据结构 Per-process kernel data structure in DRAM



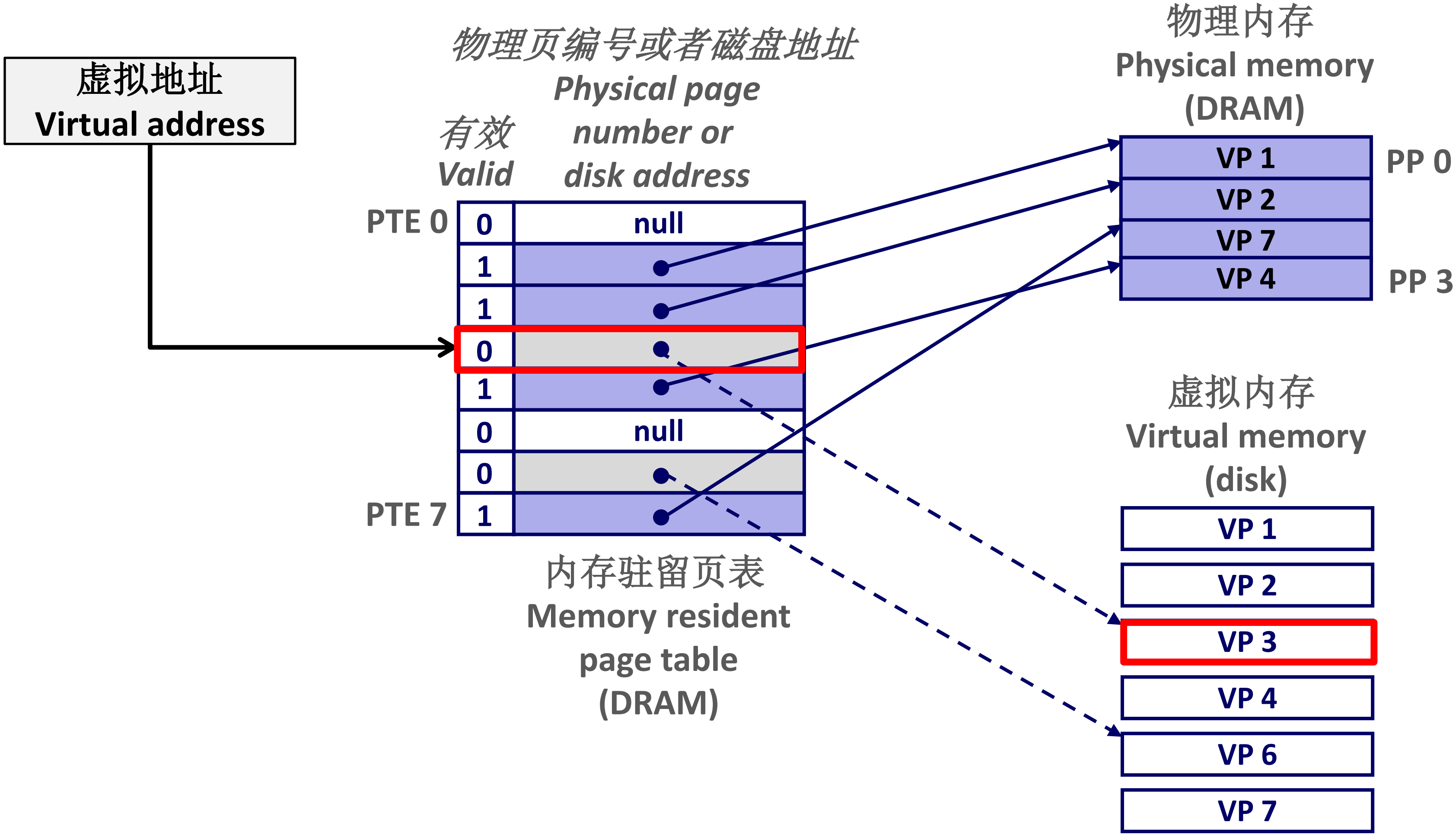
页命中 Page Hit

- *页命中*：引用的虚拟内存字在物理内存中（DRAM命中） ***Page hit:*** **reference to VM word that is in physical memory (DRAM cache hit)**



缺页中断 Page Fault

- *缺页中断*：引用的虚拟字不在物理内存中(DRAM缓存不命中) ***Page fault:*** **reference to VM word that is not in physical memory (DRAM cache miss)**



虚拟地址
Virtual address

物理页编号或者磁盘地址
Physical page number or disk address

有效
Valid

PTE 0
PTE 7

0	null
1	●
1	●
0	●
1	●
0	null
0	●
1	●

内存驻留页表
Memory resident page table
(DRAM)

物理内存
Physical memory
(DRAM)

VP 1 PP 0
VP 2
VP 7
VP 4 PP 3

虚拟内存
Virtual memory
(disk)

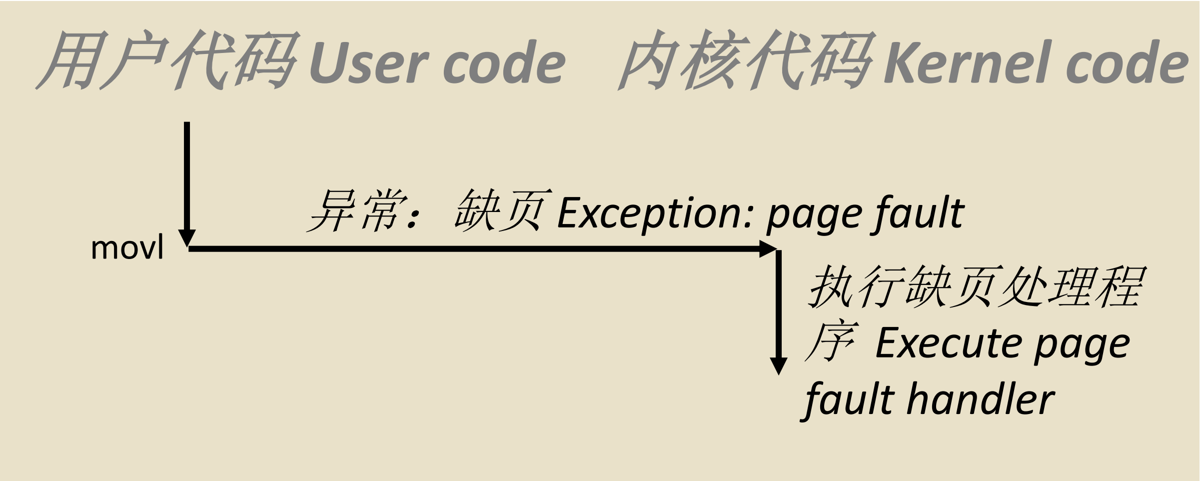
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

触发缺页中断 Triggering a Page Fault

- 用户对内存位置写入 User writes to memory location

```
80483b7:        c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

- 用户内存的这部分（页）当前在磁盘上 That portion (page) of user’s memory is currently on disk
- **MMU触发缺页异常** MMU triggers page fault exception
  - (更多细节下次课讲 More details in later lecture)
  - 提升优先级到监督态 Raise privilege level to supervisor mode
  - 引起对软件缺页中断处理程序的过程调用 Causes procedure call to software page fault handler



用户代码 **User code** *内核代码* **Kernel code**

movl

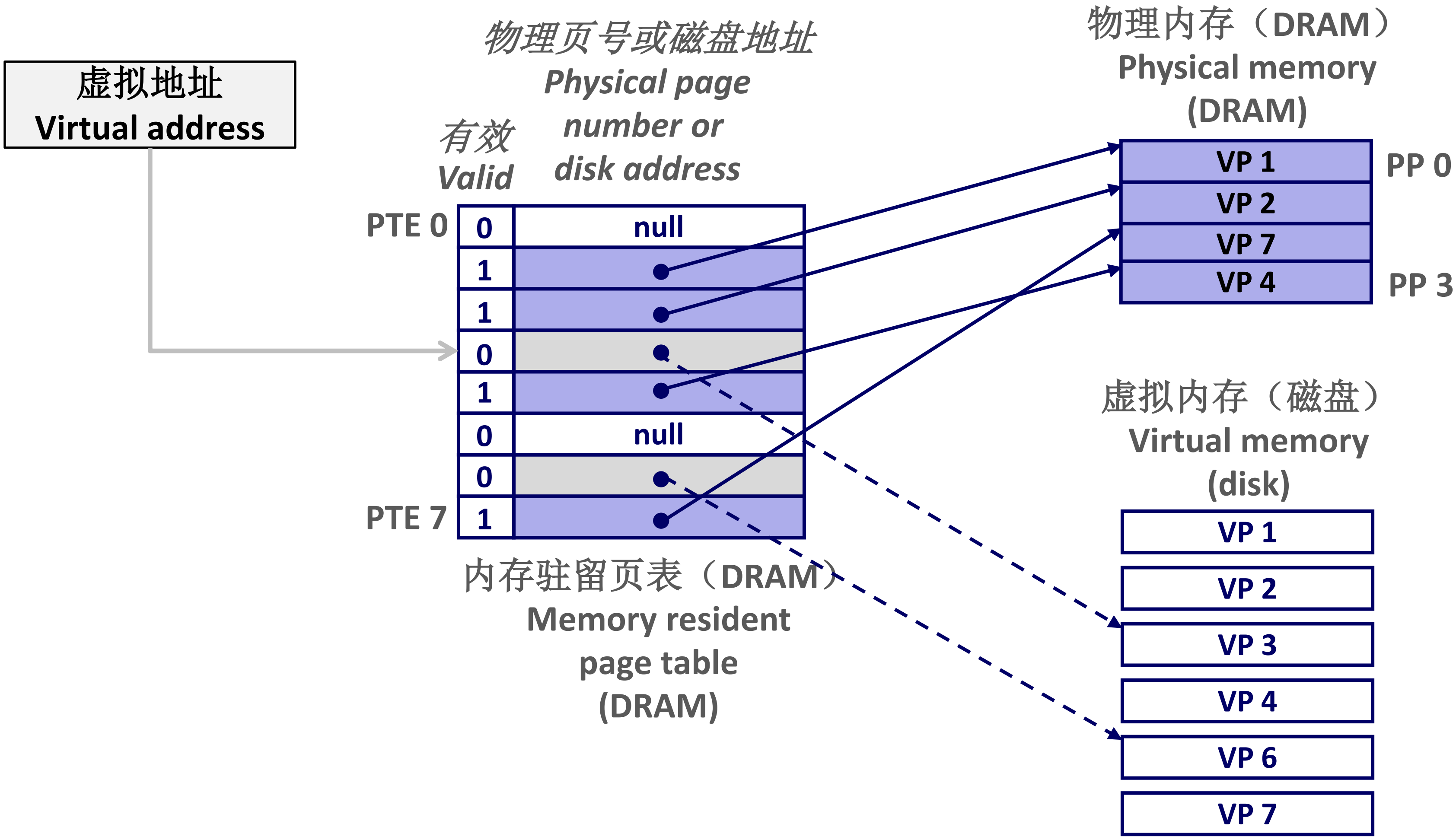
异常：缺页 Exception: page fault

执行缺页处理程序 Execute page fault handler

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

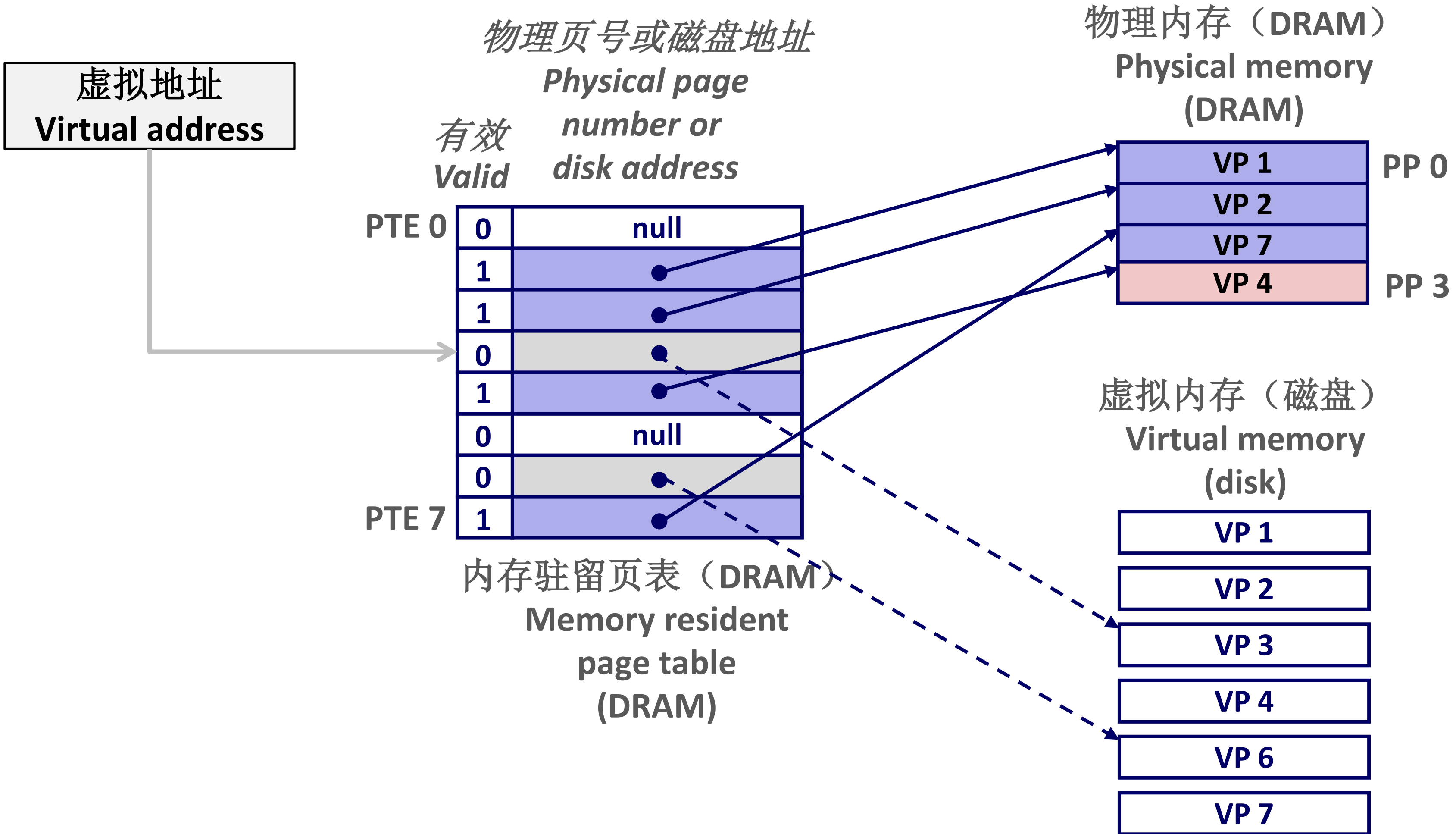
缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断（异常的一种） Page miss causes page fault (an exception)



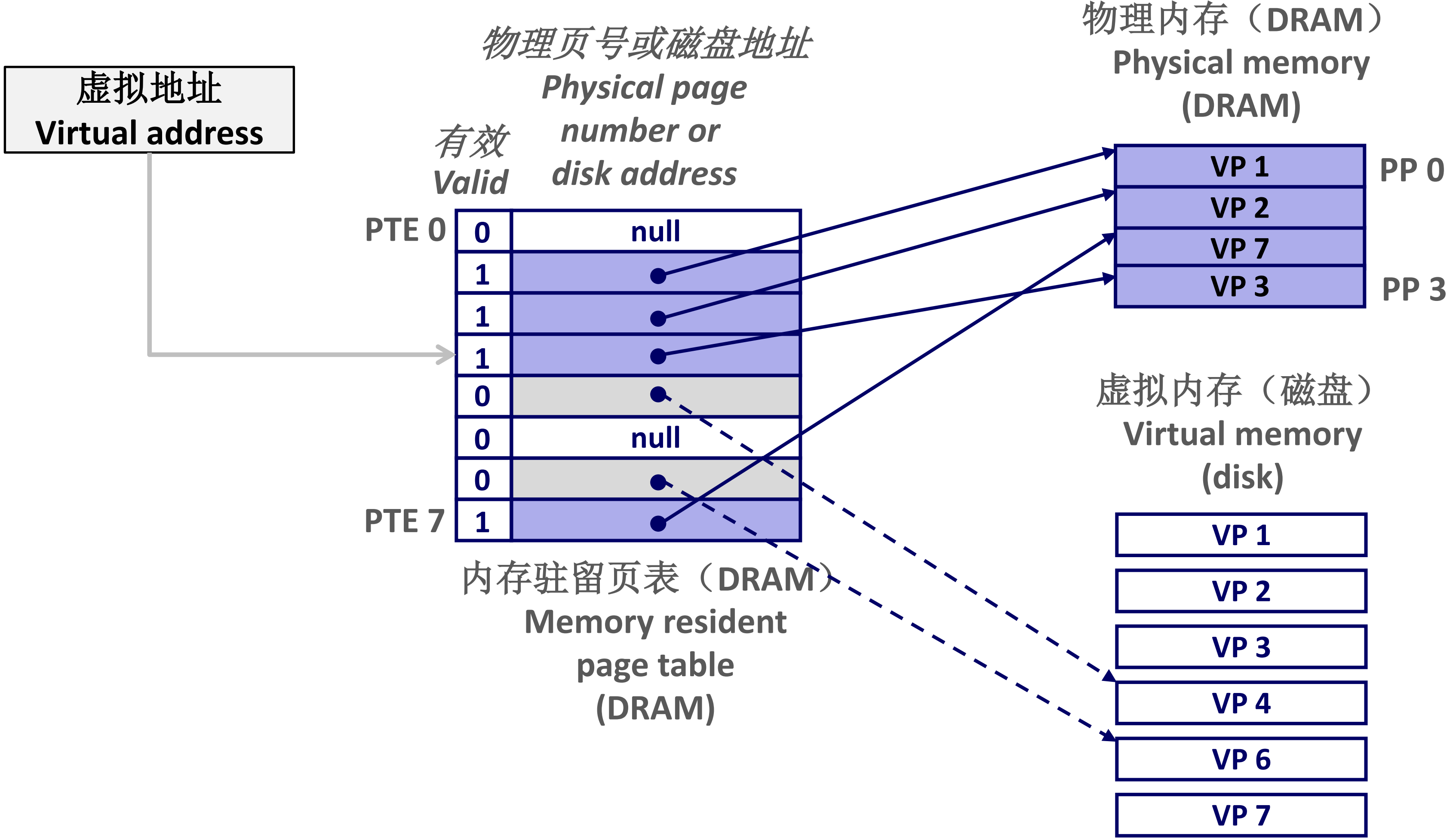
缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断（异常的一种） Page miss causes page fault (an exception)
- 缺页中断处理程序选择一个牺牲页换出（以**VP 4**为例） Page fault handler selects a victim to be evicted (here VP 4)



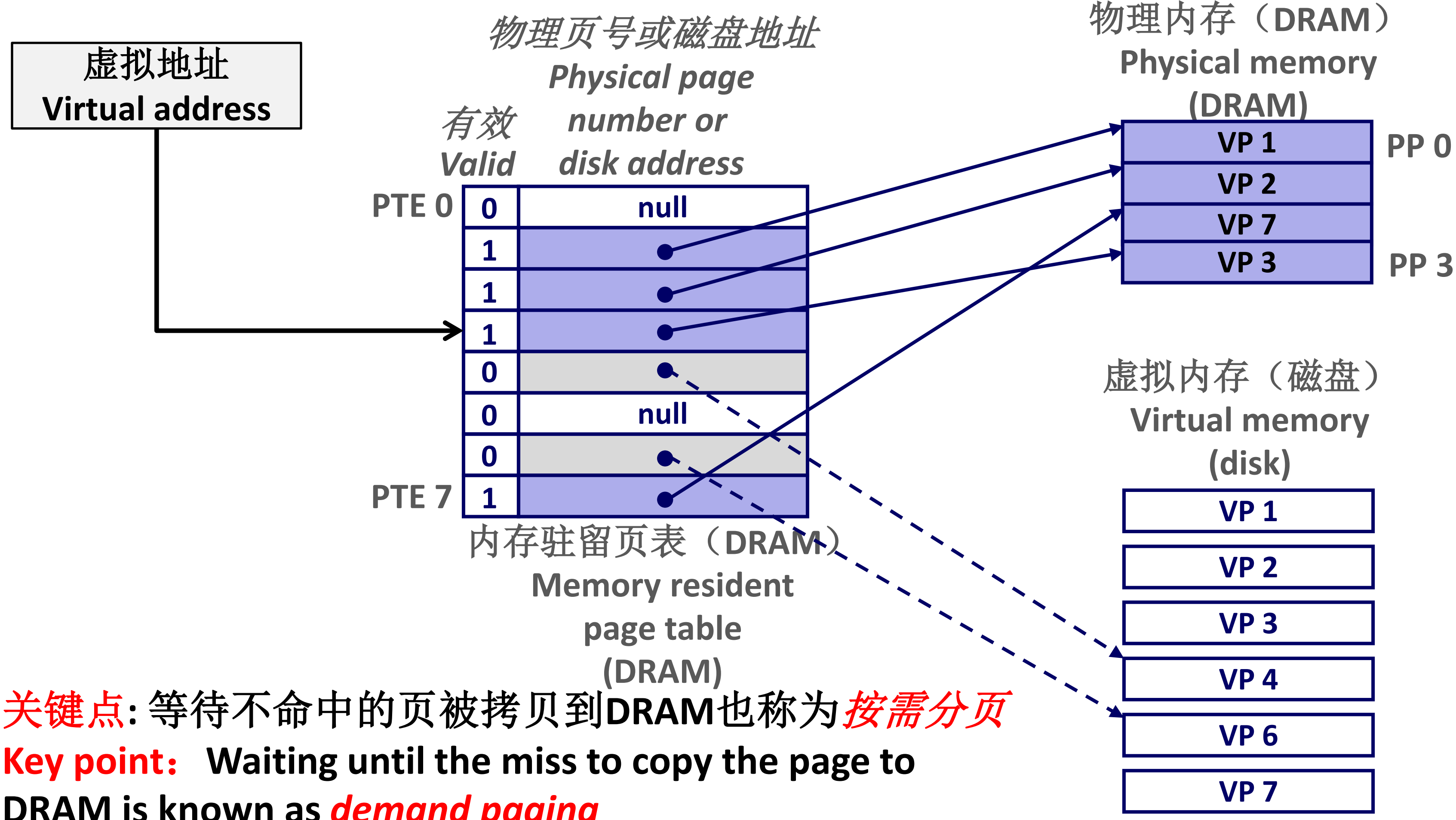
缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断（异常的一种） Page miss causes page fault (an exception)
- 缺页中断处理程序选择一个牺牲页换出（以**VP 4**为例）Page fault handler selects a victim to be evicted (here VP 4)



缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断(异常的一种) Page miss causes page fault (an exception)
- 缺页中断处理程序选择一个牺牲页换出（以VP 4为例） Page fault handler selects a victim to be evicted (here VP 4)
- 触发指令重新开始执行：页命中！ Offending instruction is restarted: page hit!



虚拟地址
Virtual address

物理页号或磁盘地址
Physical page number or disk address

有效
Valid

	Valid	Physical page number or disk address
PTE 0	0	null
	1	●
	1	●
	1	●
	0	●
	0	null
	0	●
PTE 7	1	●

内存驻留页表（DRAM）
Memory resident page table (DRAM)

物理内存（DRAM）
Physical memory (DRAM)

VP 1	**PP 0**
VP 2	
VP 7	
VP 3	**PP 3**

虚拟内存（磁盘）
Virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

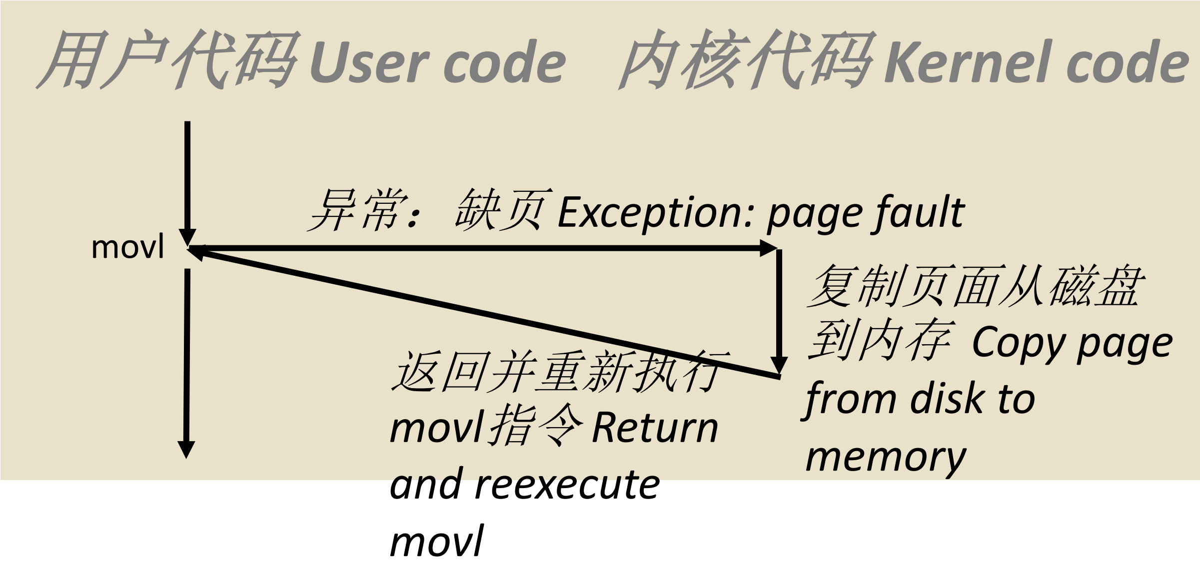
关键点: **等待不命中的页被拷贝到DRAM也称为*按需分页***
Key point： Waiting until the miss to copy the page to DRAM is known as *demand paging*

结束缺页中断 Completing page fault

- 缺页中断处理程序执行中断返回指令（iret）Page fault handler executes return from interrupt (`iret`) instruction
  - 类似于ret指令，但是还会恢复优先级 Like `ret` instruction, but also restores privilege level
  - 返回到引起故障的指令 Return to instruction that caused fault
  - 但是，这次不会产生缺页中断 But, this time there is no page fault

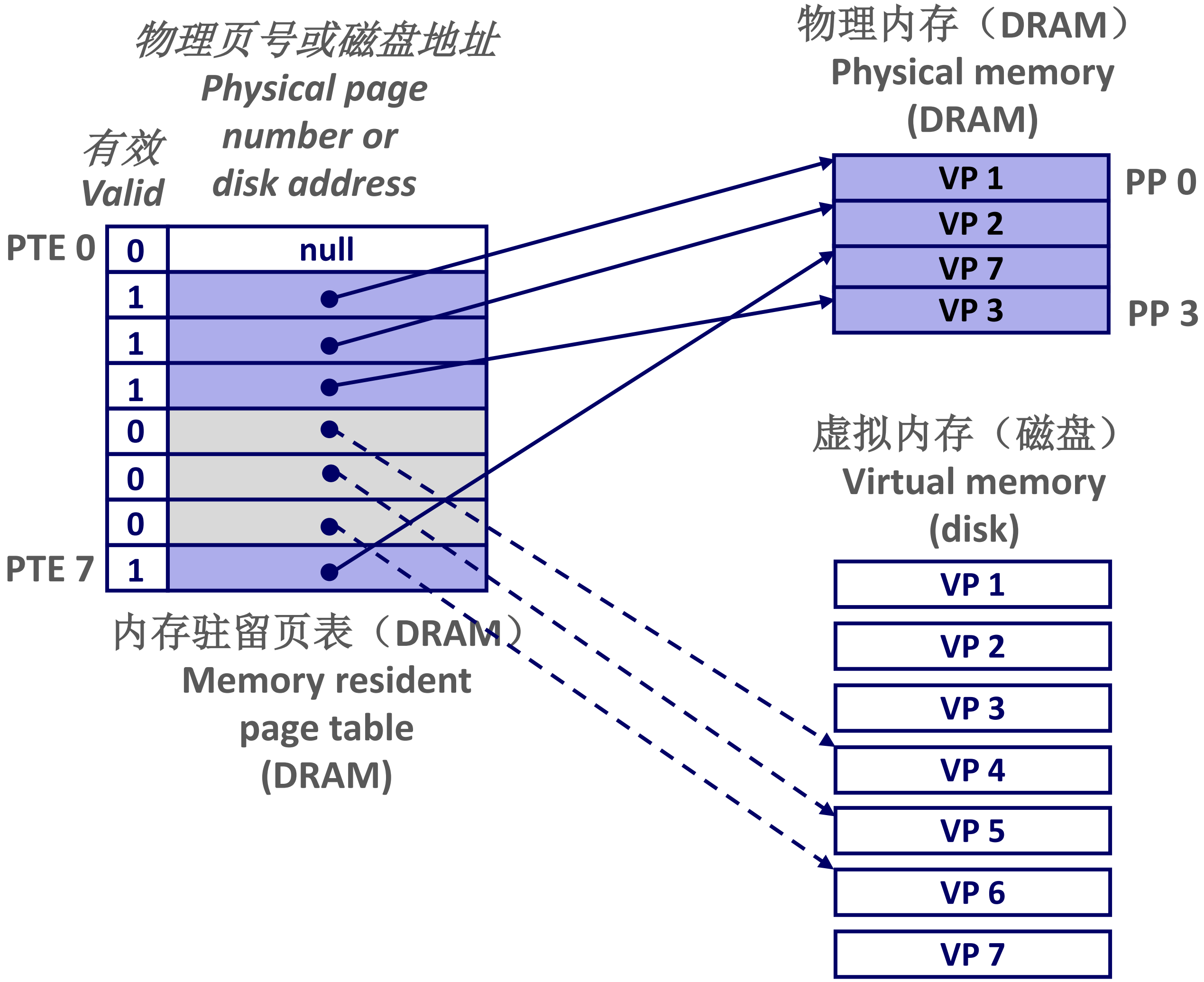
```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
 80483b7:        c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```


用户代码 User code　*内核代码 Kernel code*

页分配 Allocating Pages

- 分配虚拟内存的一个新页（VP 5）Allocating a new page (VP 5) of virtual memory.



物理页号或磁盘地址
Physical page number or disk address

有效
Valid

PTE 0
0 null
1
1
1
0
0
0
PTE 7
1

内存驻留页表（DRAM）
Memory resident page table (DRAM)

物理内存（DRAM）
Physical memory (DRAM)

VP 1 PP 0
VP 2
VP 7
VP 3 PP 3

虚拟内存（磁盘）
Virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7

局部性再次发挥作用
Locality to the Rescue Again!

- 虚拟内存看起来非常低效，能有效工作是因为局部性 **Virtual memory seems terribly inefficient, but it works because of locality.**

- 在任何时间点，程序更倾向于只访问一个活跃的虚拟页集合，也称为工作集 **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - 具有更好的时间局部性的程序会有更小的工作集 Programs with better temporal locality will have smaller working sets

- 如果工作集的大小小于主存大小 **If (working set size < main memory size)**
  - 每个进程在强制不命中后就会获得比较好的性能 Good performance for one process after compulsory misses

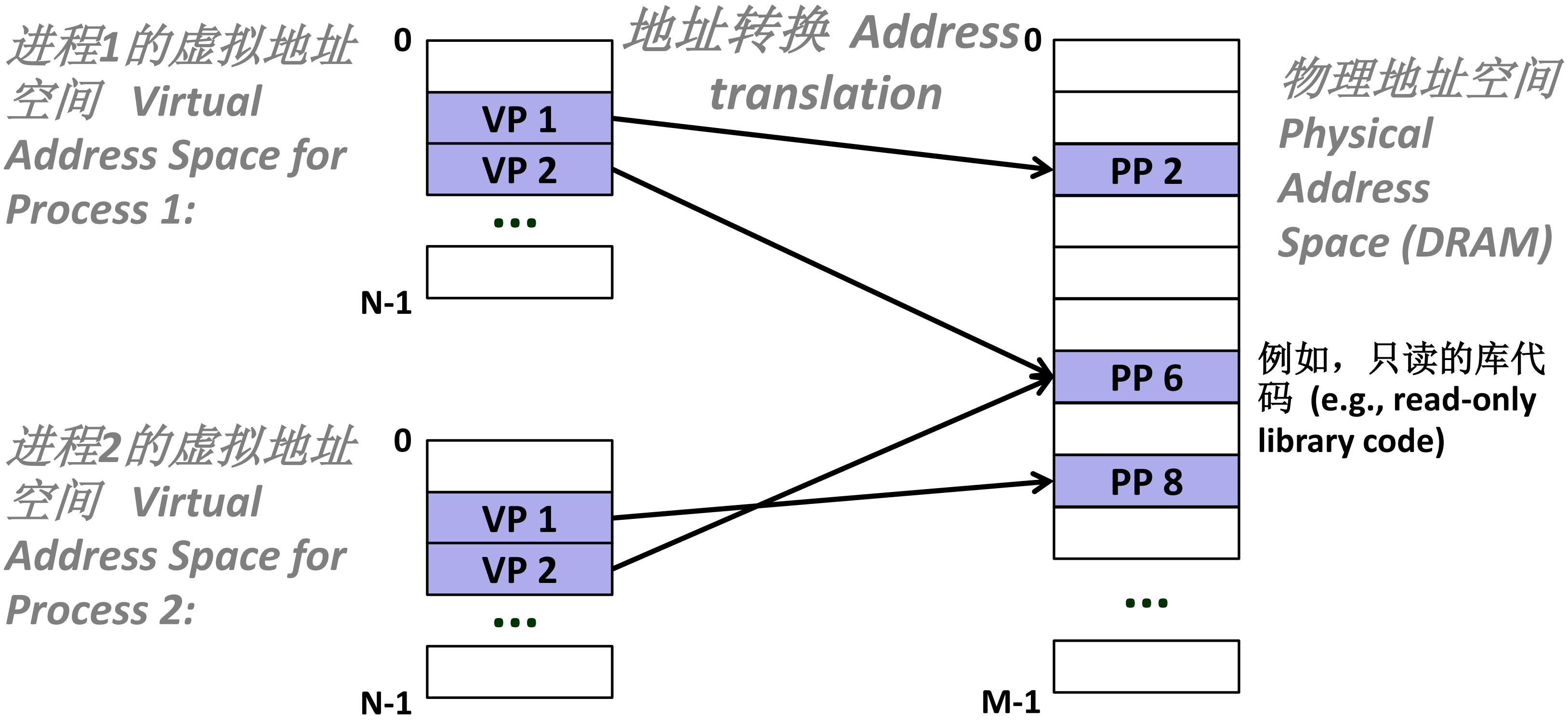
- 如果工作集的总大小大于主存大小 **If (SUM(working set sizes) > main memory size)**
  - *抖动：性能会由于持续的页面换入换出而变差* *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously
  - 如果多个进程同时运行，在它们的总工作集大小大于主存大小时发生抖动 If multiple processes run at the same time, thrashing occurs if their total working set size > main memory size

议题 Today

- 地址空间 Address spaces
- 基于虚拟内存的缓存机制 VM as a tool for caching
- **基于虚拟内存的内存管理机制 VM as a tool for memory management**
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection
- 地址翻译 Address translation

基于虚拟内存的内存管理机制
VM as a Tool for Memory Management

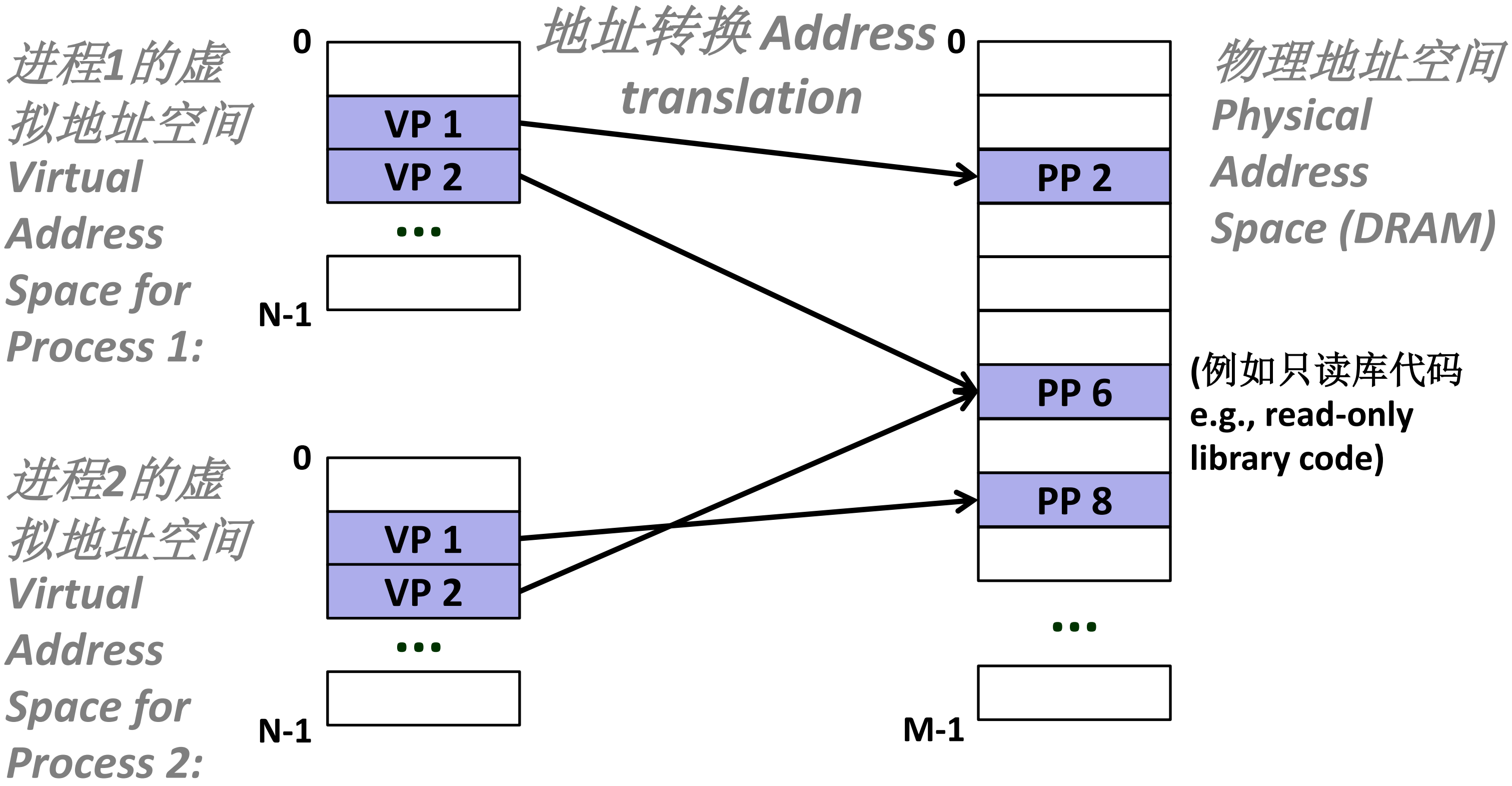
- 关键点：每个进程有自己的虚拟地址空间 **Key idea: each process has its own virtual address space**
  - 将内存看做简单的线性数组 It can view memory as a simple linear array
  - 映射函数将地址分散到物理内存中 Mapping function scatters addresses through physical memory
    - 好的映射函数会提高局部性 Well-chosen mappings can improve locality



例如，只读的库代码 **(e.g., read-only library code)**

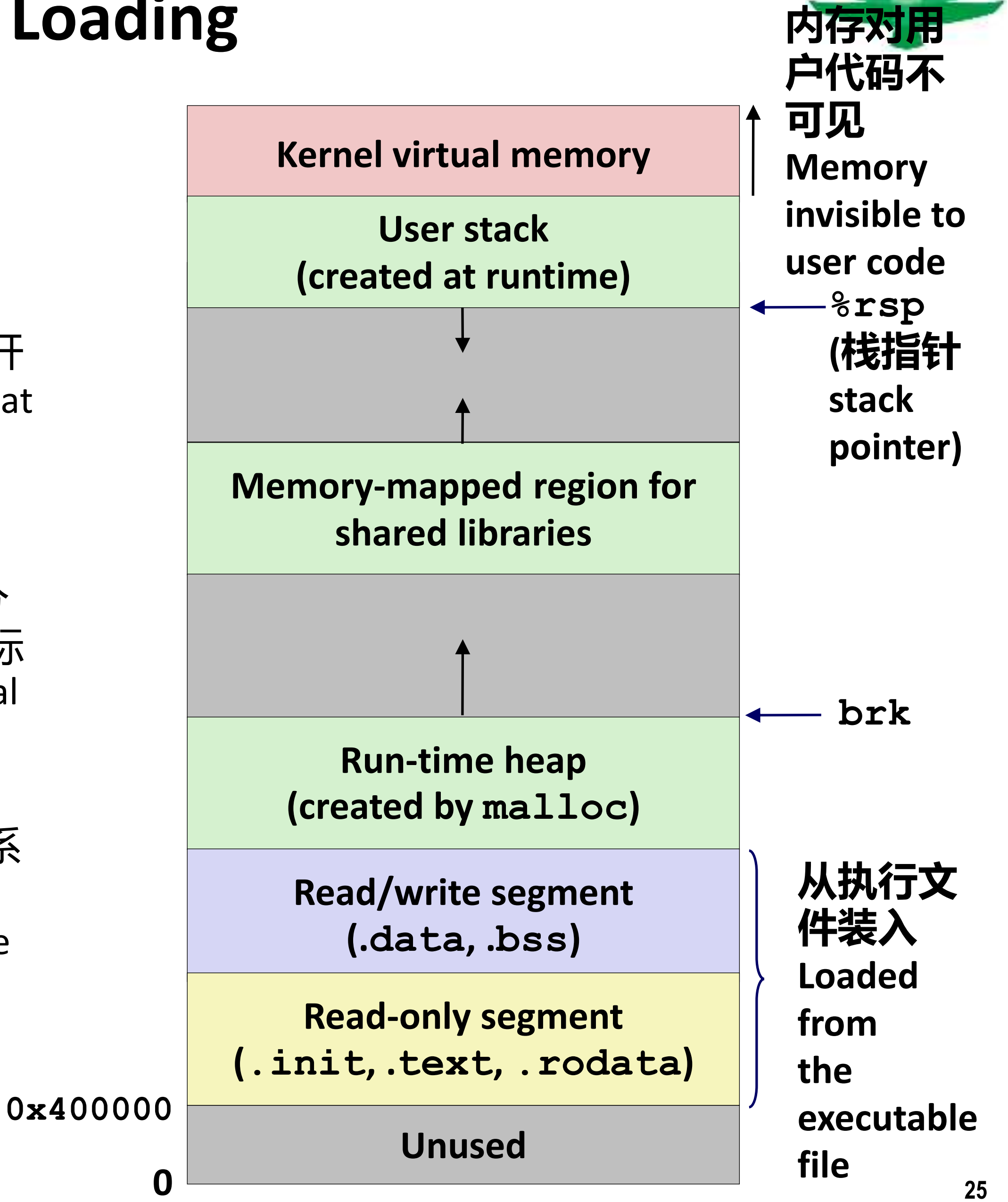
基于虚拟内存的内存管理机制
VM as a Tool for Memory Management

- 简化内存分配 **Simplifying memory allocation**
  - 每个虚拟页可以被映射到任意物理页 Each virtual page can be mapped to any physical page
  - 一个虚拟页可以在不同的时间点存储在不同的物理页中 A virtual page can be stored in different physical pages at different times
- 在进程间共享代码和数据 **Sharing code and data among processes**
  - 将虚拟页映射到同一个物理页 Map virtual pages to the same physical page (here: PP 6)



简化链接和加载
Simplifying Linking and Loading

- **链接 Linking**
  - 每个程序都有类似的虚拟地址空间 Each program has similar virtual address space
  - 代码、数据和堆总是从相同的地址开始 Code, data, and heap always start at the same addresses.
- **加载 Loading**
  - **`execve`**负责为`.text`和`.data`节分配虚拟页并创建页表条目，并将其标记为无效 **`execve`** allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  - .text和.data节中的页是由虚拟内存系统按需一页一页拷贝的 The **`.text`** and **`.data`** sections are copied, page by page, on demand by the virtual memory system

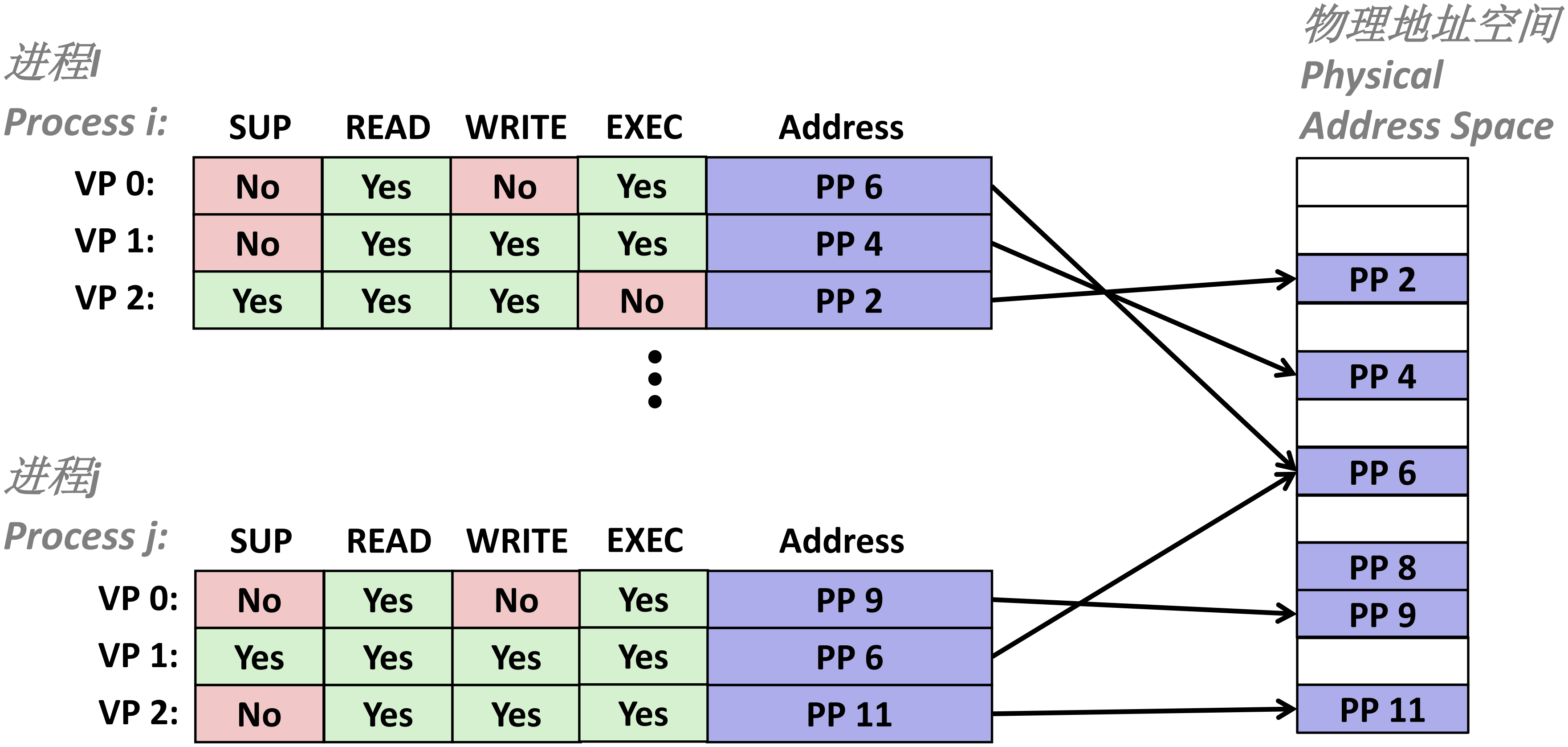


议题 Today

- 地址空间 Address spaces
- 基于虚拟内存的缓存机制 VM as a tool for caching
- 基于虚拟内存的内存管理机制 VM as a tool for memory management
- **基于虚拟内存的内存保护机制 VM as a tool for memory protection**
- 地址翻译 Address translation

基于虚拟内存的内存保护机制
VM as a Tool for Memory Protection

- 对页表记录进行扩展增加权限位 **Extend PTEs with permission bits**
- **MMU在每次内存访问时检查 MMU checks these bits on each access**



进程I
Process i:

	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 6
VP 1:	No	Yes	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

⋮

进程j
Process j:

	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 9
VP 1:	Yes	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	Yes	PP 11

物理地址空间
Physical Address Space

SUP：需要内核模式 SUP: requires kernel mode

议题 Today

- 地址空间 Address spaces
- 基于虚拟内存的缓存机制 VM as a tool for caching
- 基于虚拟内存的内存管理机制 VM as a tool for memory management
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection
- **地址翻译 Address translation**

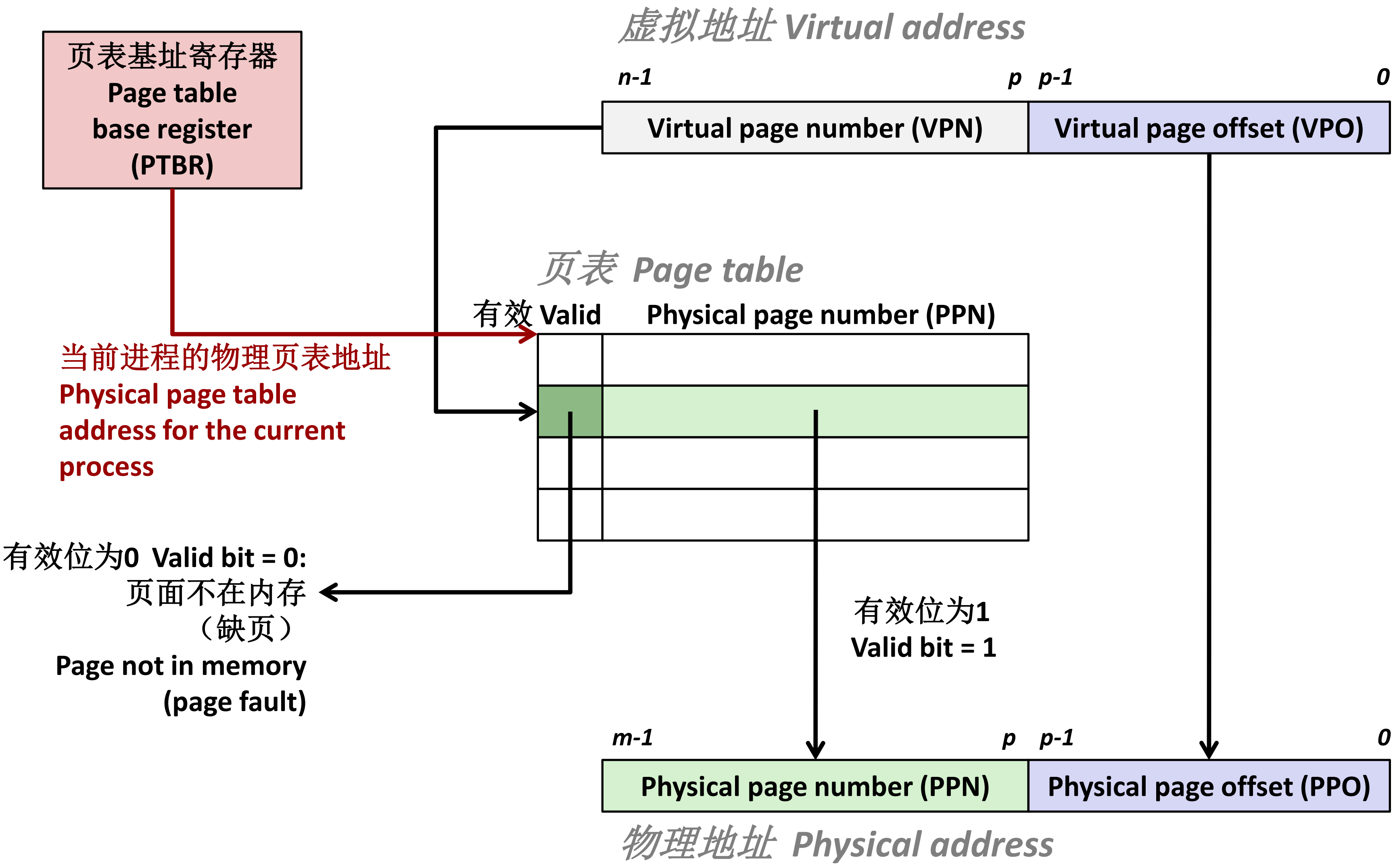
虚拟地址翻译 VM Address Translation

- **虚拟地址空间 Virtual Address Space**
  - $V = \{0, 1, \ldots, N-1\}$
- **物理地址空间 Physical Address Space**
  - $P = \{0, 1, \ldots, M-1\}$
- **地址翻译 Address Translation**
  - ***映射*** **MAP:** $V \rightarrow P\ U\ \{\varnothing\}$
  - 对于虚拟地址a For virtual address ***a***:
    - ***MAP(a) = a′*** if data at virtual address ***a*** is at physical address ***a′*** in ***P***
      如果虚拟地址a中的数据在P的物理地址a′中
    - ***MAP(a) = ∅*** if data at virtual address ***a*** is not in physical memory
      如果虚拟地址a中的数据不在物理内存中
      - 非法的或者在磁盘上 Either invalid or stored on disk

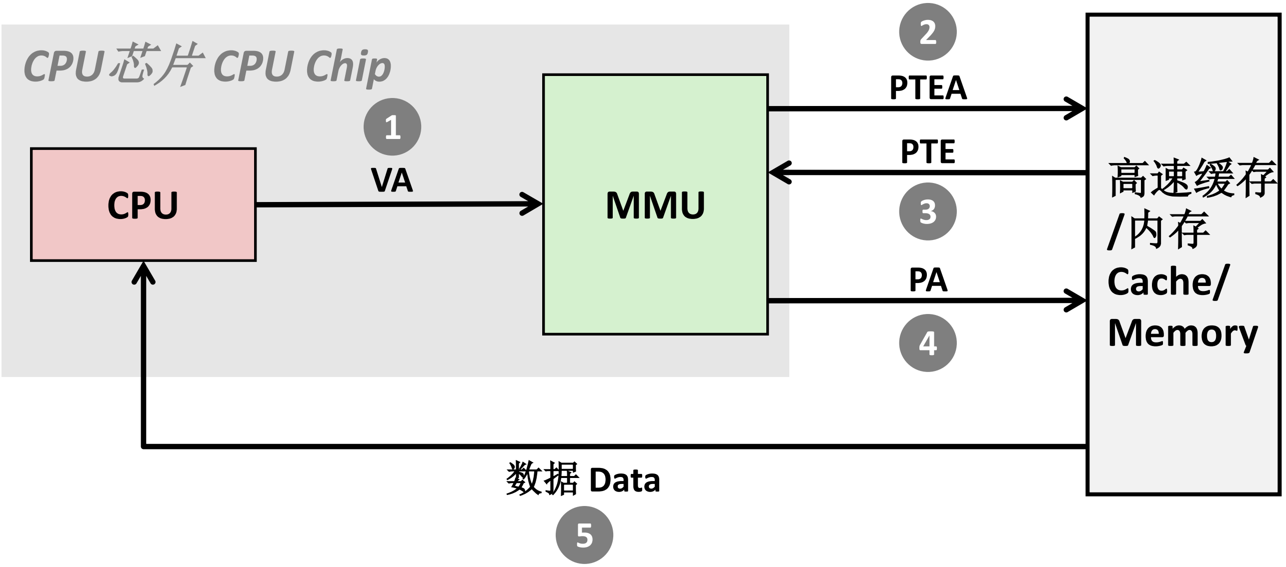
地址翻译符号总结
Summary of Address Translation Symbols

- 基本参数 **Basic Parameters**
  - **N = 2^n** : Number of addresses in virtual address space 虚拟地址空间的地址个数
  - **M = 2^m** : Number of addresses in physical address space 物理地址空间的地址个数
  - **P = 2^p** : Page size (bytes) 页大小（字节）
- 虚拟地址VA划分 **Components of the virtual address (VA)**
  - **TLBI**: TLB index TLB索引
  - **TLBT**: TLB tag TLB标记
  - **VPO**: Virtual page offset 虚拟页内偏移
  - **VPN**: Virtual page number 虚拟页号
- 物理地址PA划分 **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO) 物理页内偏移 （同VPO）
  - **PPN:** Physical page number 物理页号

基于页表的地址翻译
Address Translation With a Page Table



地址翻译：页命中 Address Translation: Page Hit



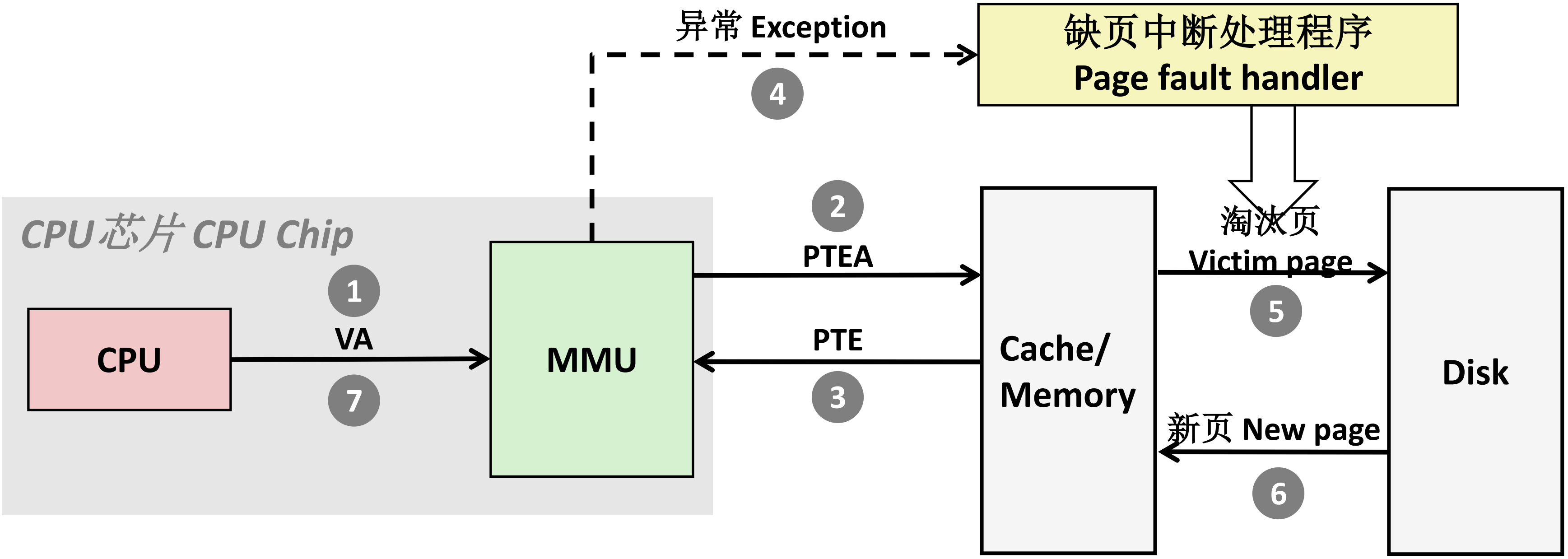
1) 处理器将虚拟地址发送给MMU Processor sends virtual address to MMU

2-3) MMU从内存页表中获取页表条目 MMU fetches PTE from page table in memory

4) MMU将物理地址发给Cache或者主存 MMU sends physical address to cache/memory

5) Cache或者主存将数据字发送给处理器 Cache/memory sends data word to processor

地址翻译：缺页中断 Address Translation: Page Fault



1) 处理器将虚拟地址发给MMU Processor sends virtual address to MMU

2-3)MMU从内存中的页表取出页表条目   MMU fetches PTE from page table in memory

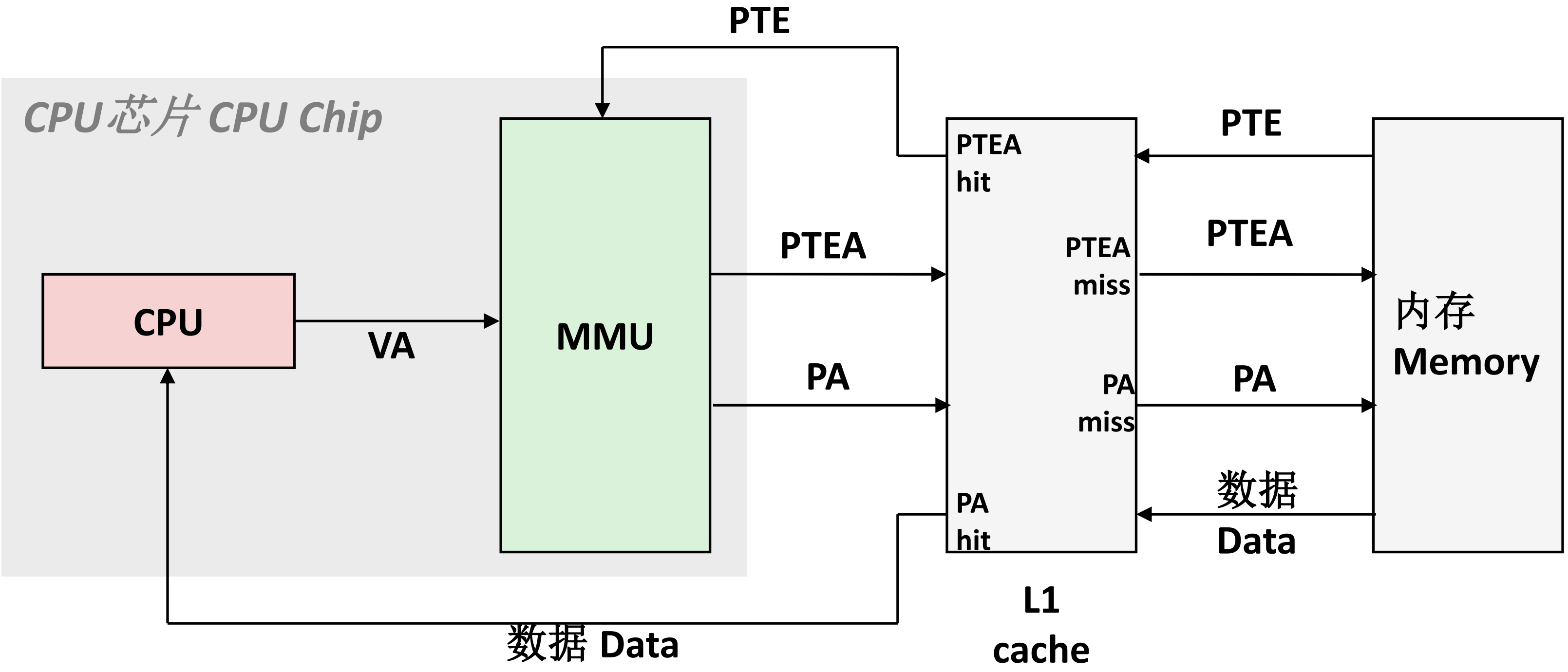
4) 当有效位为0时MMU触发缺页中断异常 Valid bit is zero, so MMU triggers page fault exception

5) 异常处理程序找到一个换出页（如果是脏页则要写回磁盘） Handler identifies victim (and, if dirty, pages it out to disk)

6) 异常处理程序拷贝页并更新页表条目 Handler pages in new page and updates PTE in memory

7)异常处理程序返回原进程中断的指令重新执行   Handler returns to original process, restarting faulting instruction

整合虚拟内存和Cache Integrating VM and Cache



VA：虚拟地址 VA: virtual address, PA：物理地址 PA: physical address,
PTE：页表条目 PTE: page table entry, PTEA时页表条目地址 PTEA = PTE address

使用TLB加速地址翻译
Speeding up Translation with a TLB

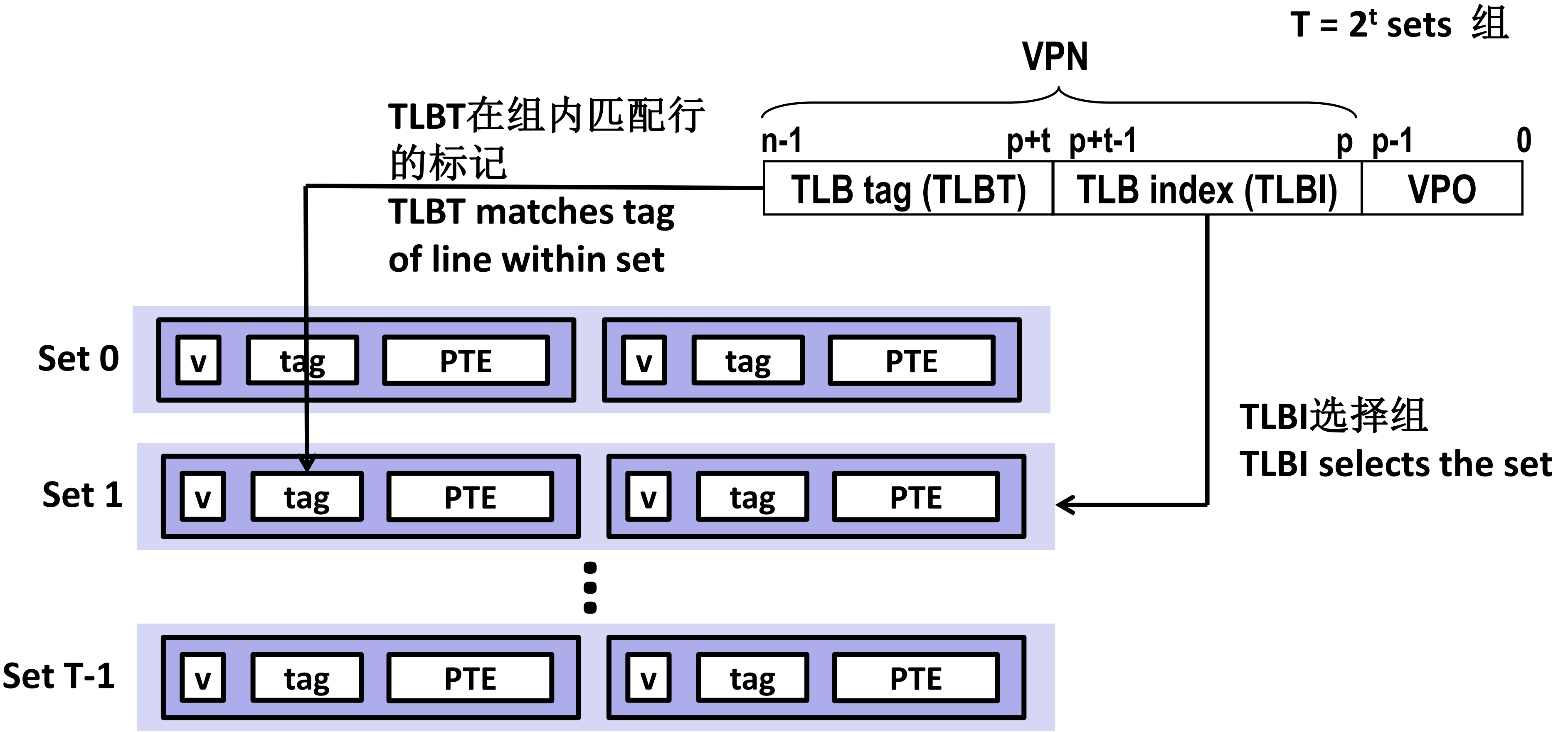
- 页表条目（PTE）像任何其他内存字一样缓存在L1 cache中 **Page table entries (PTEs) are cached in L1 like any other memory word**
  - 由于其他数据访问PTE可能会被驱逐出内存 PTEs may be evicted by other data references
  - PTE命中仍然需要较小的L1缓存延迟 PTE hit still requires a small L1 delay
- 解决方案：*翻译后备缓冲区*（**TLB**） **Solution: *Translation Lookaside Buffer* (TLB)**
  - 在MMU中的小型组相联硬件缓存 Small set-associative hardware cache in MMU
  - 将虚拟页号映射为物理页号 Maps virtual page numbers to physical page numbers
  - 包含了一少部分页面的完整页表条目 Contains complete page table entries for small number of pages

地址翻译符号总结
Summary of Address Translation Symbols

- 基本参数 **Basic Parameters**
  - **N = 2^n** : Number of addresses in virtual address space 虚拟地址空间的地址个数
  - **M = 2^m** : Number of addresses in physical address space 物理地址空间的地址个数
  - **P = 2^p** : Page size (bytes) 页大小（字节）
- 虚拟地址VA划分 **Components of the virtual address (VA)**
  - **TLBI**: TLB index TLB索引
  - **TLBT**: TLB tag TLB标记
  - **VPO**: Virtual page offset 虚拟页内偏移
  - **VPN**: Virtual page number 虚拟页号
- 物理地址PA划分 **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO) 物理页内偏移（同VPO）
  - **PPN:** Physical page number 物理页号

访问TLB Accessing the TLB

- MMU使用虚拟地址的VPN部分访问TLB MMU uses the VPN portion of the virtual address to access the TLB:



T = 2^t sets 组

VPN

n-1 p+t p+t-1 p p-1 0

TLB tag (TLBT) | TLB index (TLBI) | VPO

TLBT在组内匹配行的标记

TLBT matches tag of line within set

TLBI选择组

TLBI selects the set

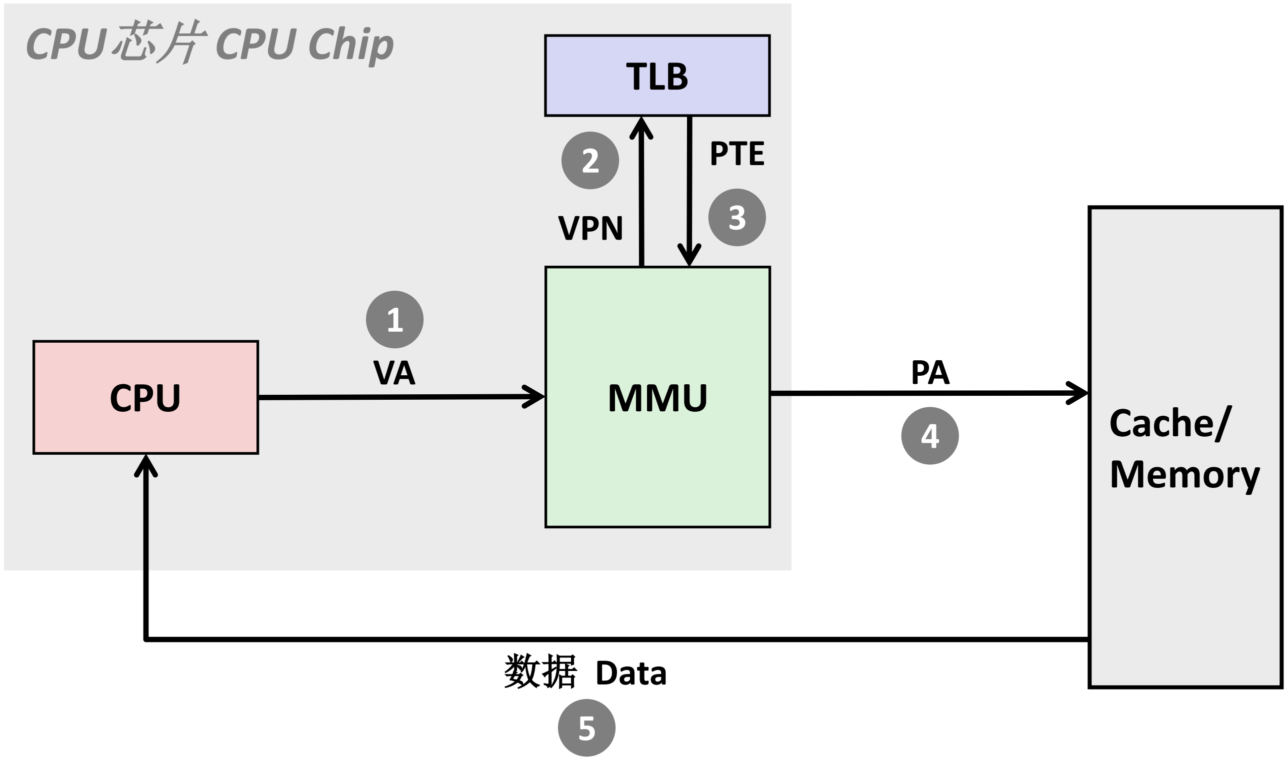
Set 0: v tag PTE | v tag PTE

Set 1: v tag PTE | v tag PTE

⋮

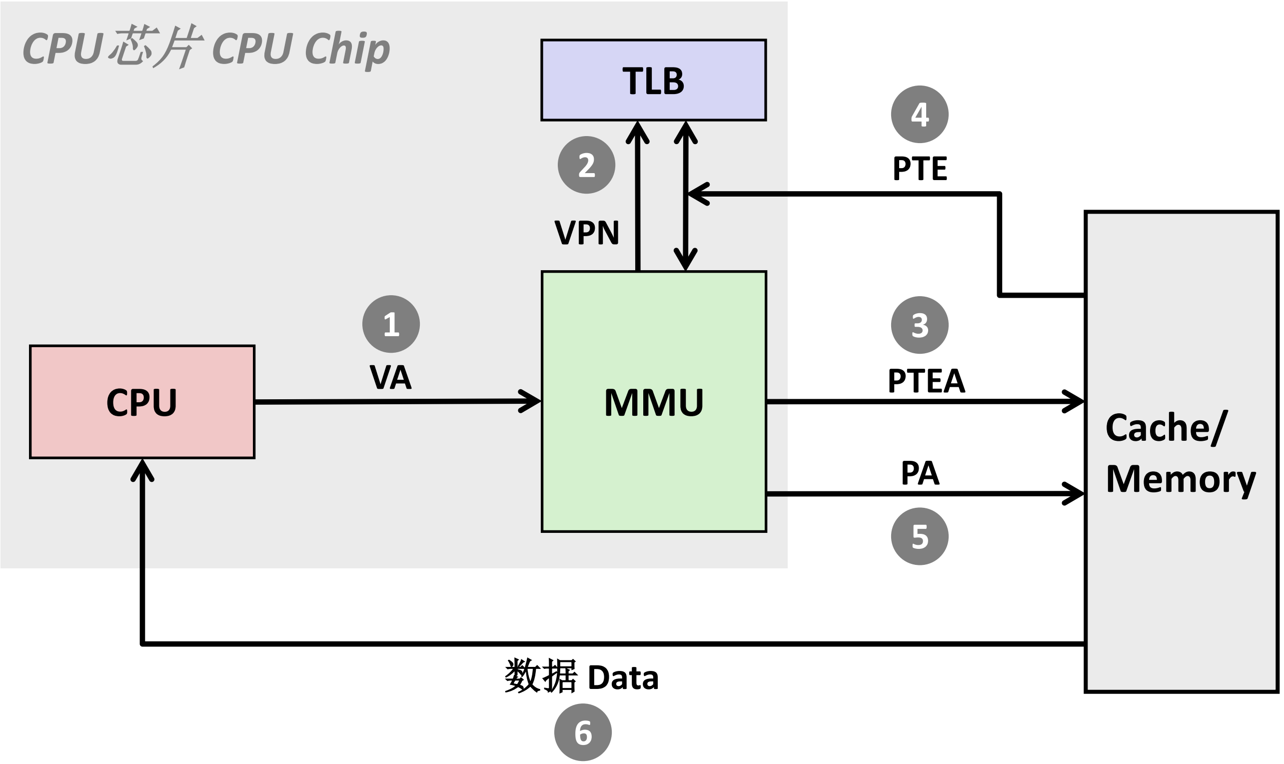
Set T-1: v tag PTE | v tag PTE

TLB命中 TLB Hit



TLB命中会减少一次内存访问 A TLB hit eliminates a memory access

TLB不命中 TLB Miss



TLB不命中会导致一个额外的内存访问（页表条目）

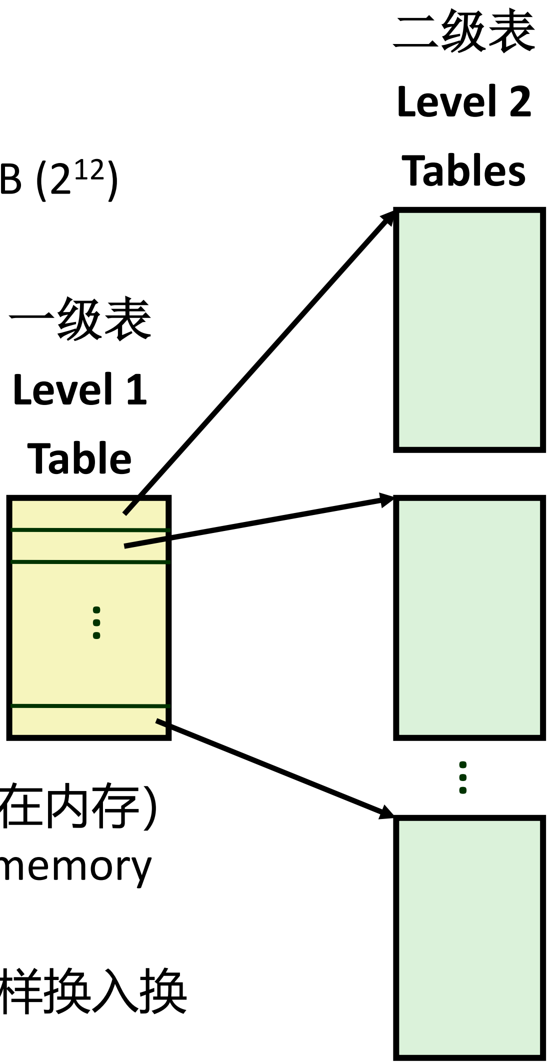
幸运的是，**TLB**不命中很少发生。为何？

A TLB miss incurs an additional memory access (the PTE)

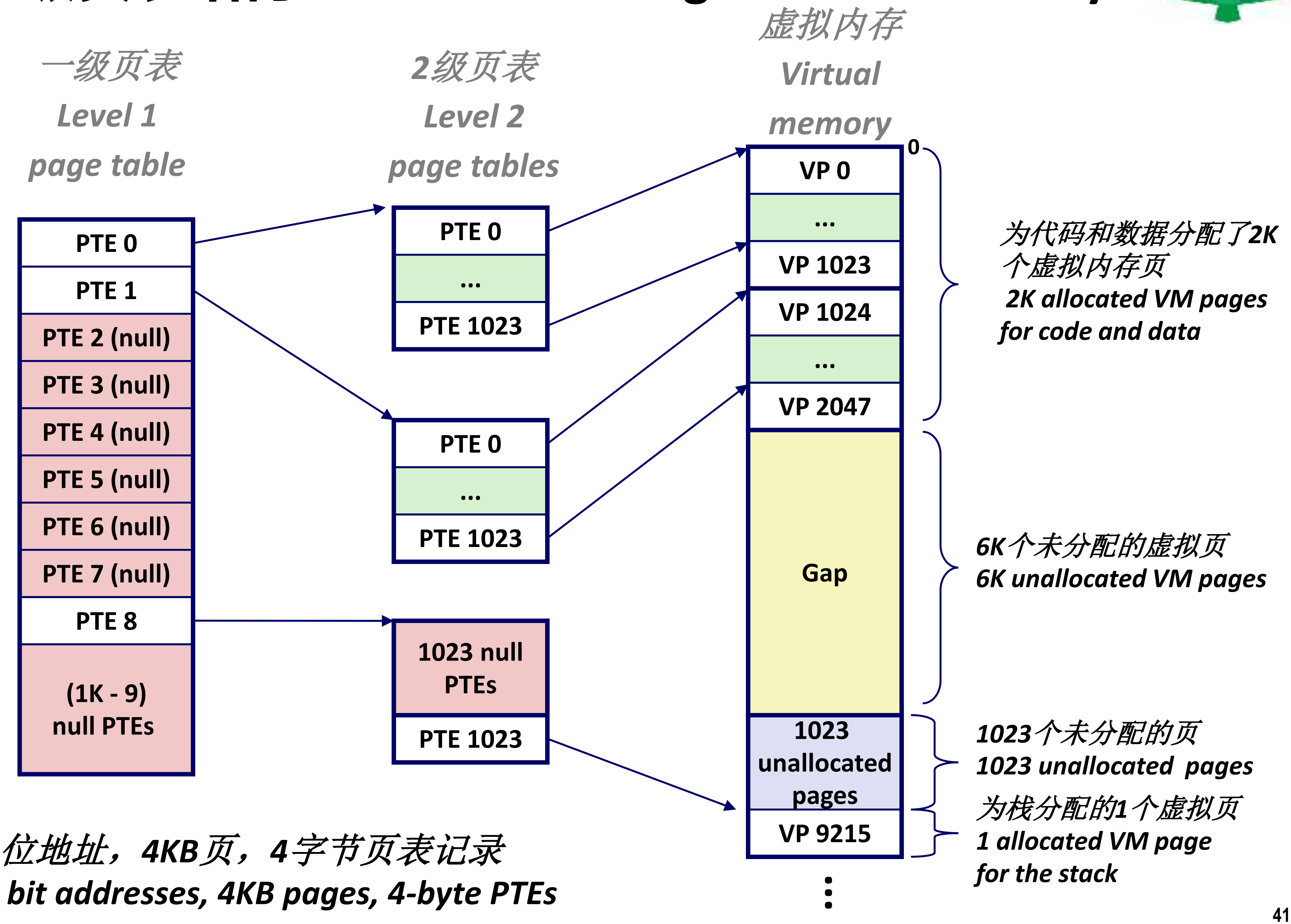
Fortunately, TLB misses are rare. Why?

多级页表 Multi-Level Page Tables

- **假设 Suppose:**
  - 4KB大小页表，48位地址空间，8字节页表记录 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- **问题 Problem:**
  - 页表占用的空间将高达512GB
  - Would need a 512 GB page table!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- **常见方法：多级页表 Common solution: Multi-level page table**
- **例如：2级页表 Example: 2-level page table**
  - 一级页表：每个页表记录指向一个页表（总是驻留在内存）Level 1 table: each PTE points to a page table (always memory resident)
  - 二级页表：每个页表记录指向一个页（像其他页一样换入换出） Level 2 table: each PTE points to a page (paged in and out like any other data)



二级页表结构 A Two-Level Page Table Hierarchy



一级页表
Level 1
page table

PTE 0
PTE 1
PTE 2 (null)
PTE 3 (null)
PTE 4 (null)
PTE 5 (null)
PTE 6 (null)
PTE 7 (null)
PTE 8
(1K - 9)
null PTEs

2级页表
Level 2
page tables

PTE 0
...
PTE 1023

PTE 0
...
PTE 1023

1023 null
PTEs
PTE 1023

虚拟内存
Virtual
memory

0
VP 0
...
VP 1023
VP 1024
...
VP 2047
Gap
1023
unallocated
pages
VP 9215
⋮

为代码和数据分配了2K个虚拟内存页
2K allocated VM pages for code and data

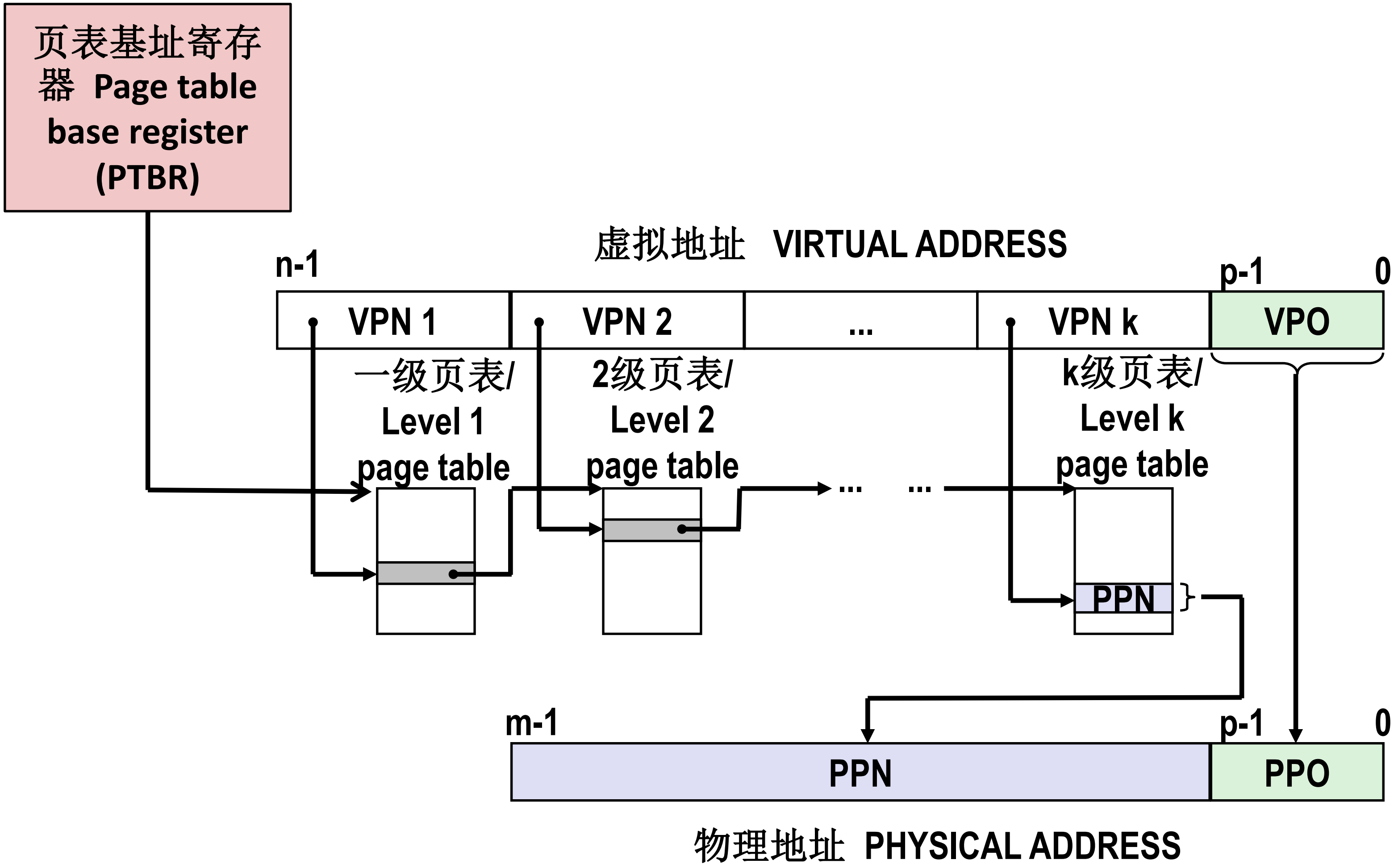
6K个未分配的虚拟页
6K unallocated VM pages

1023个未分配的页
1023 unallocated pages

为栈分配的1个虚拟页
1 allocated VM page for the stack

32位地址，4KB页，4字节页表记录
32 bit addresses, 4KB pages, 4-byte PTEs

k级页表的地址翻译
Translating with a k-level Page Table



总结 Summary

- 程序员眼中的虚拟内存 **Programmer's view of virtual memory**
  - 每个进程都有各自私有的线性地址空间 Each process has its own private linear address space
  - 不能被其他进程破坏 Cannot be corrupted by other processes
- 系统眼中的虚拟内存 **System view of virtual memory**
  - 通过缓存虚拟内存页高效地使用内存 Uses memory efficiently by caching virtual memory pages
    - 高效是因为局部性 Efficient only because of locality
  - 简化内存管理和编程 Simplifies memory management and programming
  - 通过提供方便的库打桩点来检查权限，简化了保护 Simplifies protection by providing a convenient interpositioning point to check permissions

Virtual Memory: Systems
虚拟内存:系统

100076202： 计算机系统导论

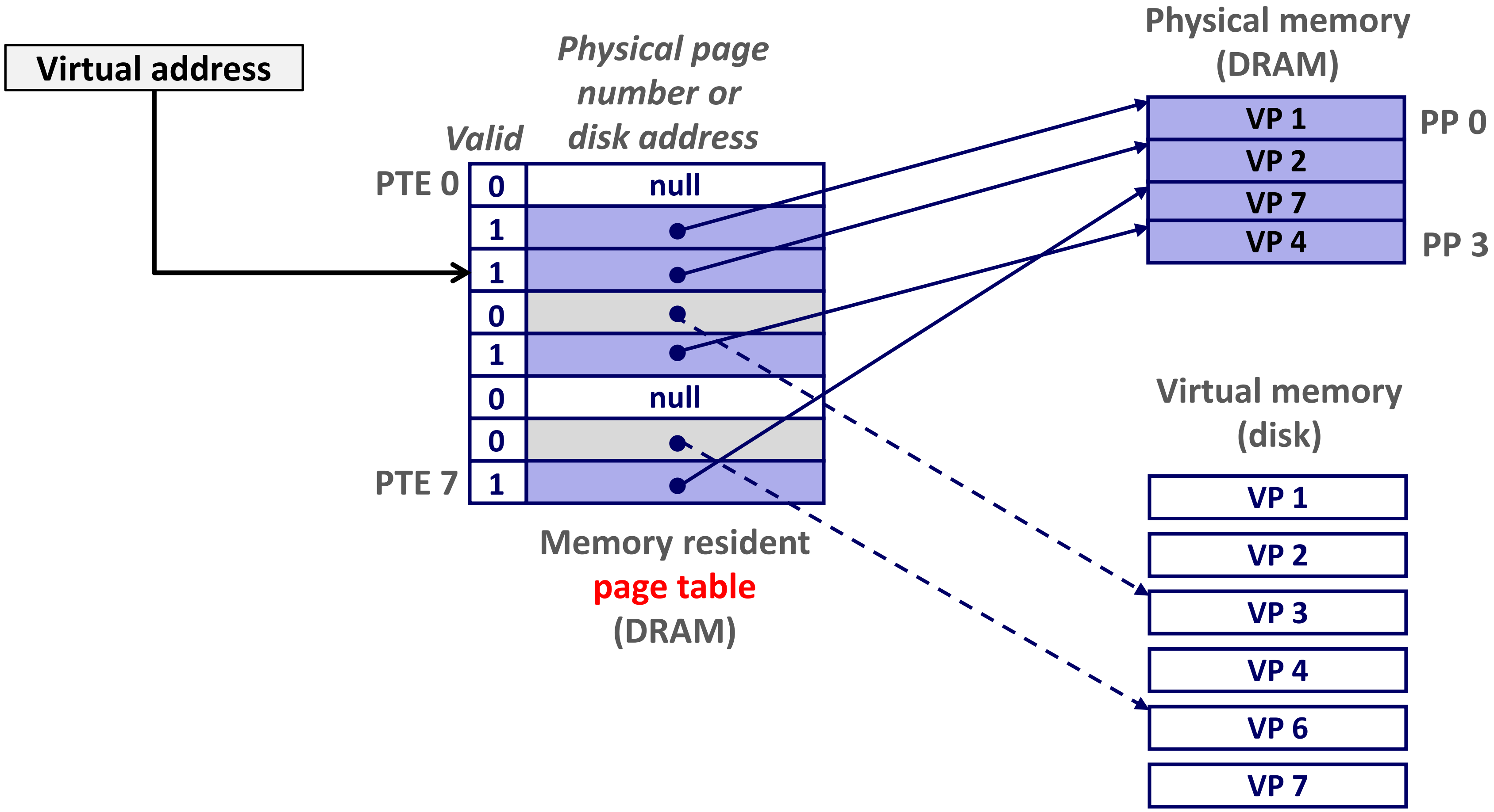
任课教师:

宿红毅　张艳　黎有琦　颜珂

原作者:

Randal E. Bryant and David R. O’Hallaron

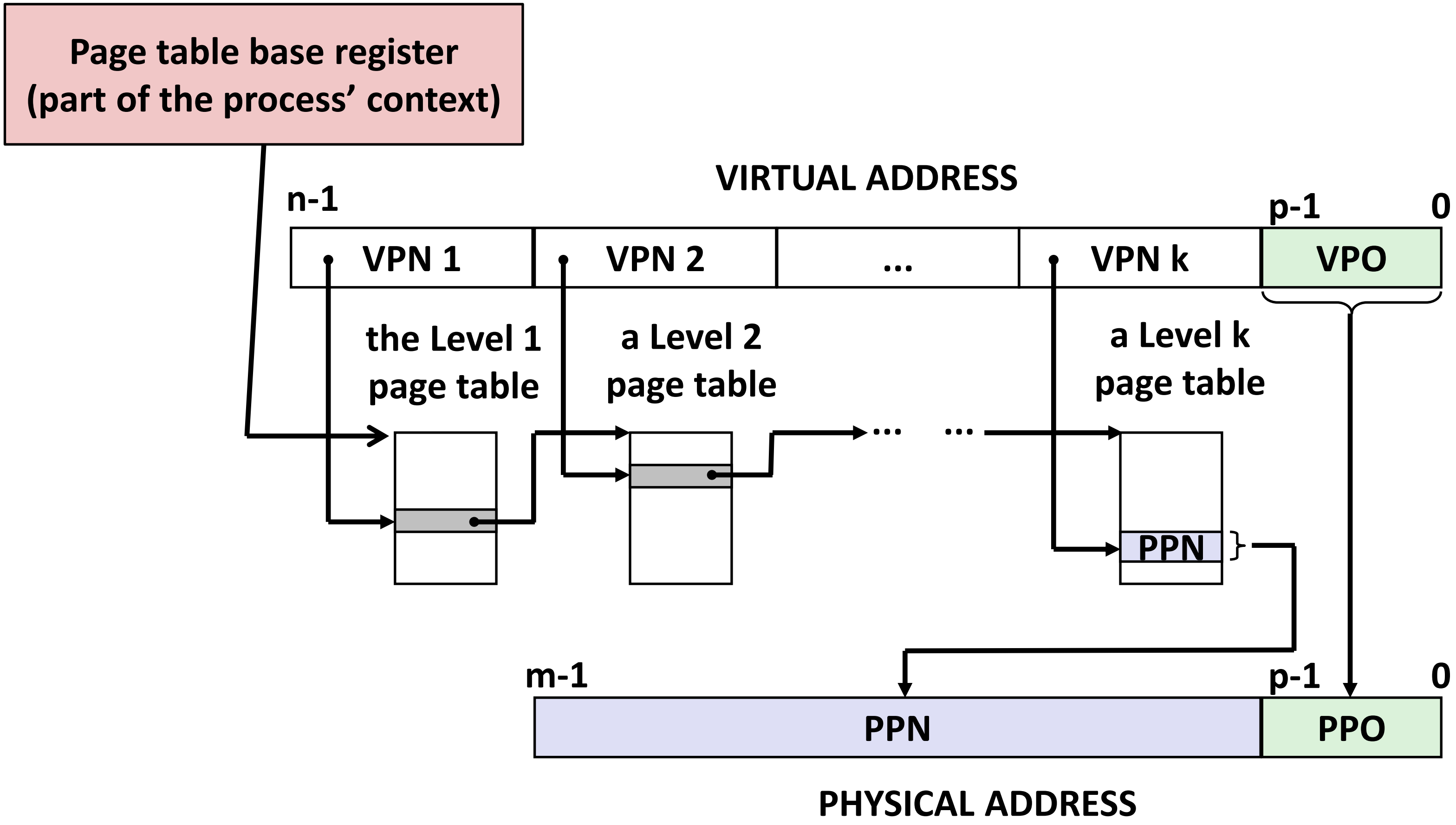
Review: Virtual Memory & Physical Memory



- A ***page table*** **contains page table entries (PTEs) that map virtual pages to physical pages.**

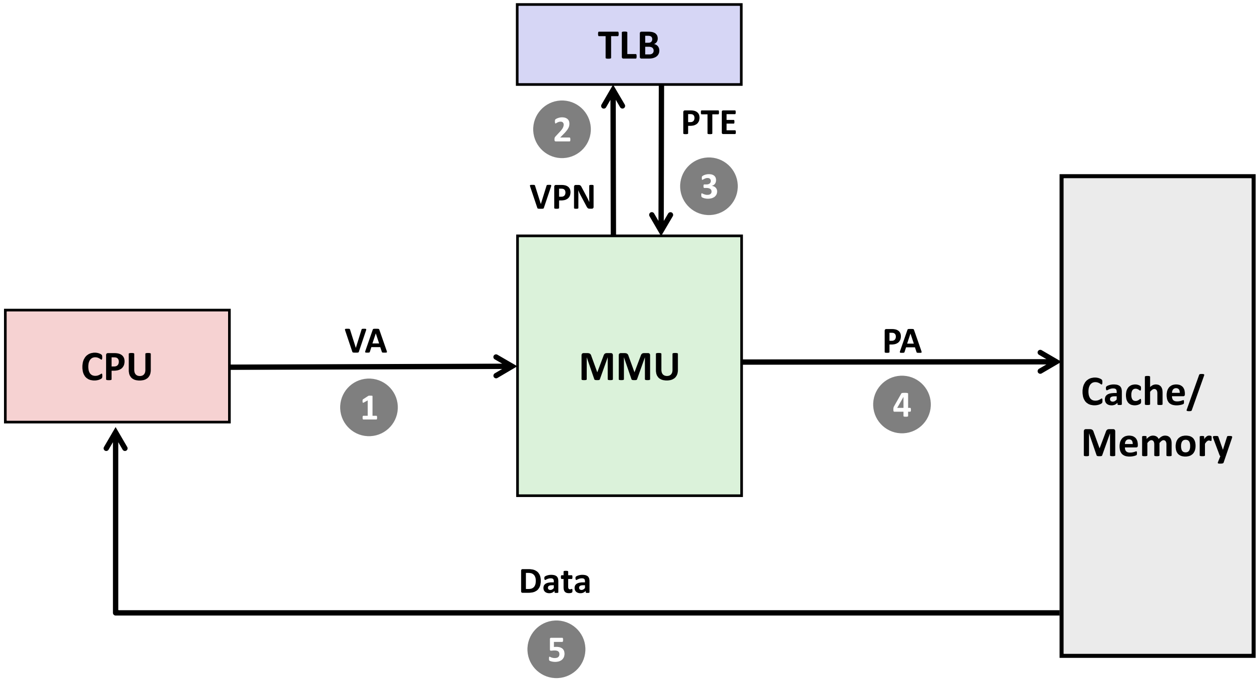
Translating with a k-level Page Table

- **Having multiple levels greatly reduces page table size**



Translation Lookaside Buffer (TLB)

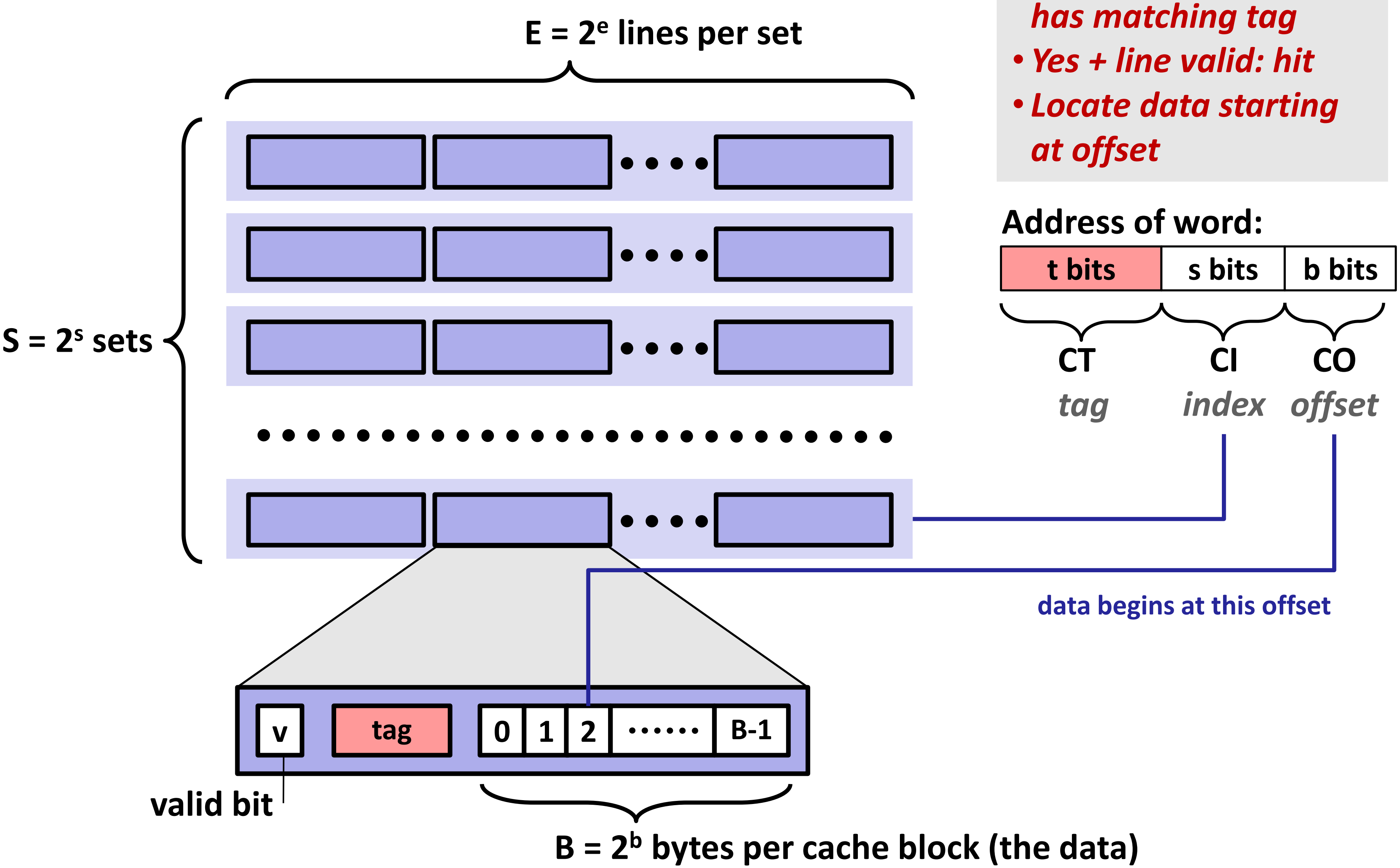
- **A small cache of page table entries with fast access by MMU**



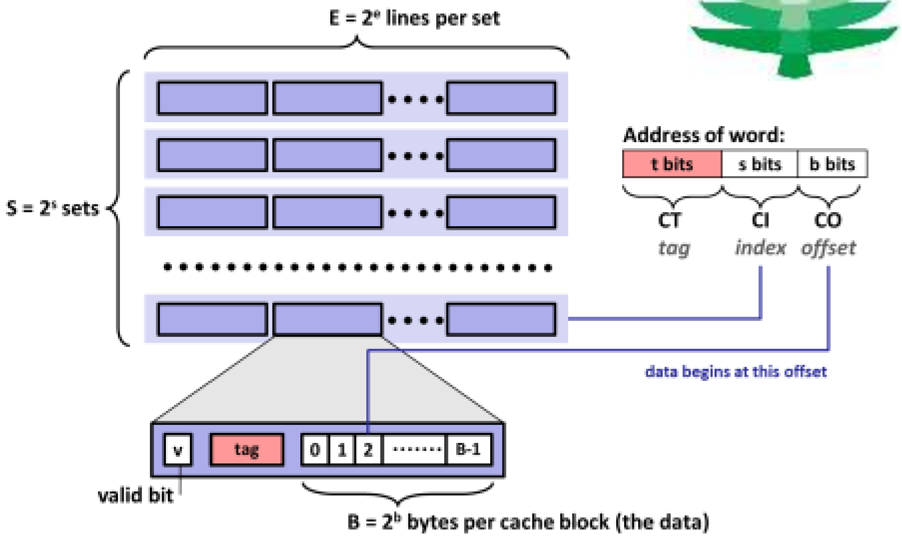
Typically, a TLB hit eliminates the k memory accesses required to do a page table lookup.

Recall: Set Associative Cache

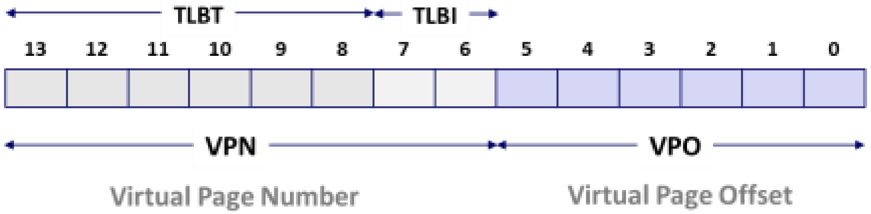
Steps for a READ:
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



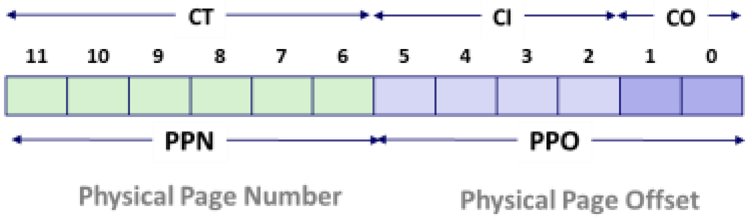
Review of Symbols



- **Basic Parameters**
  - **$N = 2^n$** : Number of addresses in virtual address space
  - **$M = 2^m$** : Number of addresses in physical address space
  - **$P = 2^p$** : Page size (bytes)
- **Components of the *virtual address* (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **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
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag



(bits per field for our simple example)



议题 Today

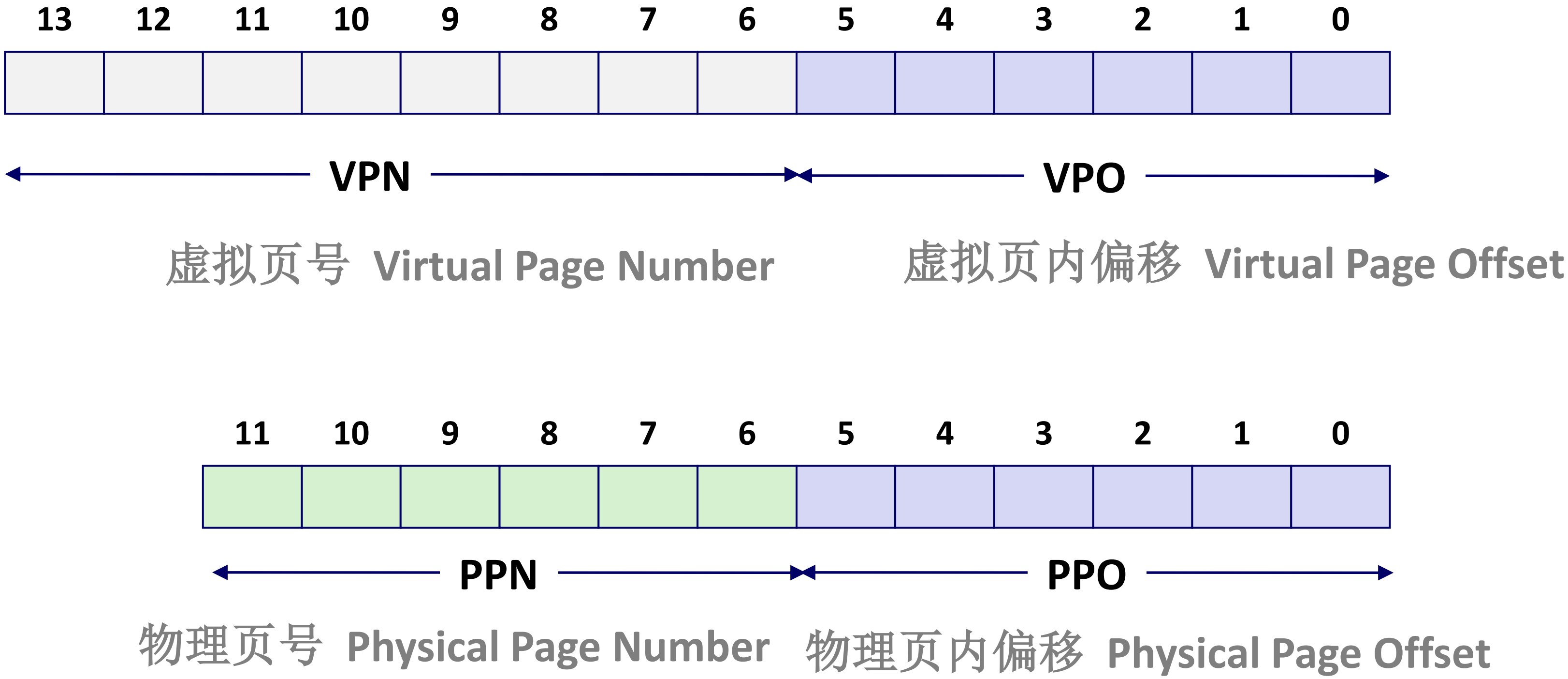
- **简单内存系统示例 Simple memory system example** CSAPP 9.6.4
- 案例研究：Core i7/Linux内存系统 Case study: Core i7/Linux memory system CSAPP 9.7
- 内存映射 Memory mapping CSAPP 9.8

符号回顾 Review of Symbols

- 基本参数 **Basic Parameters**
  - **N = 2^n** : Number of addresses in virtual address space 虚拟地址空间的地址数量
  - **M = 2^m** : Number of addresses in physical address space 物理地址空间的地址数量
  - **P = 2^p** : Page size (bytes) 页大小（字节）
- 虚拟地址VA划分 **Components of the virtual address (VA)**
  - **TLBI**: TLB index TLB索引
  - **TLBT**: TLB tag TLB标记
  - **VPO**: Virtual page offset 虚拟页内偏移
  - **VPN**: Virtual page number 虚拟页号
- 物理地址PA划分 **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO) 物理页内偏移（同VPO）
  - **PPN:** Physical page number 物理页号
  - **CO**: Byte offset within cache line Cache行中的偏移
  - **CI:** Cache index Cache索引
  - **CT**: Cache tag Cache标记

简单的内存系统示例
Simple Memory System Example

- **寻址 Addressing**
  - 14位虚拟地址 14-bit virtual addresses
  - 12位物理地址 12-bit physical address
  - 页大小为64字节 Page size = 64 bytes



虚拟页号 Virtual Page Number

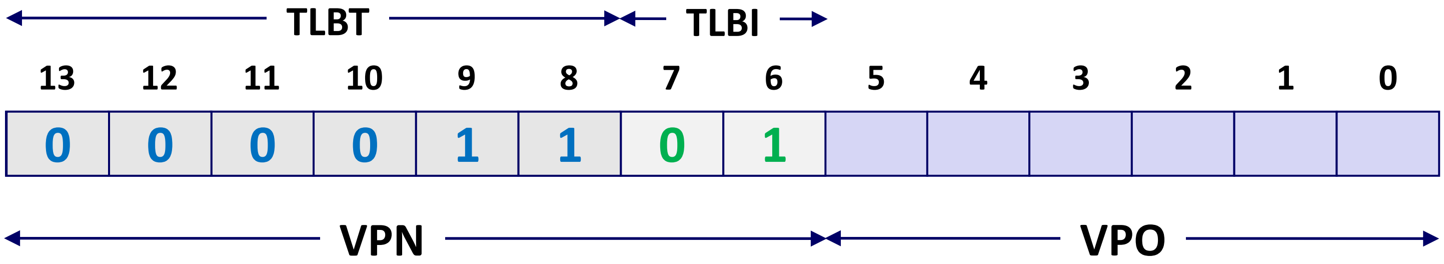
虚拟页内偏移 Virtual Page Offset

物理页号 Physical Page Number

物理页内偏移 Physical Page Offset

简单内存系统TLB
Simple Memory System TLB

- 16个条目 16 entries
- 4路组相联 4-way associative



TLBT TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1						

VPN VPO

VPN = 0b1101 = 0x0D

翻译后备缓冲区（TLB） Translation Lookaside Buffer (TLB)

Set	*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

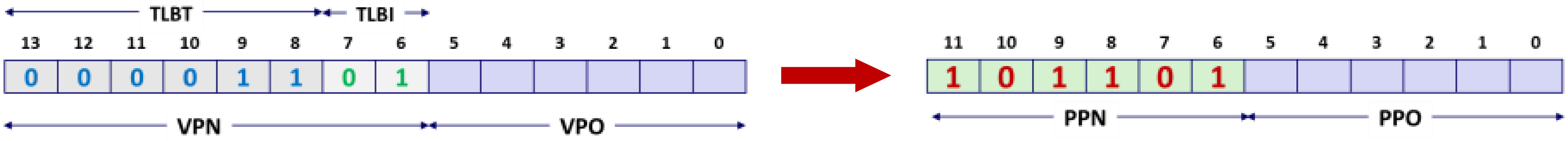
简单内存系统页表
Simple Memory System Page Table

只显示了前**16**个条目（**256**个条目） Only showing the first 16 entries (out of 256)

VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

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

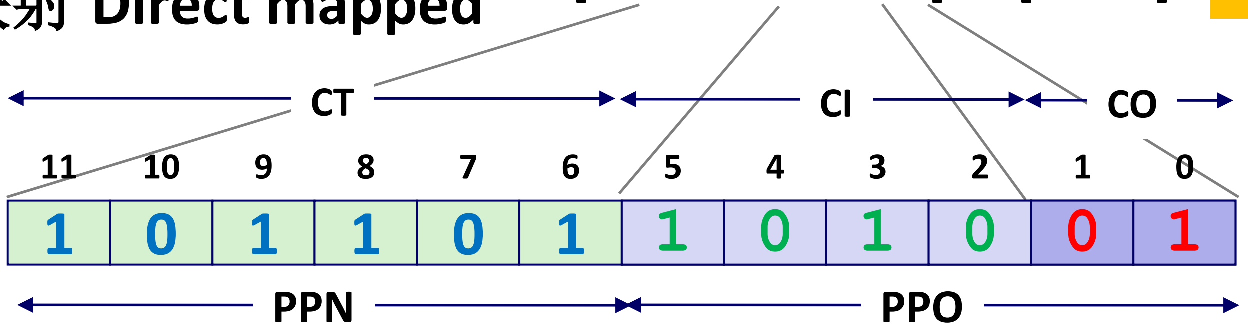
0x0D → 0x2D



简单内存系统的Cache
Simple Memory System Cache

- 16行，4字节cache行大小 16 lines, 4-byte cache line size
- 物理地址 Physically addressed
- 直接映射 Direct mapped

V[0b00001101101001] = V[0x369]
P[0b101101101001] = P[0xB69] = 0x15



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	–	–	–	–

地址翻译示例 Address Translation Example

虚拟地址 Virtual Address: 0x03D4

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO					

VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

TLB

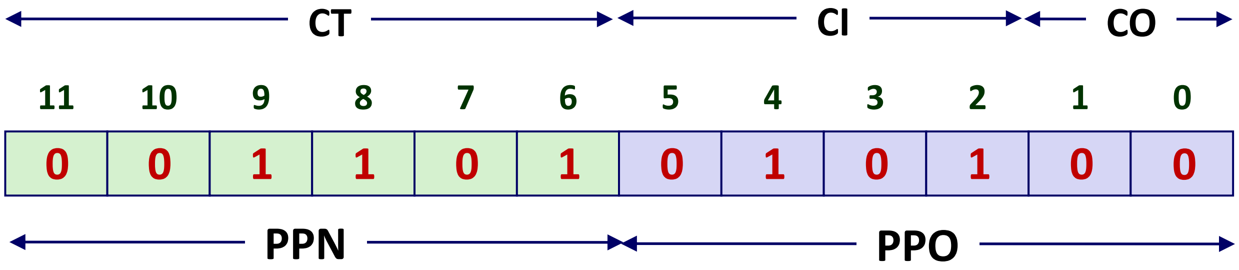
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

物理地址 Physical Address

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0
PPN						PPO					

地址翻译示例 Address Translation Example

物理地址 Physical Address



	CT						CI				CO	
Bit	11	10	9	8	7	6	5	4	3	2	1	0
Value	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					

CO __0__ CI __0x5__ CT __0x0D__ Hit? __Y__ Byte: __0x36__

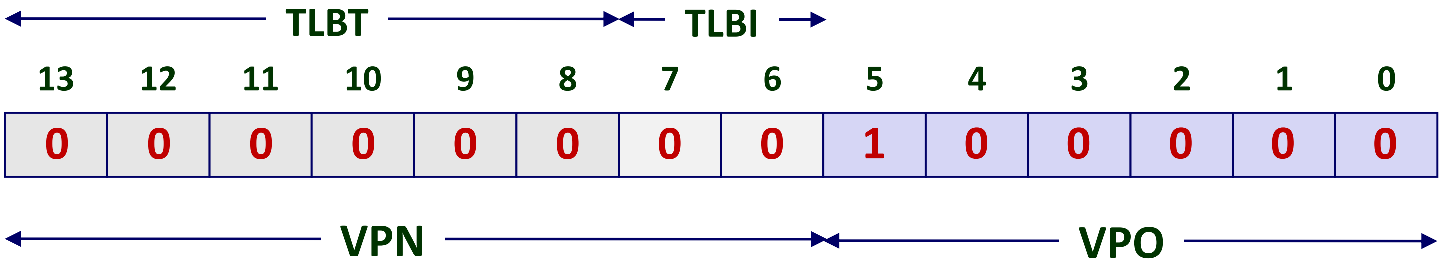
Cache

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	–	–	–	–

地址翻译示例：TLB/Cache不命中
Address Translation Example: TLB/Cache Miss

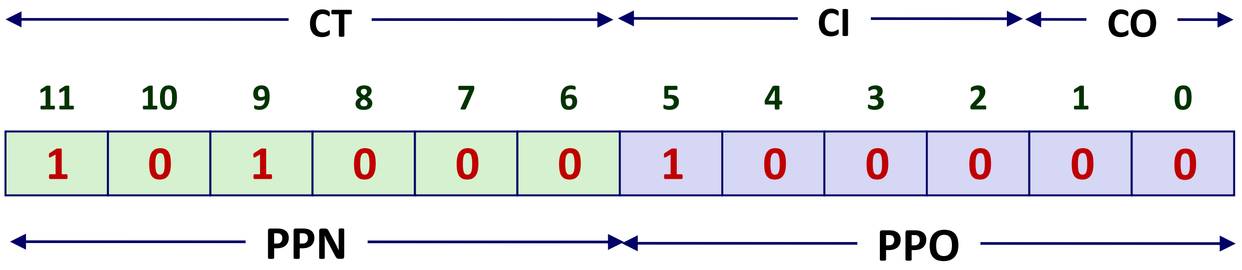
虚拟地址 Virtual Address: 0x0020



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

物理地址 Physical Address



11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

CO 0 CI 0x8 CT 0x28 Hit? __ Byte: _____

Page table

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

地址翻译示例：TLB/Cache不命中
Address Translation Example: TLB/Cache Miss

Cache

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	–	–	–	–

物理地址 Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0
PPN						PPO					

CO 0

CI 0x8

CT 0x28

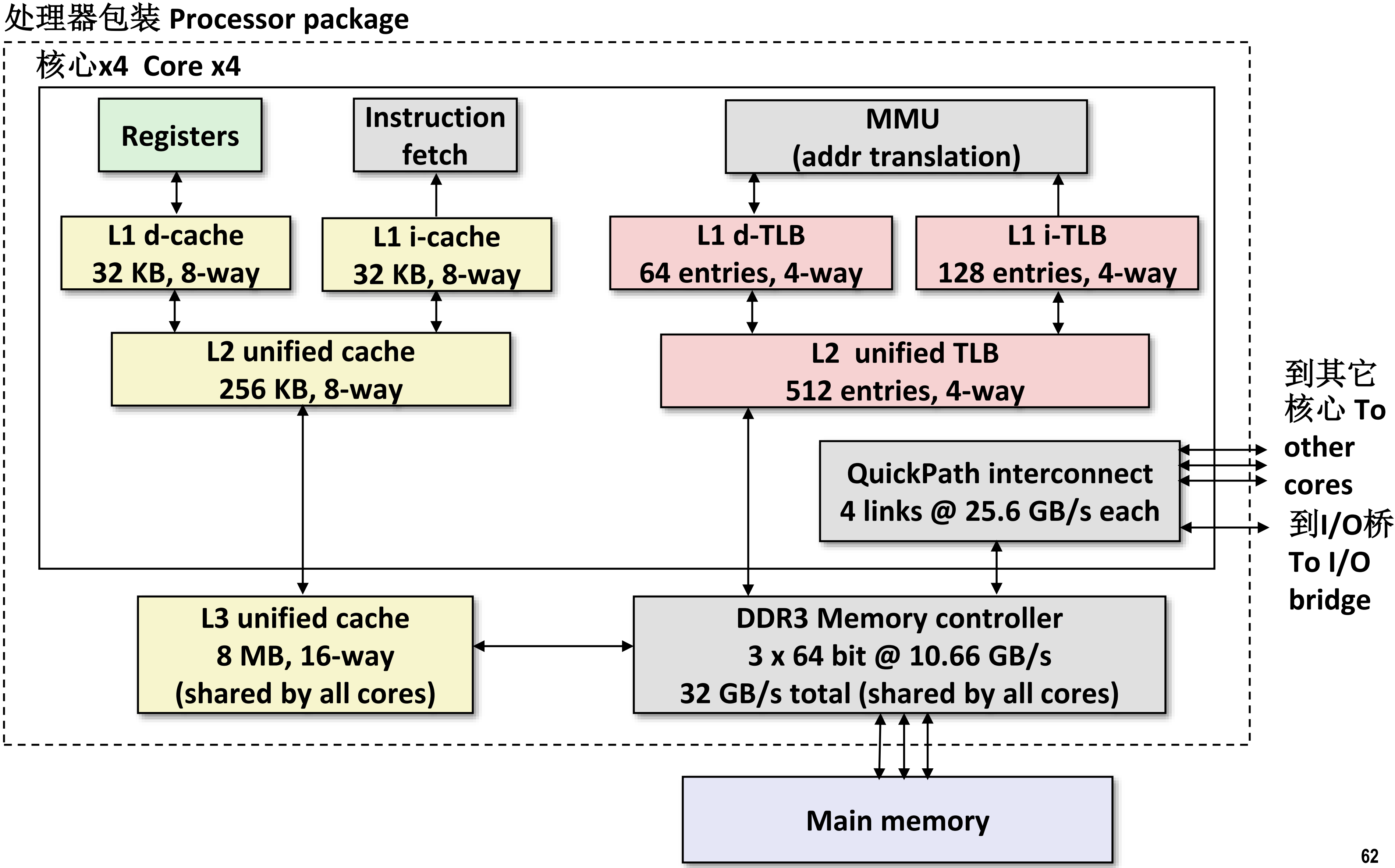
Hit? N

Byte: Mem

议题 Today

- 简单内存系统示例 Simple memory system example
- 案例研究：Core i7/Linux内存系统 Case study: Core i7/Linux memory system
- 内存映射 Memory mapping

Intel Core i7存储系统
Intel Core i7 Memory System



处理器包装 Processor package

核心x4 Core x4

Registers

Instruction fetch

MMU
(addr translation)

L1 d-cache
32 KB, 8-way

L1 i-cache
32 KB, 8-way

L1 d-TLB
64 entries, 4-way

L1 i-TLB
128 entries, 4-way

L2 unified cache
256 KB, 8-way

L2 unified TLB
512 entries, 4-way

QuickPath interconnect
4 links @ 25.6 GB/s each

到其它核心 To other cores

到I/O桥 To I/O bridge

L3 unified cache
8 MB, 16-way
(shared by all cores)

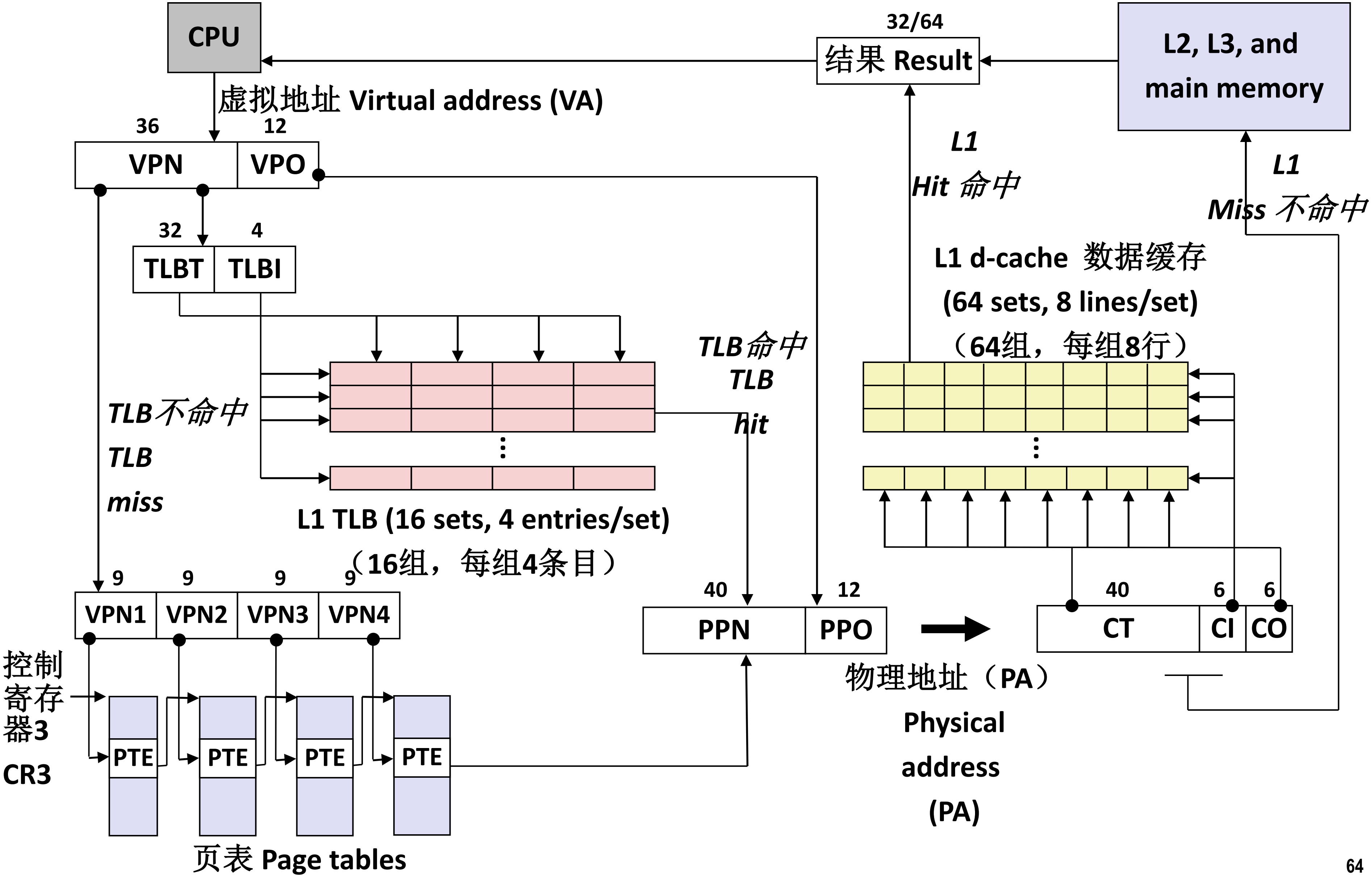
DDR3 Memory controller
3 x 64 bit @ 10.66 GB/s
32 GB/s total (shared by all cores)

Main memory

符号回顾 Review of Symbols

- **基本参数 Basic Parameters**
  - **$N = 2^n$** : Number of addresses in virtual address space 虚拟地址空间的地址数量
  - **$M = 2^m$** : Number of addresses in physical address space 物理地址空间的地址数量
  - **$P = 2^p$** : Page size (bytes) 页大小（字节）
- **虚拟地址VA划分 Components of the virtual address (VA)**
  - **TLBI**: TLB index TLB索引
  - **TLBT**: TLB tag TLB标记
  - **VPO**: Virtual page offset 虚拟页内偏移
  - **VPN**: Virtual page number 虚拟页号
- **物理地址PA划分 Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO) 物理页内偏移（同VPO）
  - **PPN:** Physical page number 物理页号
  - **CO**: Byte offset within cache line Cache行中的偏移
  - **CI:** Cache index Cache索引
  - **CT**: Cache tag Cache标记

端到端Core i7地址翻译
End-to-end Core i7 Address Translation



Core i7 1-3级页表条目
Core i7 Level 1-3 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	页表物理基地址 Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

操作系统可用（页表位于磁盘上）Available for OS (page table location on disk)	P=0

每个条目对应一个4k子页表，主要的字段包括： Each entry references a 4K child page table. Significant fields:

P:子页表是否在物理内存 Child page table present in physical memory (1) or not (0).

R/W: 只读或者读写权限标记位 Read-only or read-write access permission for all reachable pages.

U/S: 用户或特权（内核）模式标记位 user or supervisor (kernel) mode access permission for all reachable pages.

WT: 子页表的写直达或者写回Cache策略 Write-through or write-back cache policy for the child page table.

A: 引用标记(由MMU读写时设置，软件清除) Reference bit (set by MMU on reads and writes, cleared by software).

PS: 页面大小，4KB或者4MB(仅为1级PTE定义) Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

Page table physical base address: 物理页表地址的高40位(强制页表按照4KB对齐) 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: 禁止或允许取指操作 Disable or enable instruction fetches from all pages reachable from this PTE.

Core i7 4级页表条目
Core i7 Level 4 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	物理页基址/ Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1
操作系统可见（内存页位于磁盘）Available for OS (page location on disk)												P=0

每个条目对应一个4k子页表，主要的字段包括： Each entry references a 4K child page. Significant fields:

P:子页表是否在物理内存 Child page table present in physical memory (1) or not (0).

R/W: 只读或者读写权限标记位 Read-only or read-write access permission for all reachable pages.

U/S: 用户或特权（内核）模式标记位 user or supervisor (kernel) mode access permission for all reachable pages.

WT: 子页表的写直达或者写回Cache策略 Write-through or write-back cache policy for the child page table.

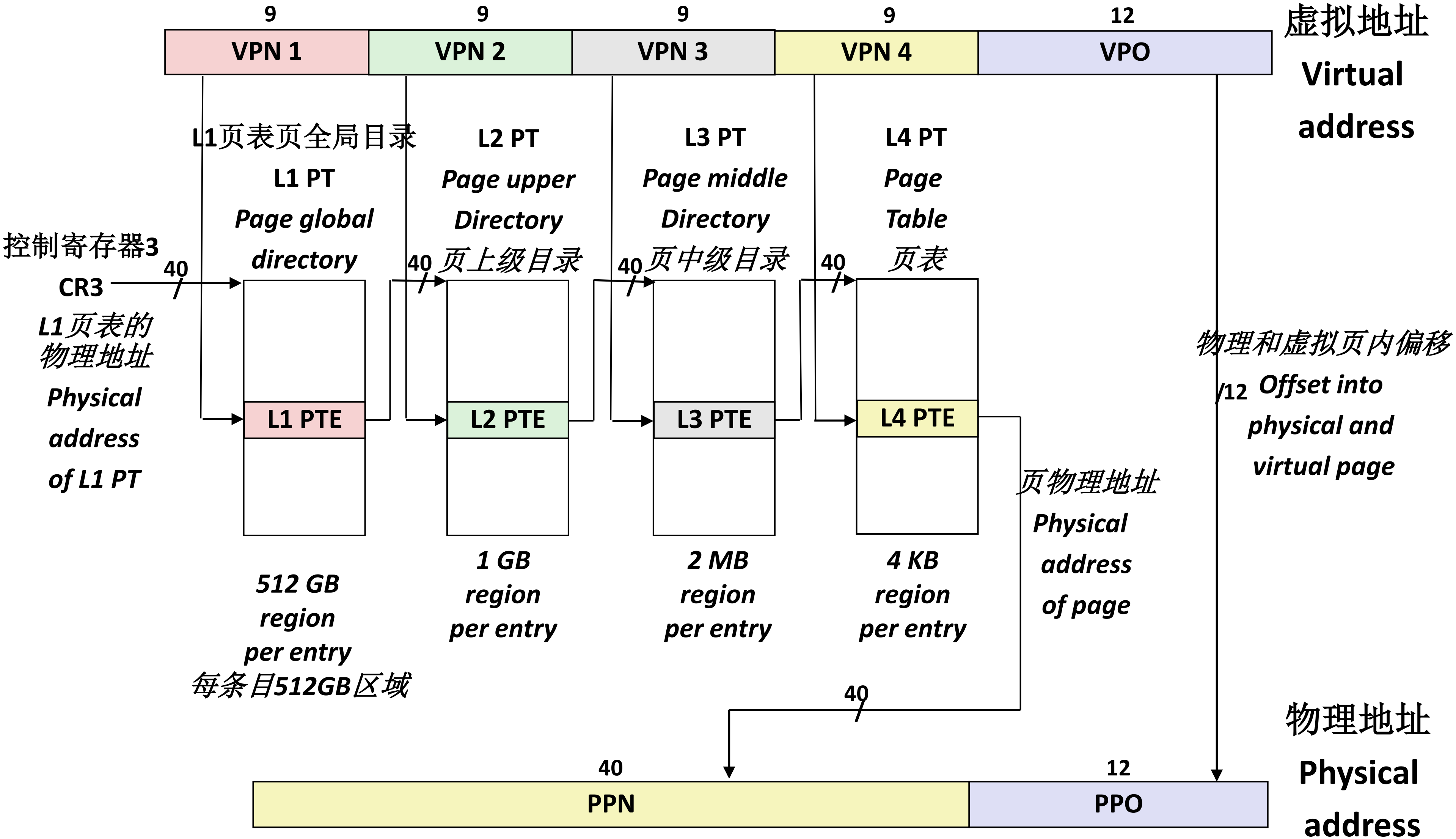
A: 引用标记(由MMU读写时设置，软件清除)/Reference bit(set by MMU on reads and writes, cleared by software).

D: 脏位(写操作时由MMU设置，软件清除) Dirty bit (set by MMU on writes, cleared by software)

Page physical base address:物理页表地址的高40位(强制页表按照4KB对齐) 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD:禁止或允许取指操作 Disable or enable instruction fetches from this page.

Core i7页表翻译
Core i7 Page Table Translation



9 9 9 9 12

VPN 1 | VPN 2 | VPN 3 | VPN 4 | VPO

虚拟地址

Virtual address

L1页表页全局目录

L1 PT

Page global directory

L2 PT

Page upper Directory

页上级目录

L3 PT

Page middle Directory

页中级目录

L4 PT

Page Table

页表

控制寄存器3

CR3

L1页表的物理地址

Physical address of L1 PT

40 40 40 40

L1 PTE

L2 PTE

L3 PTE

L4 PTE

512 GB region per entry

每条目512GB区域

1 GB region per entry

2 MB region per entry

4 KB region per entry

页物理地址

Physical address of page

物理和虚拟页内偏移

12 *Offset into physical and virtual page*

40

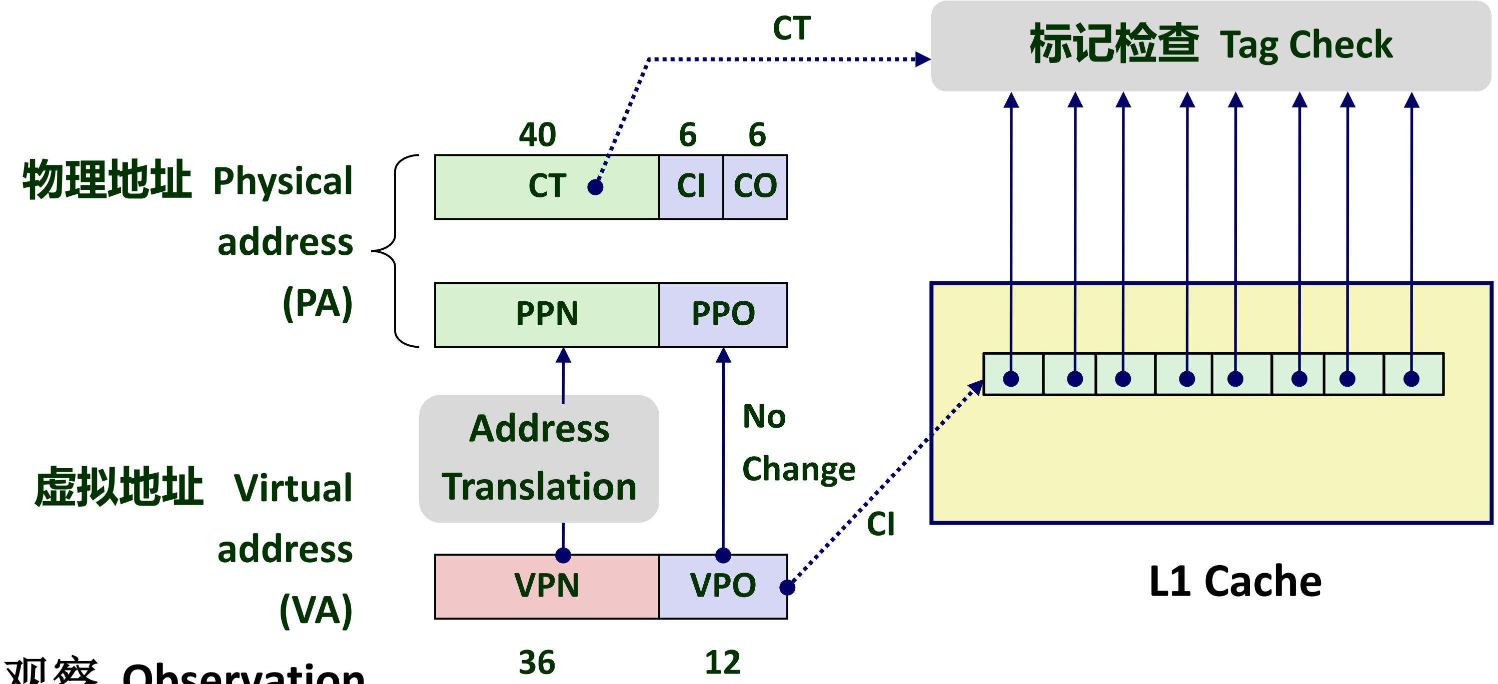
40 12

PPN | PPO

物理地址

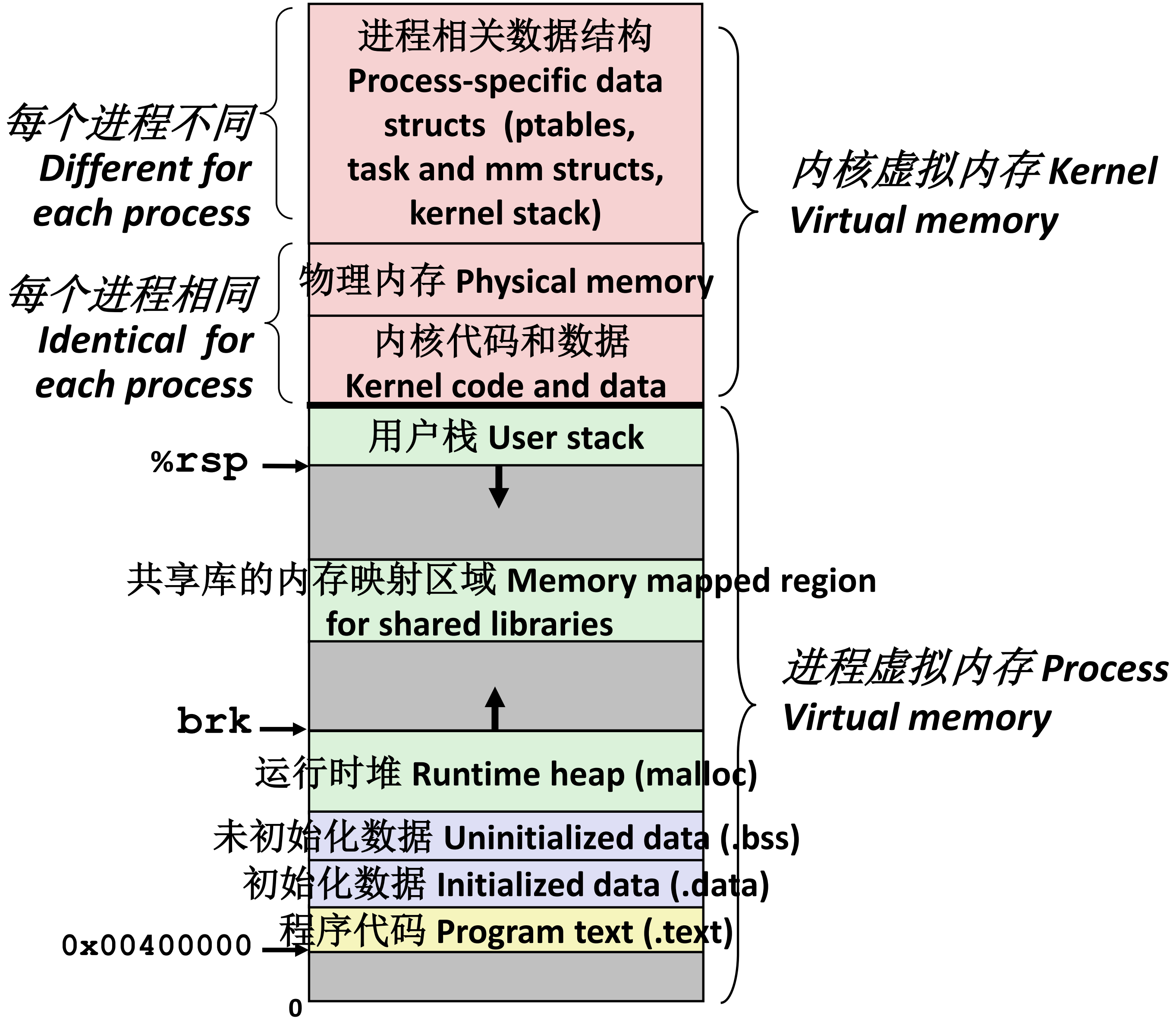
Physical address

L1访问加速小技巧
Cute Trick for Speeding Up L1 Access

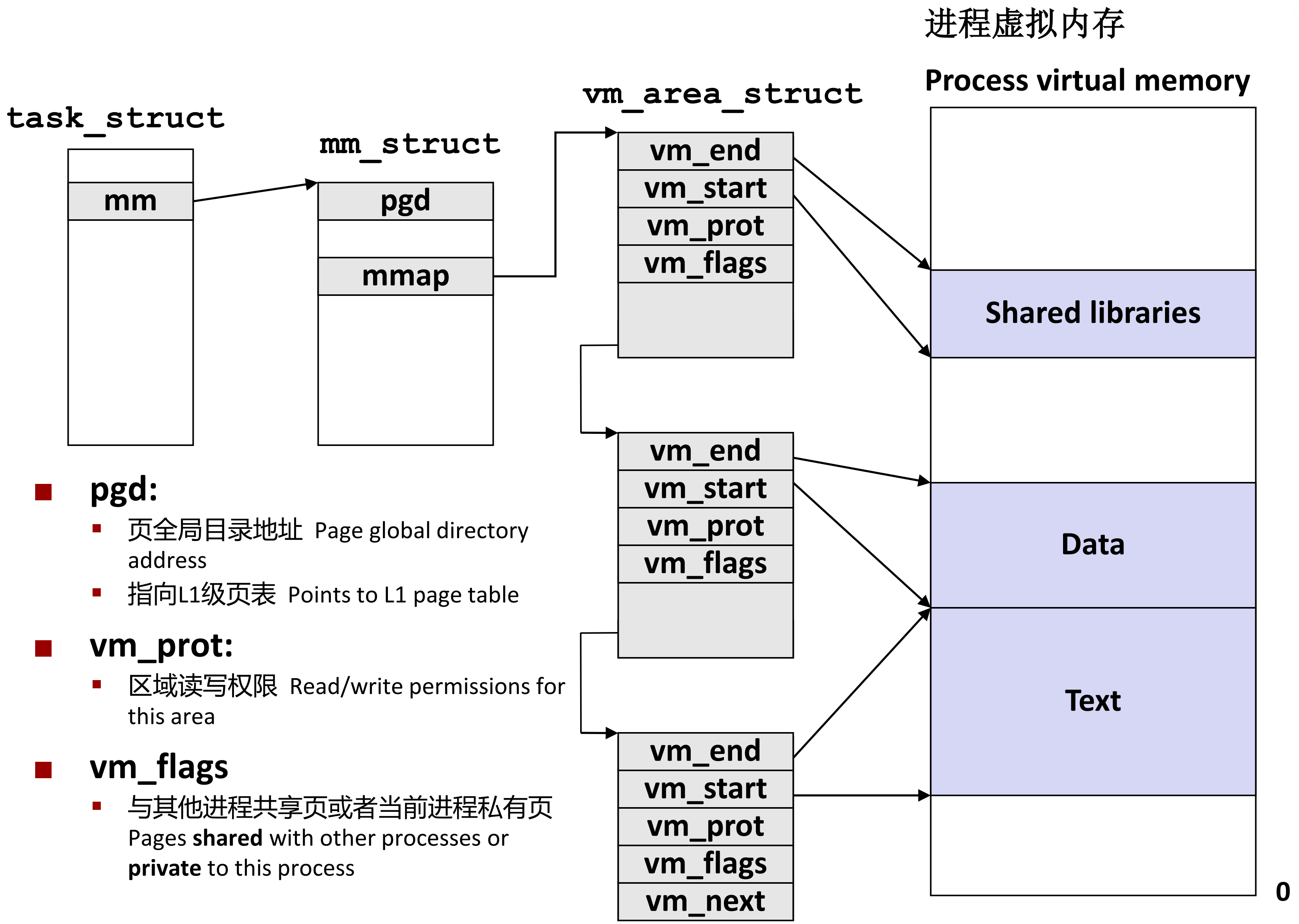


- 观察 **Observation**
  - 虚拟地址和物理地址中用于Cache索引的位是相同的 Bits that determine CI identical in virtual and physical address
  - 地址翻译的同时可以进行Cache索引 Can index into cache while address translation taking place
  - 通常情况下TLB会命中，PPN（Cache标记）接下来会可用 Generally we hit in TLB, so PPN bits (CT bits) available next
  - 虚拟索引，物理标记 “Virtually indexed, physically tagged”
  - Cache大小设计需要注意才能这样并行做 Cache carefully sized to make this possible

Linux进程的虚拟地址空间
Virtual Address Space of a Linux Process



Linux将虚拟内存组织为一些区域的集合
Linux Organizes VM as Collection of "Areas"



进程虚拟内存

Process virtual memory

`task_struct`

mm

`mm_struct`

pgd

mmap

`vm_area_struct`

vm_end
vm_start
vm_prot
vm_flags

vm_end
vm_start
vm_prot
vm_flags

vm_end
vm_start
vm_prot
vm_flags
vm_next

Shared libraries

Data

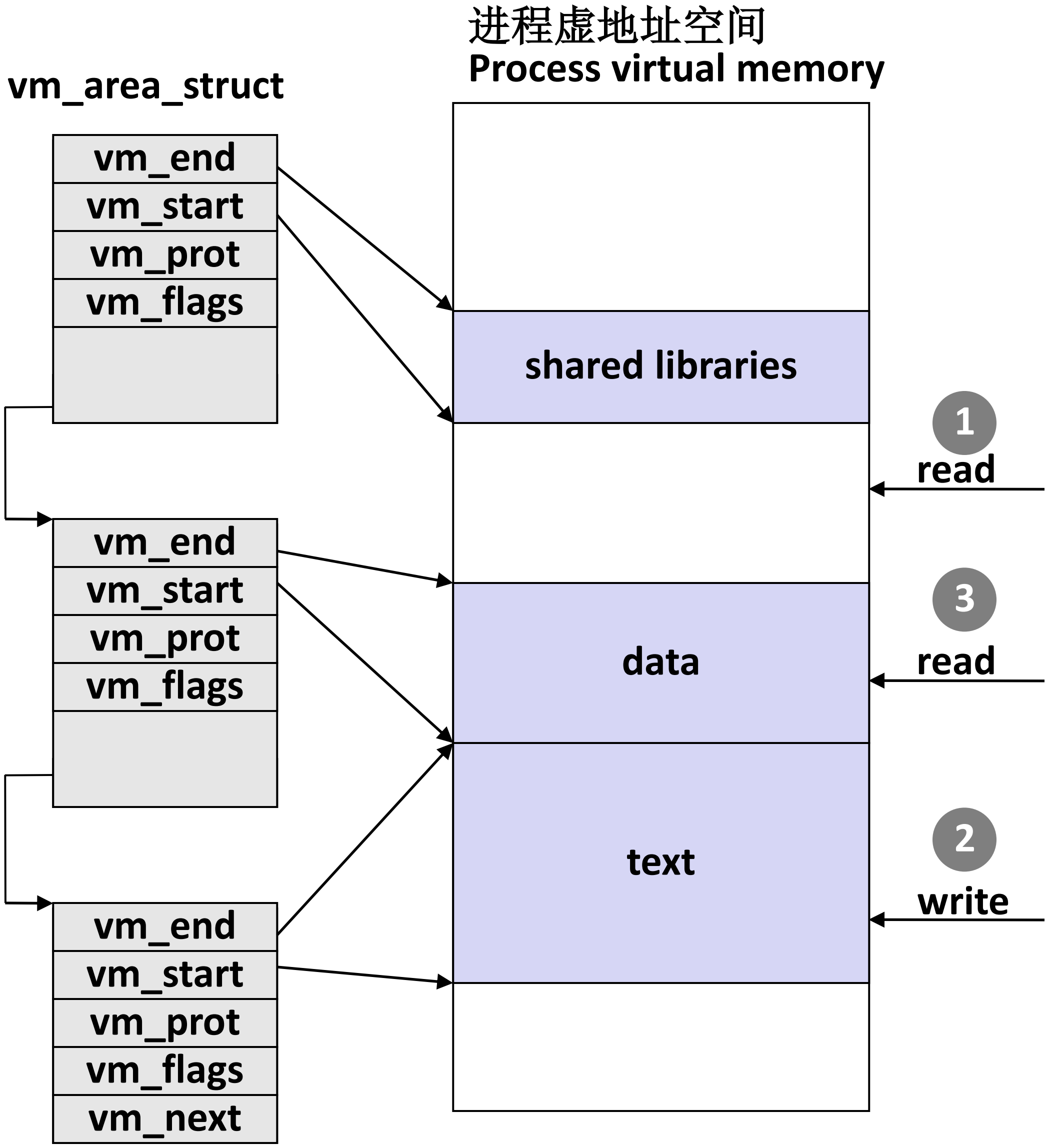
Text

0

- **pgd:**
  - 页全局目录地址 Page global directory address
  - 指向L1级页表 Points to L1 page table
- **vm_prot:**
  - 区域读写权限 Read/write permissions for this area
- **vm_flags**
  - 与其他进程共享页或者当前进程私有页 Pages **shared** with other processes or **private** to this process

每个进程有自己的task_struct等 Each process has own **`task_struct`**, etc

Linux中的缺页中断处理
Linux Page Fault Handling



段错误 Segmentation fault:
访问不存在的页
accessing a non-existing page

普通缺页中断
Normal page fault

保护异常 Protection exception:
例如，对只读页进行违规写操作（Linux报告为段错误） e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

议题 Today

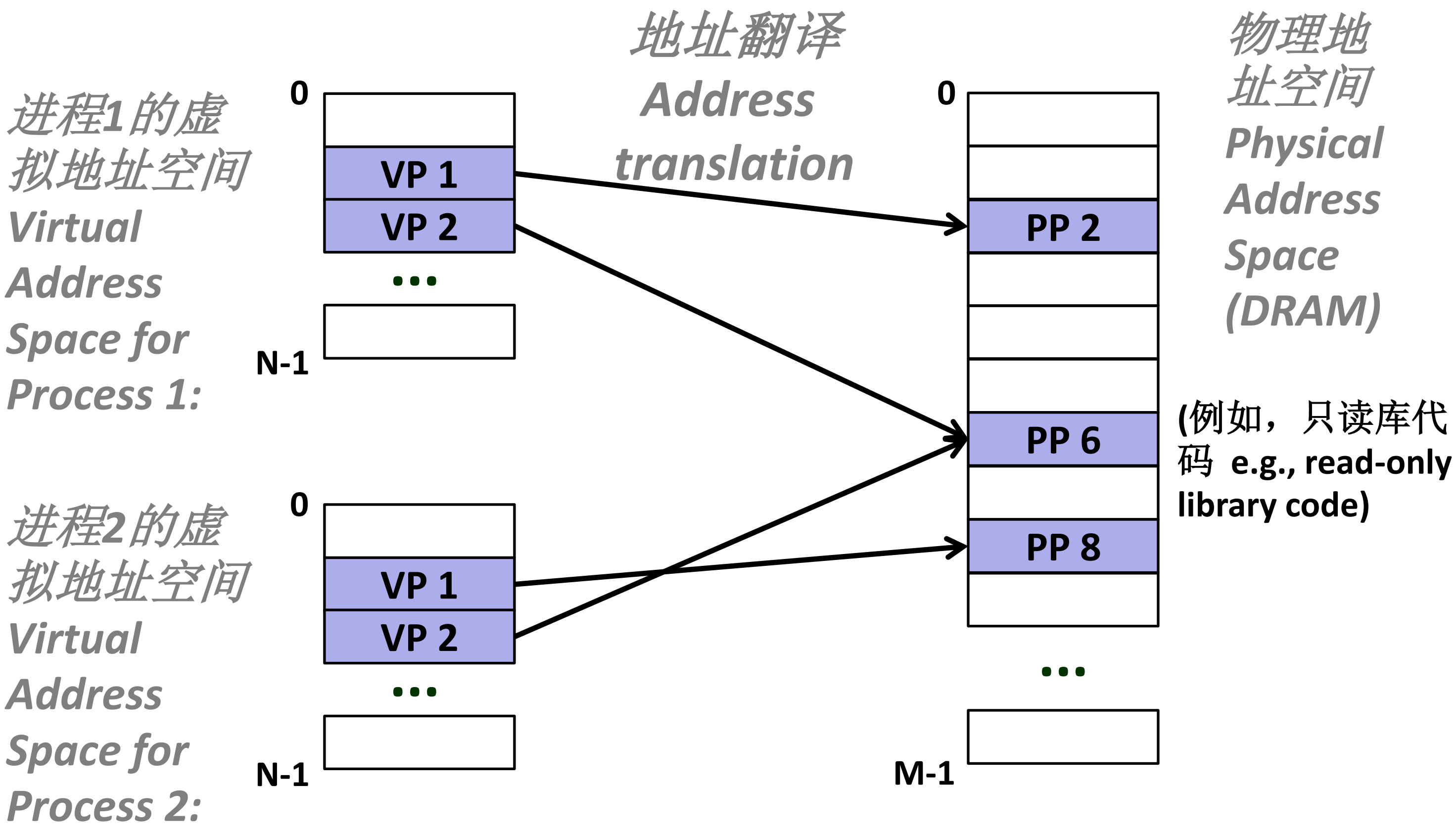
- 简单内存系统示例 Simple memory system example
- 案例研究：Core i7/Linux内存系统 Case study: Core i7/Linux memory system
- 内存映射 Memory mapping

内存映射 Memory Mapping

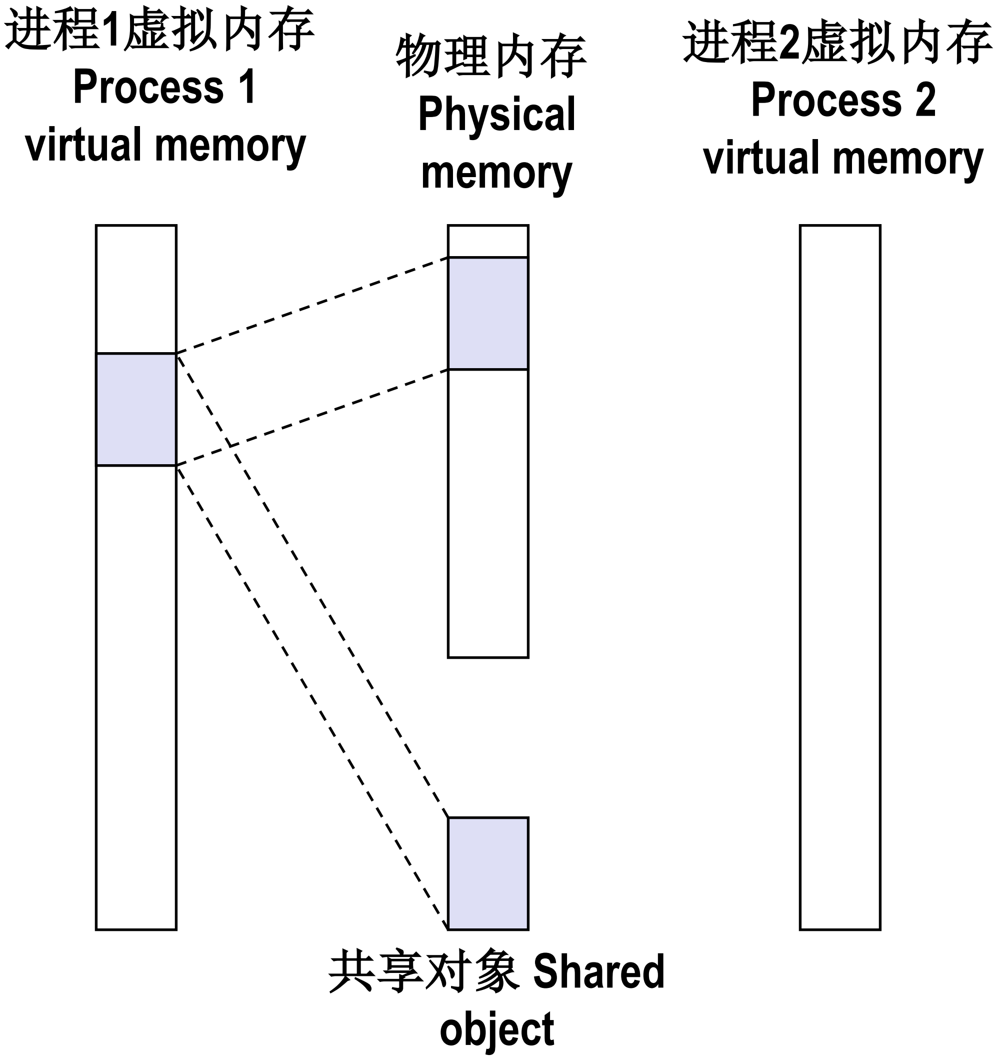
- **VM区域由与其相关联磁盘对象初始化 VM areas initialized by associating them with disk objects.**
  - 这一过程称为*内存映射* Process is known as ***memory mapping***.
- 区域可以由以下提供：　**Area can be *backed by* (**即从以下获得初始值 **i.e., get its initial values from) :**
  - 磁盘上的***常规文件*** ***Regular file*** on disk (例如一个可执行目标文件 e.g., an executable object file)
    - 通过文件的节初始化页中数据 Initial page bytes come from a section of a file
  - ***匿名文件*** ***Anonymous file*** (e.g., nothing)
    - 第一次缺页时分配一个填充为0的物理页（请求二进制零的页） First fault will allocate a physical page full of 0's (***demand-zero page***)
    - 页面被写之后（脏页）就和其他页一样 Once the page is written to (***dirtied***), it is like any other page
- 脏页会在内存和一个特殊的交换文件之间来回拷贝 **Dirty pages are copied back and forth between memory and a special *swap file*.**

回顾：内存管理和保护
Review: Memory Management & Protection

- 代码和数据能够在进程之间隔离或共享 **Code and data can be isolated or shared among processes**

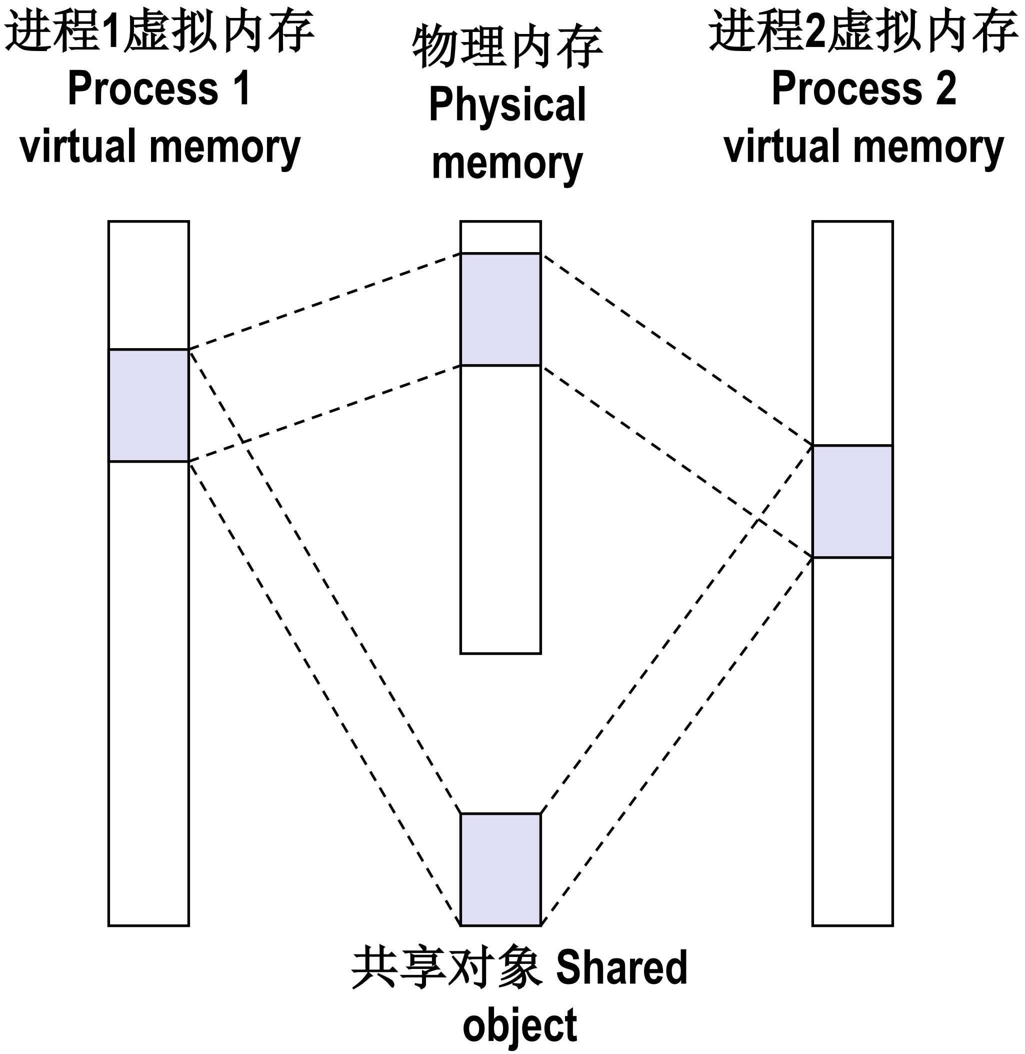


共享重回顾：共享对象
Sharing Revisited: Shared Objects



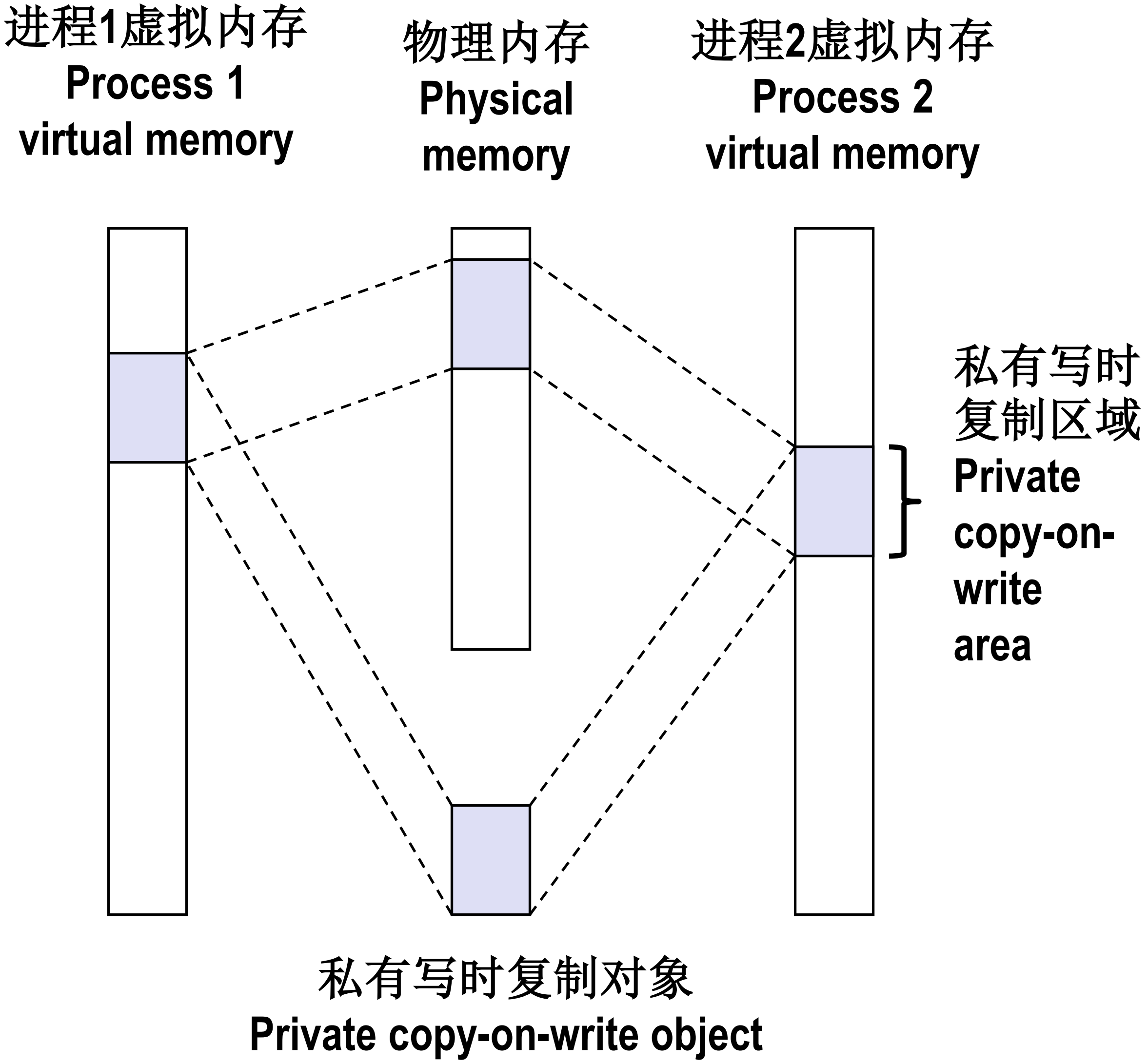
- 进程1映射共享对象 Process 1 maps the shared object.

共享重回顾：共享对象
Sharing Revisited: Shared Objects



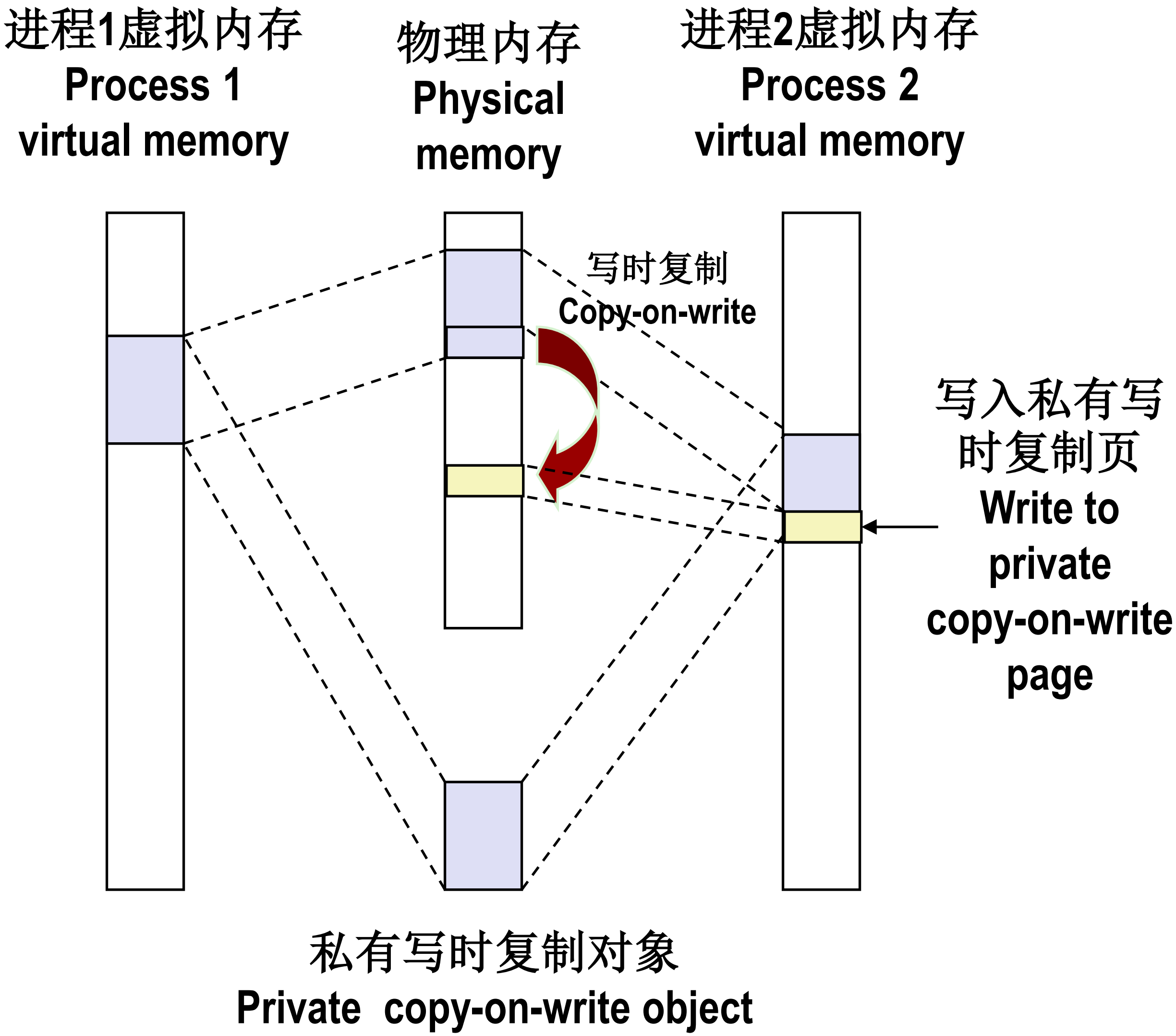
- 进程2映射共享对象 Process 2 maps the shared object.
- 注意虚拟地址如何不同 Notice how the virtual addresses can be different.
- 但是，不同必须是页大小的整倍数 But, difference must be multiple of page size.

共享重回顾:私有写时复制对象
Sharing Revisited: Private Copy-on-write (COW) Objects



- 两个进程映射了一个*私有写时复制（COW）*对象 **Two processes mapping a *private copy-on-write (COW)* object.**
- 区域被标记为私有写时复制 **Area flagged as private copy-on-write**
- 私有区域的PTE被标记为只读 **PTEs in private areas are flagged as read-only**

共享重回顾:私有写时复制对象
Sharing Revisited: Private Copy-on-write (COW) Objects



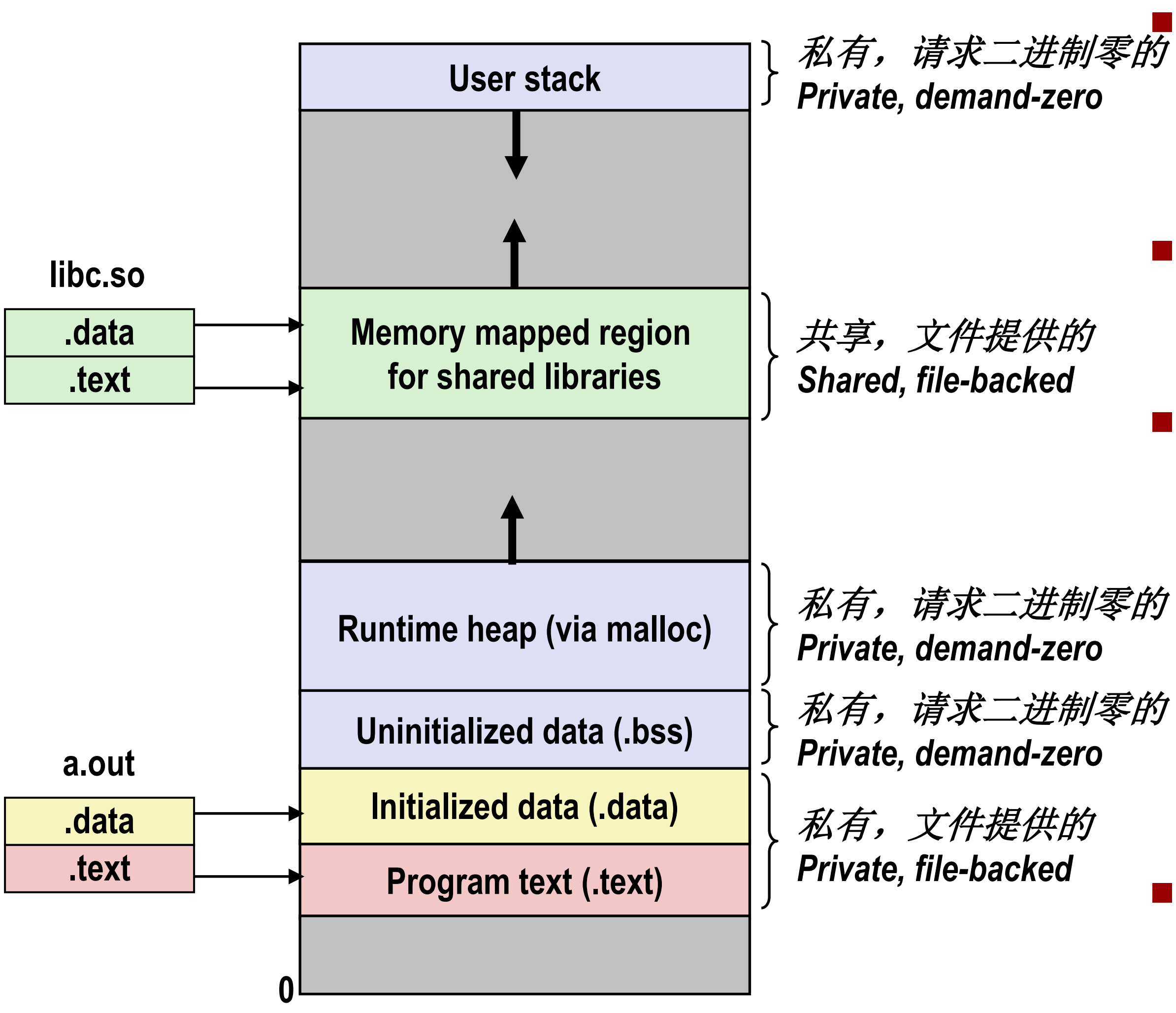
私有写时复制对象
Private copy-on-write object

- 写私有页指令会触发保护异常 Instruction writing to private page triggers protection fault.
- 处理程序创建一个新的R/W页 Handler creates new R/W page.
- 处理程序返回后重新执行指令 Instruction restarts upon handler return.
- 尽可能延迟复制操作 Copying deferred as long as possible!

fork函数重回顾
The `fork` Function Revisited

- **VM和内存映射解释了fork如何为每个进程设置私有地址空间 VM and memory mapping explain how `fork` provides private address space for each process.**
- 为新进程创建虚拟地址 **To create virtual address for new process**
  - 创建完全与现有的完全相同的内存数据结构和页表 Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - 每个进程都将其标记为只读 Flag each page in both processes as read-only
  - 在两个进程空间中的`vm_area_struct` 设置为私有`COW` Flag each `vm_area_struct` in both processes as private COW
- 返回时，每个进程有完全相同的虚拟内存 **On return, each process has exact copy of virtual memory**
- 后续写操作会因为**COW**创建新的页 **Subsequent writes create new pages using COW mechanism.**

execve重回顾
The `execve` Function Revisited



- 在当前进程用execve加载并运行一个新的程序a.out **To load and run a new program `a.out` in the current process using `execve`:**
- 释放旧区域的相关数据结构和页表 **Free `vm_area_struct`'s and page tables for old areas**
- 创建新区域的相关数据结构和页表 **Create `vm_area_struct`'s and page tables for new areas**
  - 程序和初始化过的数据由目标文件提供 Programs and initialized data backed by object files.
  - `.bss`和栈由匿名文件提供 `.bss` and stack backed by anonymous files .
- 设置PC为.text中入口点 **Set PC to entry point in `.text`**
  - Linux将根据需要换入代码和数据页 Linux will fault in code and data pages as needed.

发现可共享页面 Finding Shareable Pages

- 内核相同页面合并 **Kernel Same-Page Merging**
  - OS扫描所有物理内存，查找重复页面 OS scans through all of physical memory, looking for duplicate pages
  - 当找到时合并成单一页面，标记为写时复制 When found, merge into single copy, marked as copy-on-write
  - 2009年在Linux内核实现 Implemented in Linux kernel in 2009
  - 仅限于标记为可能候选的页面 Limited to pages marked as likely candidates
  - 当处理器运行很多个虚拟机时特别有用 Especially useful when processor running many virtual machines

用户级内存映射 User-Level Memory Mapping

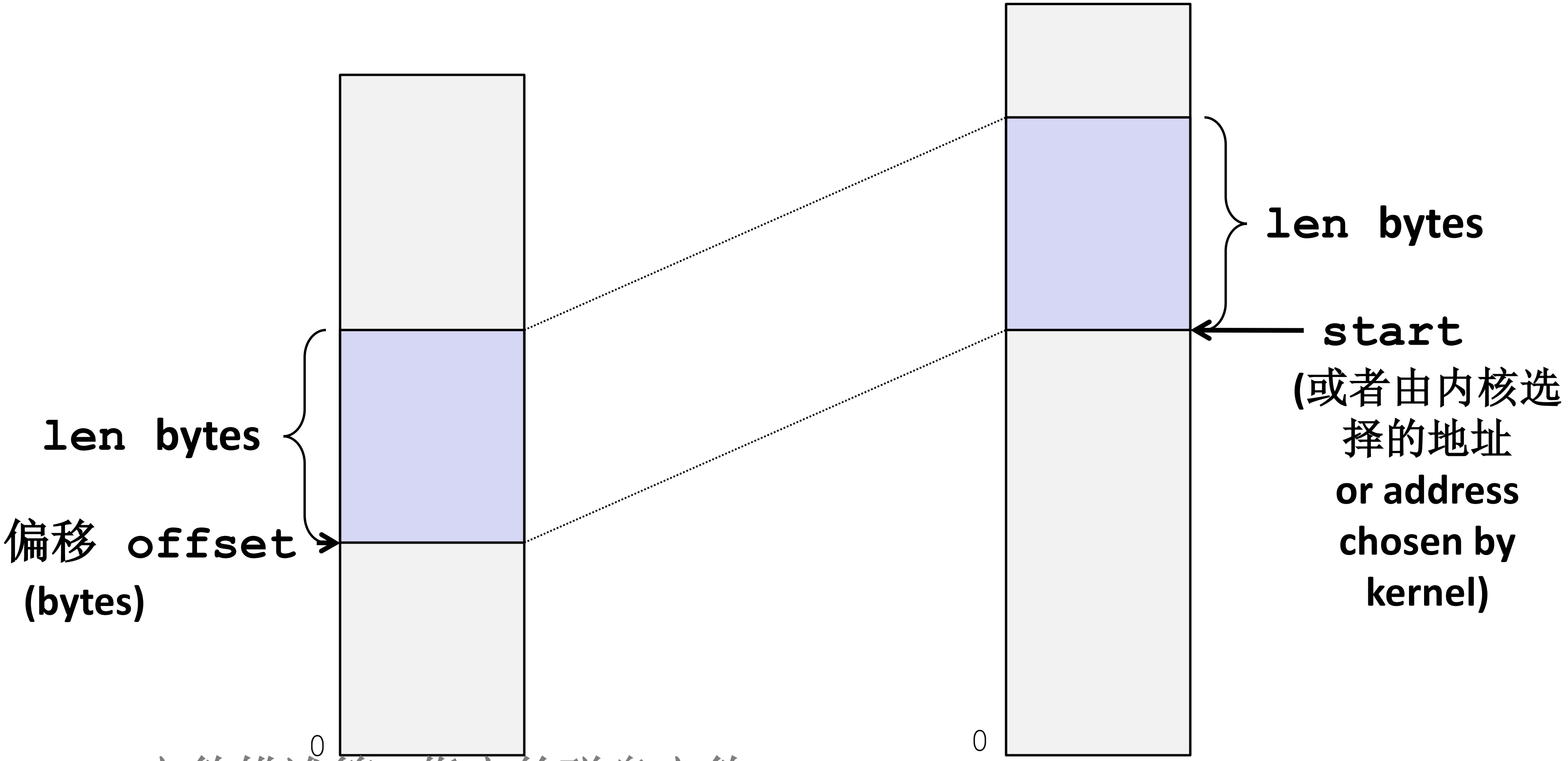
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- **将文件描述符fd中偏移量`offset`开始的长度为`len`的字节映射到地址`start` Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`**
  - `start`: may be 0 for "pick an address" 有可能是0，以选取一个地址
  - `prot`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- 返回一个映射区域的开始地址指针（有可能不是start）**Return a pointer to start of mapped area (may not be `start`)**

用户级内存映射 User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



`len` **bytes**

偏移 `offset` **(bytes)**

0

`len` **bytes**

`start` **(或者由内核选择的地址 or address chosen by kernel)**

0

文件描述符fd指定的磁盘文件 Disk file specified by file descriptor `fd`

进程虚拟内存 Process virtual memory

mmap的使用 Uses of mmap

- 读大文件 **Reading big files**
  - 使用页调度机制将文件调入内存 Uses paging mechanism to bring files into memory
- 共享数据结构 **Shared data structures**
  - 当用MAP_SHARE标志调用时/When call with **`MAP_SHARED`** flag
    - 多个进程访问同样的内存区域 Multiple processes have access to same region of memory
    - 有风险！ Risky!
- 基于文件的数据结构 **File-based data structures**
  - 例如数据库 E.g., database
  - 给出prot参数为：Give `prot` argument **`PROT_READ | PROT_WRITE`**
  - 当释放映射区域时，文件通过写回进行更新 When unmap region, file will be updated via write-back
  - 可以实现从文件加载/更新/写回到文件 Can implement load from file / update / write back to file

示例：使用mmap拷贝文件
Example: Using `mmap` to Copy Files

- 不用传输数据到用户空间来，就可以将一个文件拷贝到标准输出
  Copying a file to `stdout` without transferring data to user space .

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

   /* Ptr to memory mapped area */
   char *bufp;

   bufp = Mmap(NULL, size,
            PROT_READ,
            MAP_PRIVATE,
            fd, 0);
   Write(1, bufp, size);
   return;
}
```

mmapcopy.c

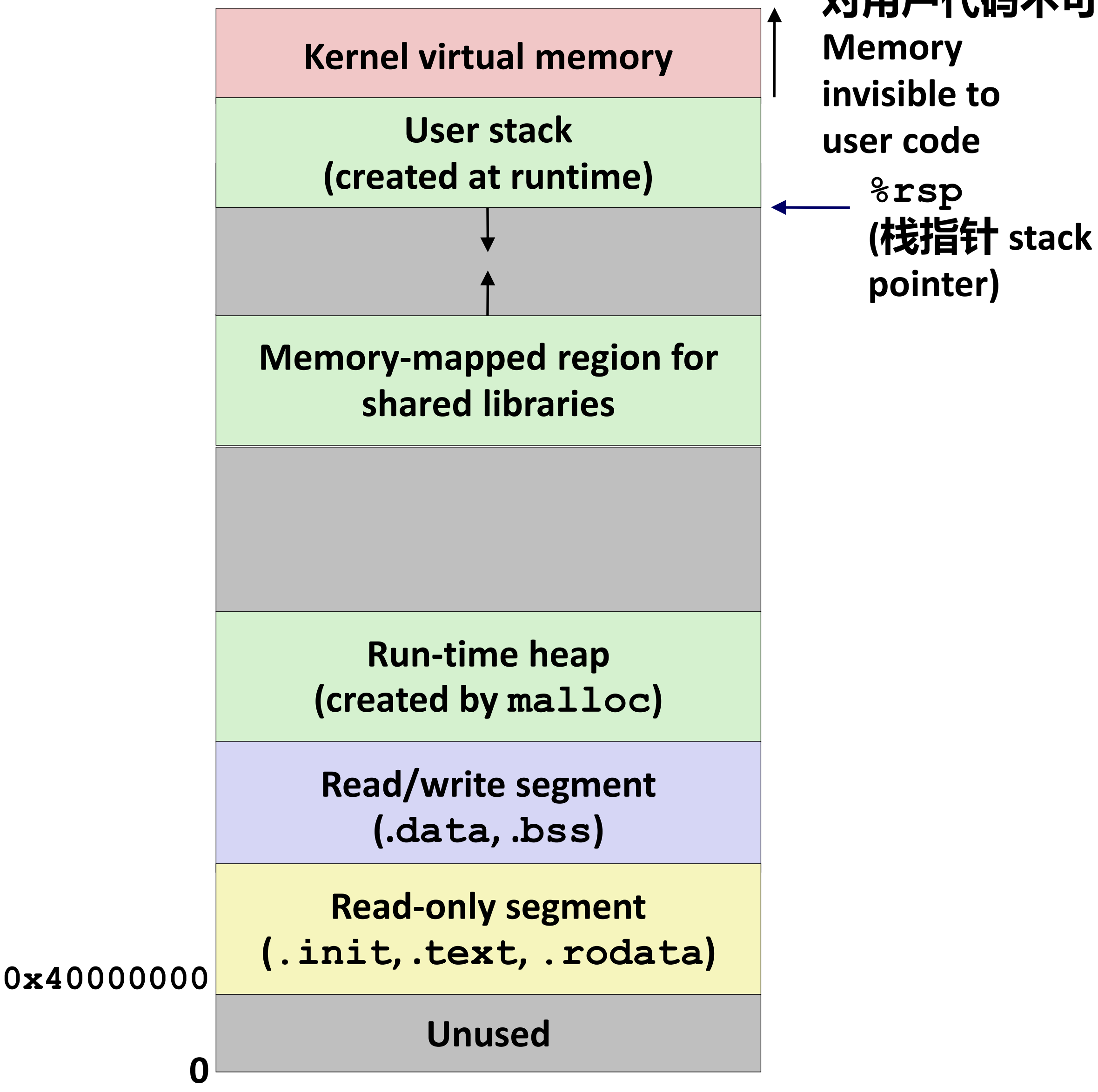
```
/* mmapcopy driver */
int main(int argc, char **argv)
{
   struct stat stat;
   int fd;

   /* Check for required cmd line arg */
   if (argc != 2) {
      printf("usage: %s <filename>\n",
           argv[0]);
      exit(0);
   }

   /* Copy input file to stdout */
   fd = Open(argv[1], O_RDONLY, 0);
   Fstat(fd, &stat);
   mmapcopy(fd, stat.st_size);
   exit(0);
}
```

mmapcopy.c

示例：使用mmap支持攻击实验
Example: Using `mmap` to Support Attack Lab

- 问题 Problem
  - 期望学生能够执行代码注入攻击 Want students to be able to perform code injection attacks
  - 我们实验机器栈不能执行代码 Shark machine stacks are not executable
- 解决方案 Solution
  - 由Sam King建议（现在在Davis）Suggested by Sam King (now at UC Davis)
  - 使用mmap分配标记为可执行的内存区域 Use `mmap` to allocate region of memory marked executable
  - 转向攻击到新的区域 Divert stack to new region
  - 执行学生的攻击代码 Execute student attack code
  - 恢复回原始栈 Restore back to original stack
  - 删除映射的区域 Remove mapped region

使用mmap支持攻击实验
Using mmap to Support Attack Lab



Kernel virtual memory

User stack
(created at runtime)

Memory-mapped region for shared libraries

Run-time heap
(created by `malloc`)

Read/write segment
(`.data`, `.bss`)

Read-only segment
(`.init`, `.text`, `.rodata`)

Unused

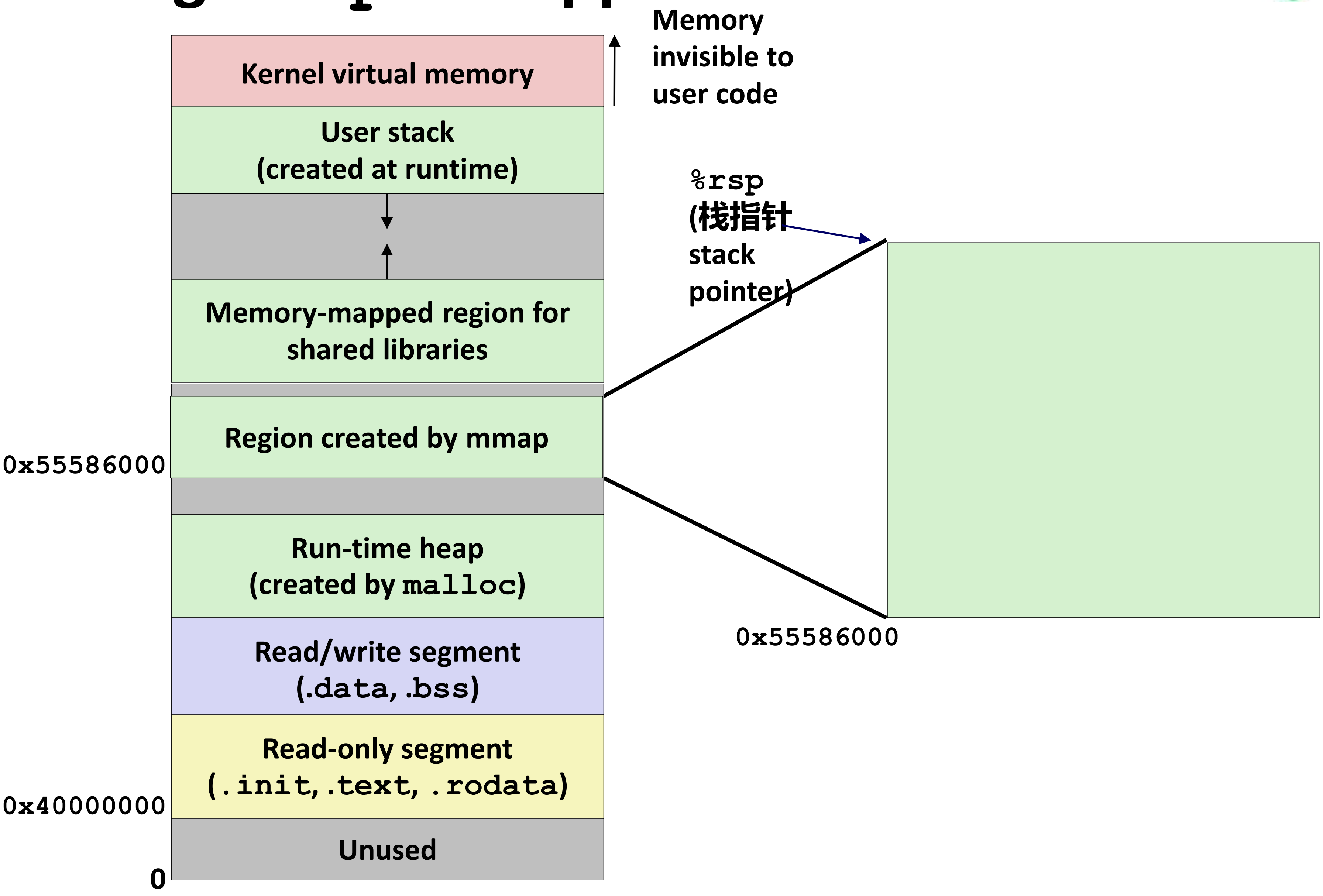
对用户代码不可见内存
Memory
invisible to
user code

`%rsp`
(栈指针 stack pointer)

`0x40000000`

0

使用mmap支持攻击实验
Using `mmap` to Support Attack Lab



Memory invisible to user code

Kernel virtual memory

User stack (created at runtime)

Memory-mapped region for shared libraries

Region created by mmap

Run-time heap (created by `malloc`)

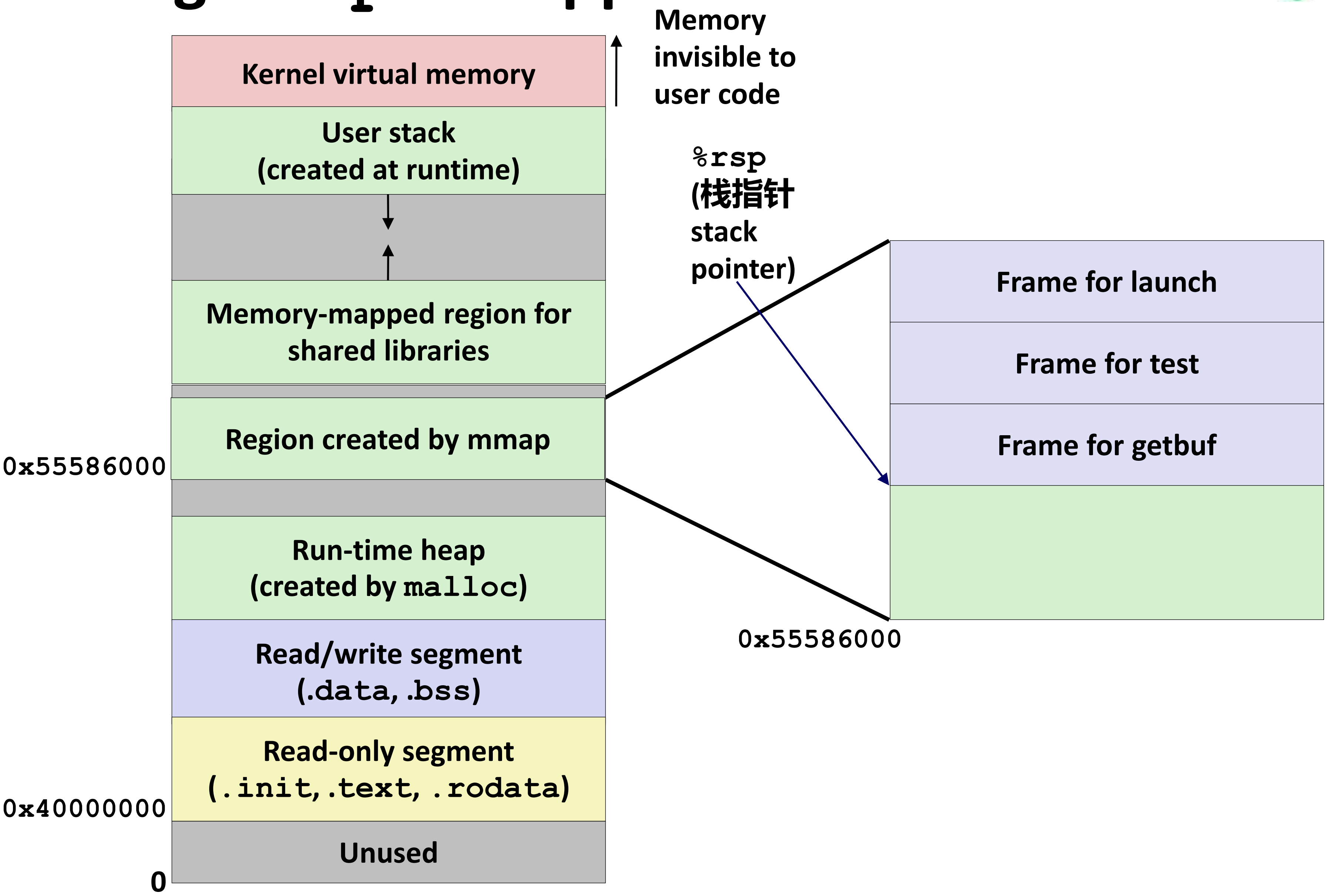
Read/write segment (`.data`, `.bss`)

Read-only segment (`.init`, `.text`, `.rodata`)

Unused

`%rsp` (栈指针 stack pointer)

使用mmap支持攻击实验
Using mmap to Support Attack Lab



Memory invisible to user code

Kernel virtual memory

User stack
(created at runtime)

`%rsp`
(栈指针
stack
pointer)

Memory-mapped region for
shared libraries

Region created by mmap

`0x55586000`

Frame for launch

Frame for test

Frame for getbuf

`0x55586000`

Run-time heap
(created by `malloc`)

Read/write segment
(`.data`, `.bss`)

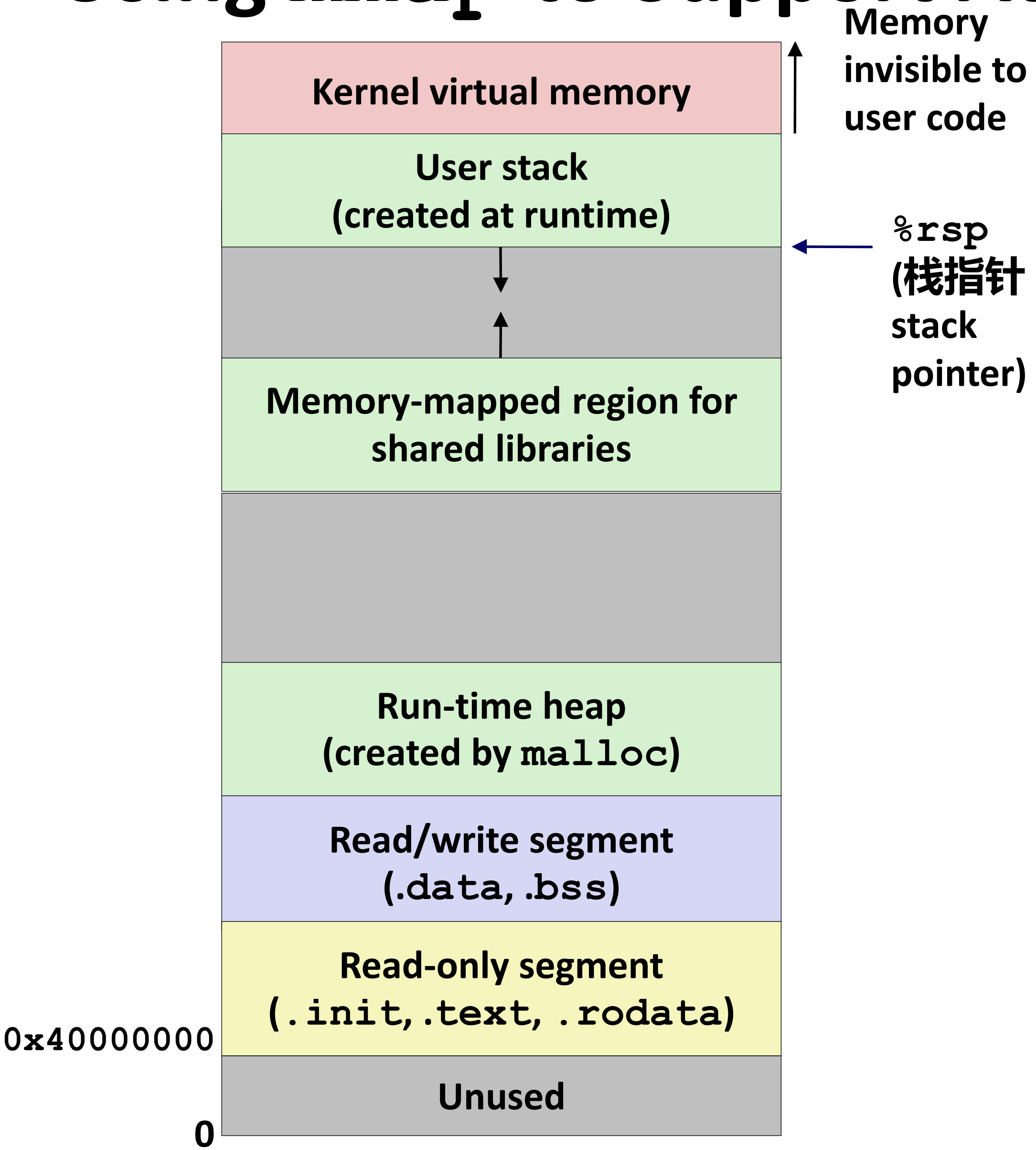
Read-only segment
(`.init`, `.text`, `.rodata`)

`0x40000000`

Unused

0

使用mmap支持攻击实验
Using `mmap` to Support Attack Lab



Memory invisible to user code

Kernel virtual memory

User stack (created at runtime)

`%rsp` (栈指针 stack pointer)

Memory-mapped region for shared libraries

Run-time heap (created by `malloc`)

Read/write segment (`.data`, `.bss`)

Read-only segment (`.init`, `.text`, `.rodata`)

`0x40000000`

Unused

0

总结 Summary

- 虚拟内存需要硬件支持 **VM requires hardware support**
  - 异常处理机制 Exception handling mechanism
  - TLB
  - 各种控制寄存器 Various control registers
- 虚拟内存需要操作系统支持 **VM requires OS support**
  - 管理页表 Managing page tables
  - 实现页替换策略 Implementing page replacement policies
  - 管理文件系统 Managing file system
- 虚拟内存使能许多能力 **VM enables many capabilities**
  - 从内存加载程序 Loading programs from memory
  - 提供内存保护 Providing memory protection

使用mmap支持攻击实验
Using mmap to Support Attack Lab

分配新的区域 Allocate new region

```
void *new_stack = mmap(START_ADDR, STACK_SIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
                MAP_PRIVATE | MAP_GROWSDOWN | MAP_ANONYMOUS | MAP_FIXED,
                0, 0);
if (new_stack != START_ADDR) {
    munmap(new_stack, STACK_SIZE);
    exit(1);
}
```

转向栈到新区域并执行攻击代码
Divert stack to new region & execute attack code

```
stack_top = new_stack + STACK_SIZE - 8;
asm("movq %%rsp,%%rax ; movq %1,%%rsp ;
movq %%rax,%0"
    : "=r" (global_save_stack) // %0
    : "r"  (stack_top)         // %1
);

launch(global_offset);
```

恢复栈并删除区域
Restore stack and remove region

```
asm("movq %0,%%rsp"
    :
    : "r" (global_save_stack) // %0
);

munmap(new_stack, STACK_SIZE);
```

第9章 虚拟内存

Dynamic Memory Allocation: Basic Concepts

动态存储分配:基本概念

100076202： 计算机系统导论

任课教师:

宿红毅　　张艳　　黎有琦　　颜珂

原作者:

Randal E. Bryant and David R. O’Hallaron

议题 Today

- **基本概念 Basic concepts**
- 隐式空闲列表 Implicit free lists

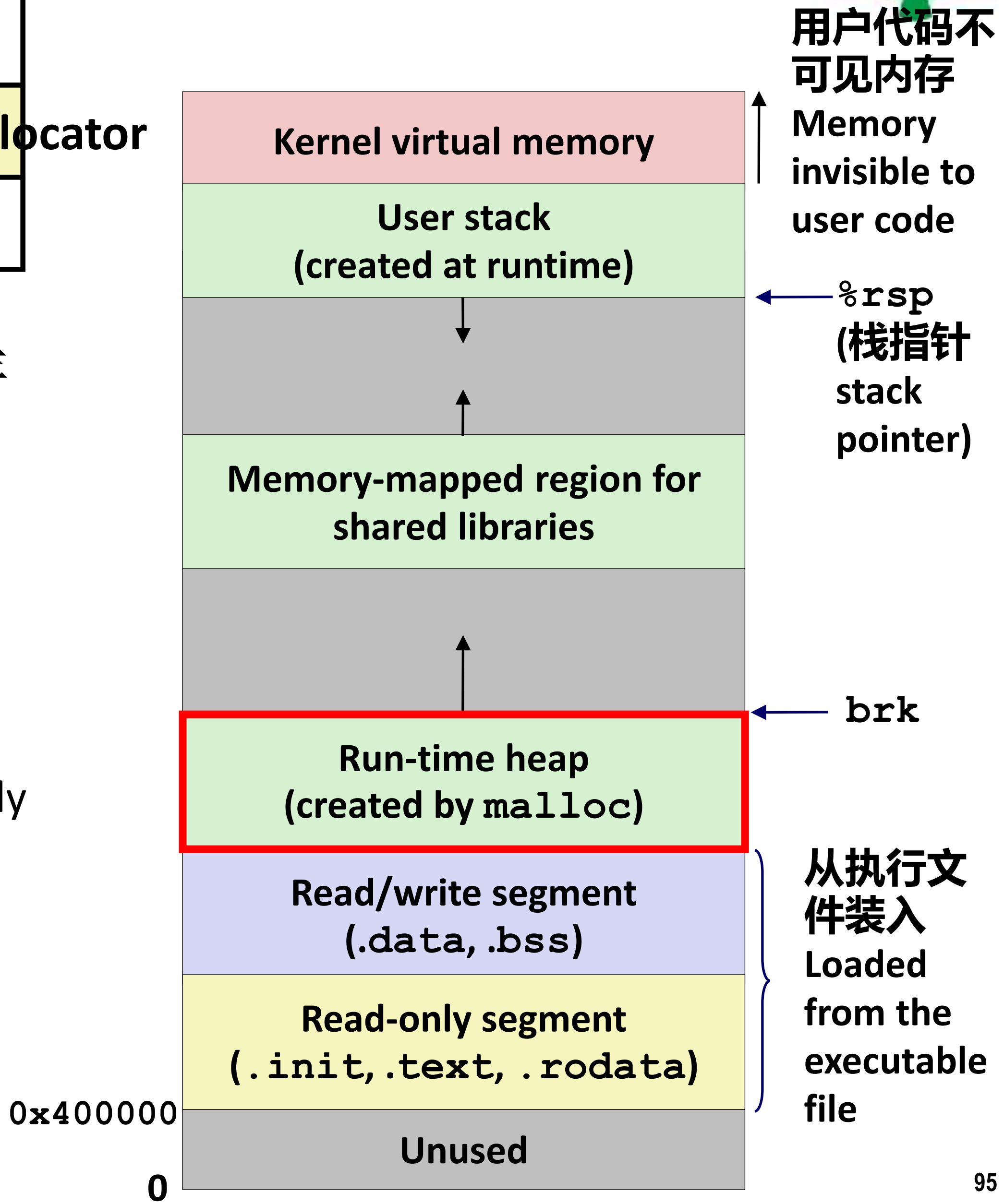
动态内存分配 Dynamic Memory Allocation

应用 **Application**

动态内存分配器 **Dynamic Memory Allocator**

堆 **Heap**

- 程序员使用动态内存分配器(malloc)在运行时申请虚拟内存 **Programmers use *dynamic memory allocators* (such as `malloc`) to acquire virtual memory (VM) at runtime**
  - 对于那些数据结构大小在运行时才能知道的数据结构 For data structures whose size is only known at runtime
- 动态内存分配器管理进程虚拟内存中一个称为堆的区域 **Dynamic memory allocators manage an area of process VM known as the *heap***



用户代码不可见内存 Memory invisible to user code

Kernel virtual memory

User stack (created at runtime)

`%rsp` **(栈指针 stack pointer)**

Memory-mapped region for shared libraries

`brk`

Run-time heap (created by `malloc`)

Read/write segment (`.data`, `.bss`)

Read-only segment (`.init`, `.text`, `.rodata`)

从执行文件装入 Loaded from the executable file

`0x400000`

Unused

0

动态内存分配 Dynamic Memory Allocation

- 分配器将堆当做不同大小的块的集合进行管理，不是已分配就是空闲 **Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free***
- 分配器类型 **Types of allocators**
  - ***显式分配器：***应用程序分配和释放空间 ***Explicit allocator*:** application allocates and frees space
    - 例如C中的malloc和free E.g., `malloc` and `free` in C
  - ***隐式分配器：***应用只负责分配但是不释放空间 ***Implicit allocator:*** application allocates, but does not free space
    - 例如Java、ML和Lisp中的垃圾收集 E.g. garbage collection in Java, ML, and Lisp
- 今天主要讨论简单的显式内存分配 **Will discuss simple explicit memory allocation today**

malloc包 The `malloc` Package

`#include <stdlib.h>`

`void *malloc(size_t size)`

- 成功 Successful:
  - 返回大小至少是size的内存块指针，x86上是按8字节对齐，x86-64是按16字节对齐 Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - 如果size为0，则返回NULL If **`size == 0`**, returns NULL
- 不成功：返回NULL并设置errno Unsuccessful: returns NULL (0) and sets **`errno`**

`void free(void *p)`

- 将p指向的内存块返回给可用内存池 Returns the block pointed at by **`p`** to pool of available memory
- **`p`**必须是之前调用`malloc`或者`realloc`获得的 **`p`** must come from a previous call to **`malloc`** or **`realloc`**

其他函数 **Other functions**

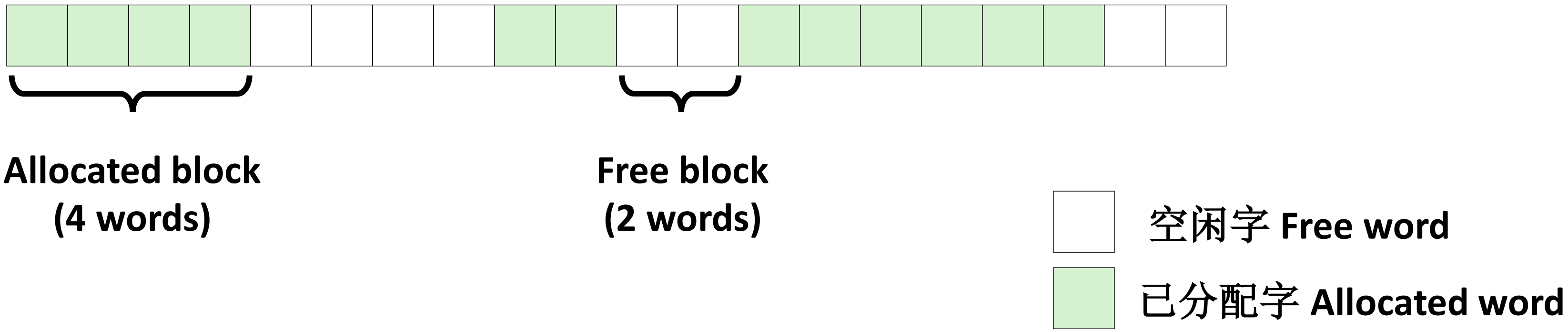
- **`calloc`**: malloc的另一个版本，会将分配的内存块初始化为0 Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: 改变之前分配的块的大小 Changes the size of a previously allocated block.
- **`sbrk`**: 分配器内部用来增加或者减小堆的大小 Used internally by allocators to grow or shrink the heap

`malloc`示例 `malloc` Example

```
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
   int i, *p;

   /* Allocate a block of n ints */
   p = (int *) malloc(n * sizeof(int));
   if (p == NULL) {
      perror("malloc");
      exit(0);
   }

   /* Initialize allocated block */
   for (i=0; i<n; i++)
            p[i] = i;


   /* Return allocated block to the heap */
   free(p);
}
```

可视化展示规则 Visualization Conventions

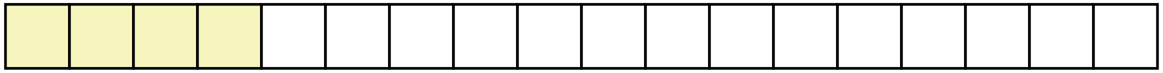
- 显式8字节字为一个方块 **Show 8-byte words as squares**
- 分配采用双字对齐 **Allocations are double-word aligned**



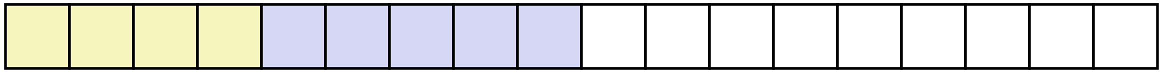
分配示例 Allocation Example (概念上 Conceptual)

`#define SIZ sizeof(size_t)`

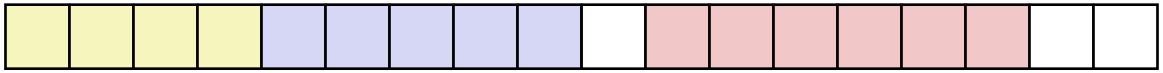
`p1 = malloc(4*SIZ)`



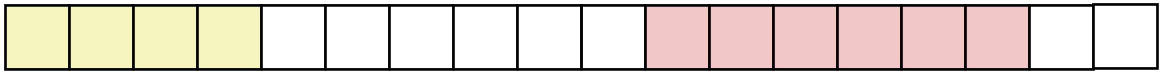
`p2 = malloc(5*SIZ)`



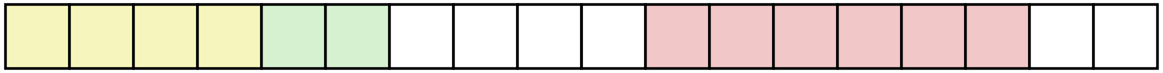
`p3 = malloc(6*SIZ)`



`free(p2)`



`p4 = malloc(2*SIZ)`



限制 Constraints

- **应用 Applications**
  - 可以发出任意malloc和free请求序列 Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`**请求必须针对一个`malloc`请求的块 **`free`** request must be to a **`malloc`**’d block

- **显式分配器 Explicit Allocators**
  - 无法控制分配的块的数量和大小 Can’t control number or size of allocated blocks
  - 必须及时响应malloc请求 Must respond immediately to **`malloc`** requests
    - 例如，不能对请求排序和缓冲 *i.e.*, can’t reorder or buffer requests
  - 必须从空闲空间分配内存块 Must allocate blocks from free memory
    - 例如，分配的块必须在空闲内存中 *i.e.*, can only place allocated blocks in free memory
  - 必须按照需求实现块对齐 Must align blocks so they satisfy all alignment requirements
    - Linux中x86是8字节对齐，x86-64是16字节对齐 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
  - 只能操作和修改空闲内存 Can manipulate and modify only free memory
  - 一旦分配后不能移动内存块 Can’t move the allocated blocks once they are **`malloc`**’d
    - 例如，压缩是不允许的 *i.e.*, compaction is not allowed

性能目标：吞吐率 Performance Goal: Throughput

- **对于给定的malloc和free序列 Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- **目标：最大化吞吐率和峰值内存利用率 Goals: maximize throughput and peak memory utilization**
  - 这些目标通常是互相冲突的 These goals are often conflicting

- **吞吐率 Throughput:**
  - 单位时间内完成的请求数量 Number of completed requests per unit time
  - 例如： Example:
    - 10秒内完成5000次malloc和5000次free 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - 吞吐率就是1000次操作/秒 Throughput is 1,000 operations/second

性能目标：最小化开销
Performance Goal: Minimize Overhead

- 对于给定的**malloc**和**free**某个请求序列 **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- ***K次请求之后，我们得到： After k requests we have:***

- *定义：总有效载荷* ***Def: Aggregate payload*** P_k
  - `malloc(p)` 返回一个载荷为p字节的块 `malloc(p)` results in a block with a ***payload*** of `p` bytes
  - 请求R_k完成后，总有效载荷P_k是目前已分配的载荷的总大小 After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads

- *定义：*当前堆大小**H_k** ***Def: Current heap size*** H_k
  - 假设H_k单调不递减 Assume H_k is monotonically nondecreasing
    - 即当分配器使用sbrk时堆增加 i.e., heap only grows when allocator uses `sbrk`

- *定义：* ***k+1***次请求之后峰值内存利用率 ***Def: Peak memory utilization after k+1 requests***
  - $U_k = (\max_{i<=k} P_i) \ / \ H_k$

性能目标：最小化开销
Performance Goal: Minimize Overhead

- **对于给定的malloc和free一些请求序列 Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***K次请求之后，我们得到： After k requests we have:***
- ***定义：总有效载荷 Def: Aggregate payload P_k***
  - **`malloc(p)`** 返回一个载荷为p字节的块/**`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - 总有效载荷P_k是目前已分配的载荷的总和 The ***aggregate payload*** P_k is the sum of currently allocated payloads
  - 峰值总有效载荷是请求序列中任何点处最大总有效载荷 The ***peak aggregate payload*** $\max_{i \le k} P_i$ is the maximum aggregate payload at any point in the sequence up to request

性能目标：最小化开销
Performance Goal: Minimize Overhead

- **对于给定的malloc和free一些请求序列 Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***K次请求之后，我们得到： After k requests we have:***
- *定义：*当前堆大小**H_k** ***Def:* Current heap size H_k**
  - 假设当分配器使用sbrk时堆仅增加，从不收缩 Assume heap only *grows* when allocator uses **`sbrk`**, never shrinks
- ***定义：开销，O_k Def: Overhead, O_k***
  - 堆空间没有为程序数据使用的比例 Fraction of heap space *NOT* used for program data
  - $O_k = (H_k / \max_{i \le k} P_i) - 1.0$

基准测试示例 Benchmark Example

- 基准测试 Benchmark

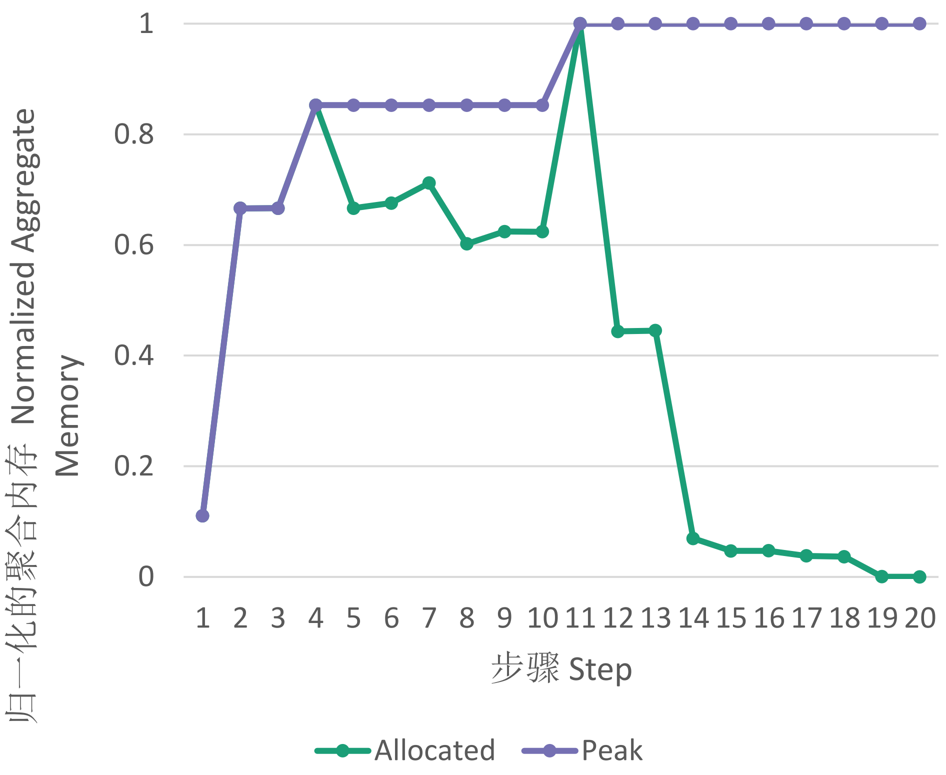
  syn-array-short

  - malloc实验提供的跟踪 Trace provided with malloc lab
  - 分配和释放各10个块 Allocate & free 10 blocks
  - a代表分配 a = allocate
  - f代表释放 f = free
  - 偏置在开始时分配，在结束时释放 Bias toward allocate at beginning & free at end
  - 块号1-10 Blocks number 1–10
  - 已分配：所有分配量的和 Allocated: Sum of all allocated amounts
  - 峰值：曾经分配的最大值 Peak: Max so far of Allocated

步骤 Step	命令 Command	偏置 Delta	已分配 Allocated	峰值 Peak
1	`a 0 9904`	9904	9904	9904
2	`a 1 50084`	50084	59988	59988
3	`a 2 20`	20	60008	60008
4	`a 3 16784`	16784	76792	76792
5	`f 3`	-16784	60008	76792
6	`a 4 840`	840	60848	76792
7	`a 5 3244`	3244	64092	76792
8	`f 0`	-9904	54188	76792
9	`a 6 2012`	2012	56200	76792
10	`f 2`	-20	56180	76792
11	`a 7 33856`	33856	90036	90036
12	`f 1`	-50084	39952	90036
13	`a 8 136`	136	40088	90036
14	`f 7`	-33856	6232	90036
15	`f 6`	-2012	4220	90036
16	`a 9 20`	20	4240	90036
17	`f 4`	-840	3400	90036
18	`f 8`	-136	3264	90036
19	`f 5`	-3244	20	90036
20	`f 9`	-20	0	90036

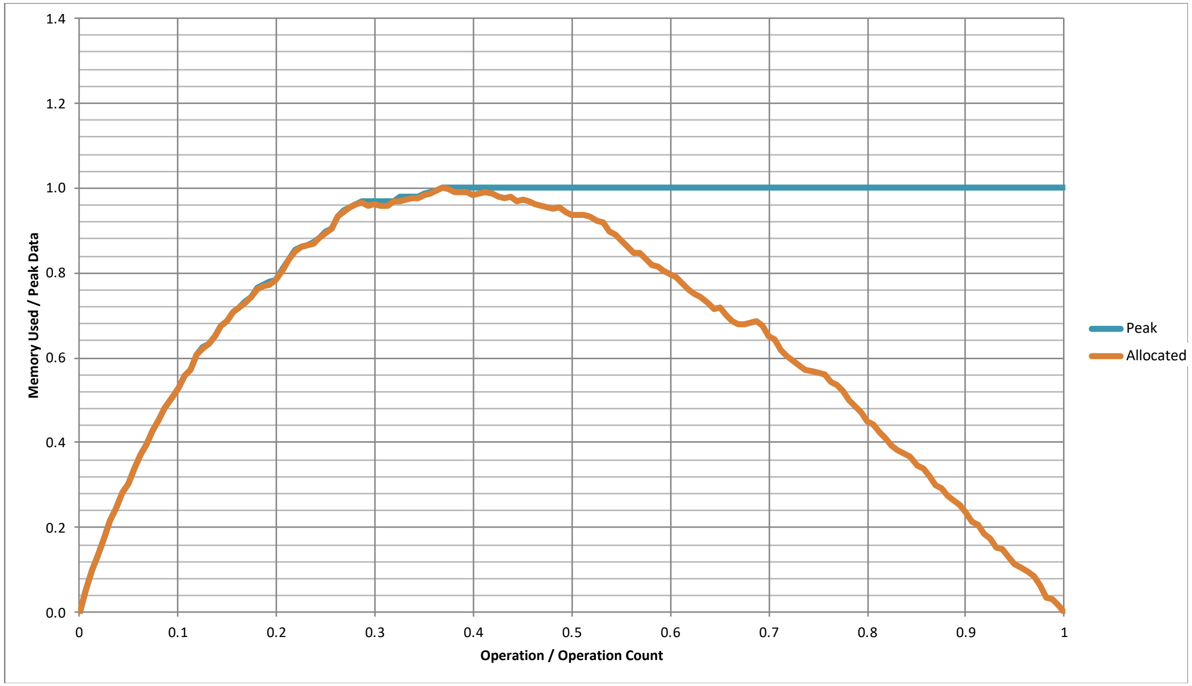
基准测试可视化 Benchmark Visualization

步骤 Step	命令 Command	偏置 Delta	已分配 Allocated	峰值 Peak
1	a 0 9904	9904	9904	9904
2	a 1 50084	50084	59988	59988
3	a 2 20	20	60008	60008
4	a 3 16784	16784	76792	76792
5	f 3	-16784	60008	76792
6	a 4 840	840	60848	76792
7	a 5 3244	3244	64092	76792
8	f 0	-9904	54188	76792
9	a 6 2012	2012	56200	76792
10	f 2	-20	56180	76792
11	a 7 33856	33856	90036	90036
12	f 1	-50084	39952	90036
13	a 8 136	136	40088	90036
14	f 7	-33856	6232	90036
15	f 6	-2012	4220	90036
16	a 9 20	20	4240	90036
17	f 4	-840	3400	90036
18	f 8	-136	3264	90036
19	f 5	-3244	20	90036
20	f 9	-20	0	90036



- 已分配内存和峰值内存是步骤k的函数绘图 Plot P_k (allocated) and $\max_{i \le k} P_k$ (peak) as a function of k (step)
- Y轴归一化处理—占最大值的比例 Y-axis normalized — fraction of maximum

典型的基准测试行为
Typical Benchmark Behavior



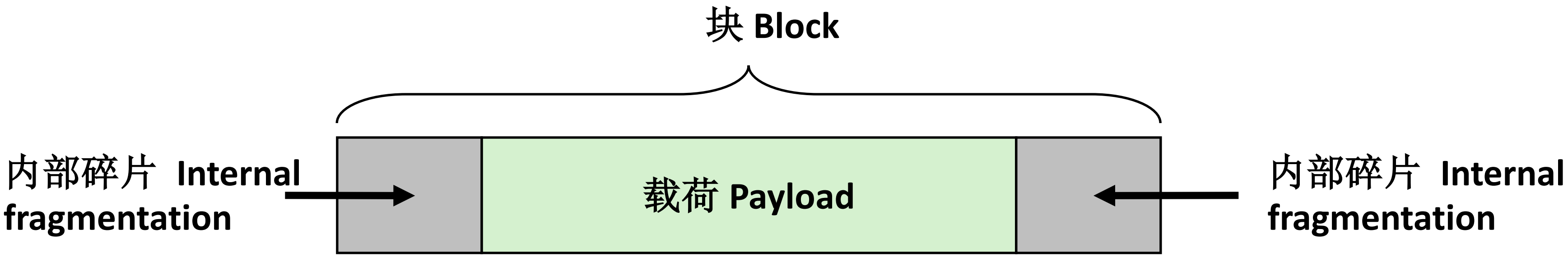
- 分配和释放内存的长序列（40000块） **Longer sequence of mallocs & frees (40,000 blocks)**
  - 开始都是分配内存，然后转向释放内存 Starts with all mallocs, and shifts toward all frees
- 分配器必须整个时间段内有效管理空间 **Allocator must manage space efficiently the whole time**
- 生产分配器可以收缩堆 **Production allocators can shrink the heap**

内存碎片 Fragmentation

- 由内存碎片导致的内存低利用率 **Poor memory utilization caused by *fragmentation***
  - 内部碎片 ***internal*** fragmentation
  - 外部碎片 ***external*** fragmentation

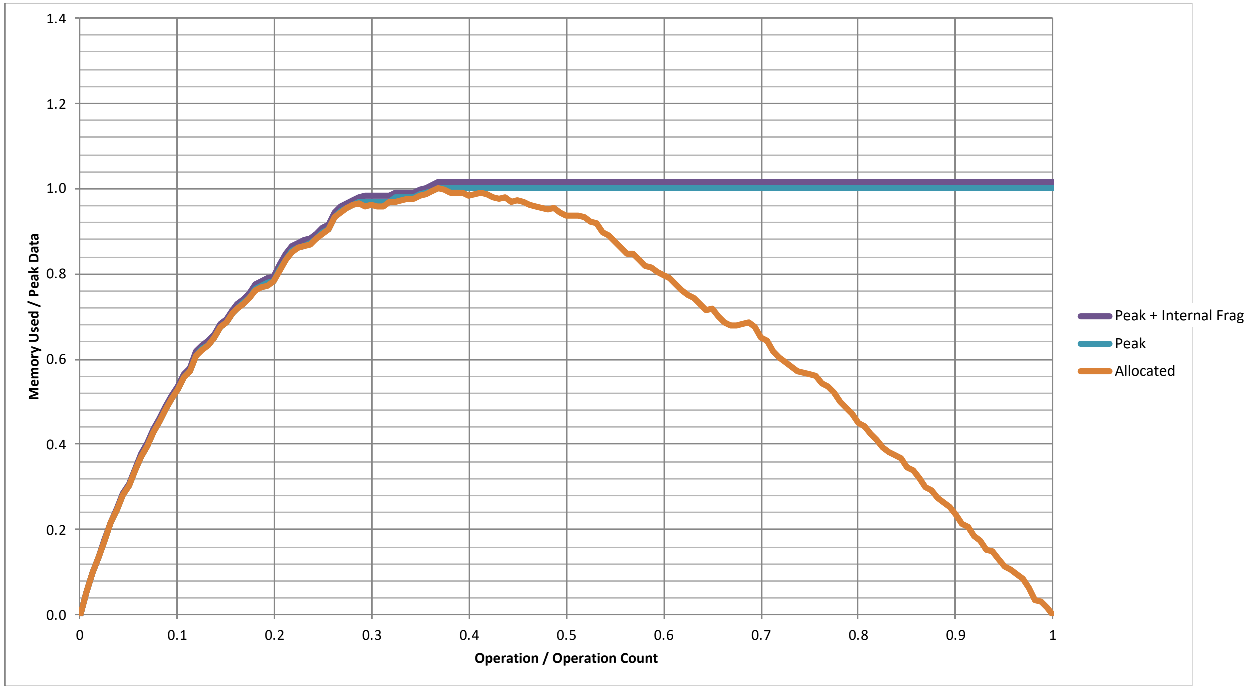
内部碎片 Internal Fragmentation

- **对于给定的块，如果载荷小于块大小就会导致内部碎片 For a given block, *internal fragmentation* occurs if payload is smaller than block size**



- **原因 Caused by**
  - 维护堆数据结构开销 Overhead of maintaining heap data structures
  - 为了对齐填充的部分 Padding for alignment purposes
  - 显式策略导致 Explicit policy decisions
    (例如：为了满足一个小的请求返回一个大的块 e.g., to return a big block to satisfy a small request)
- **只是与之前的请求的模式相关 Depends only on the pattern of *previous* requests**
  - 因此易于度量 Thus, easy to measure

内部碎片效应
Internal Fragmentation Effect

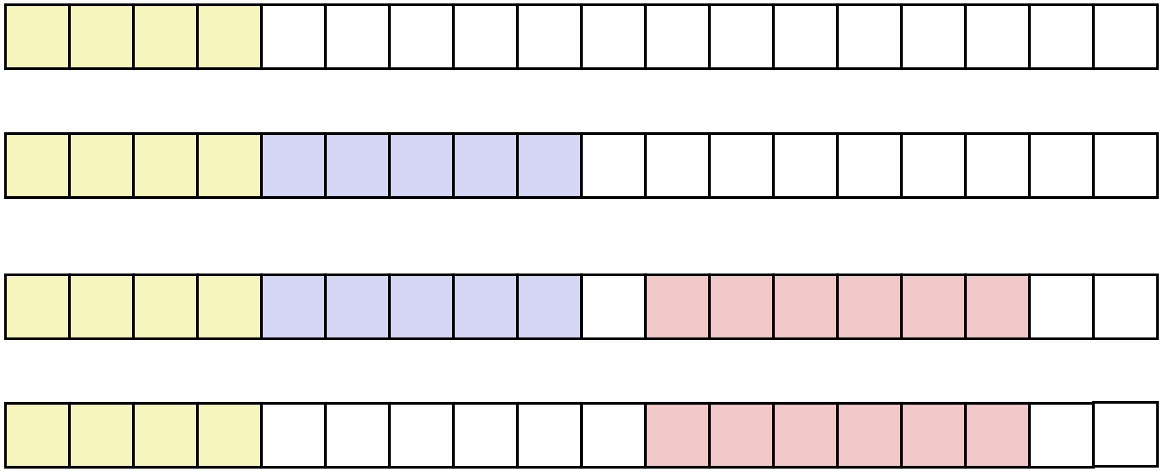


- 紫色线条：由于分配器的数据+对齐填充，堆大小增加 **Purple line: additional heap size due to allocator’s data + padding for alignment**
  - 对于该基准，1.5%的开销 For this benchmark, 1.5% overhead
  - 无法在实践中实现 Cannot achieve in practice
  - 特别是因为无法移动已分配的块 Especially since cannot move allocated blocks

外部碎片 External Fragmentation

`#define SIZ sizeof(size_t)`

- 当有足够的聚合堆内存，但是没有单一的空闲块足够大时产生外部碎片 **Occurs when there is enough aggregate heap memory, but no single free block is large enough**



```
p1 = malloc(4*SIZ)
```

```
p2 = malloc(5*SIZ)
```

```
p3 = malloc(6*SIZ)
```

```
free(p2)
```

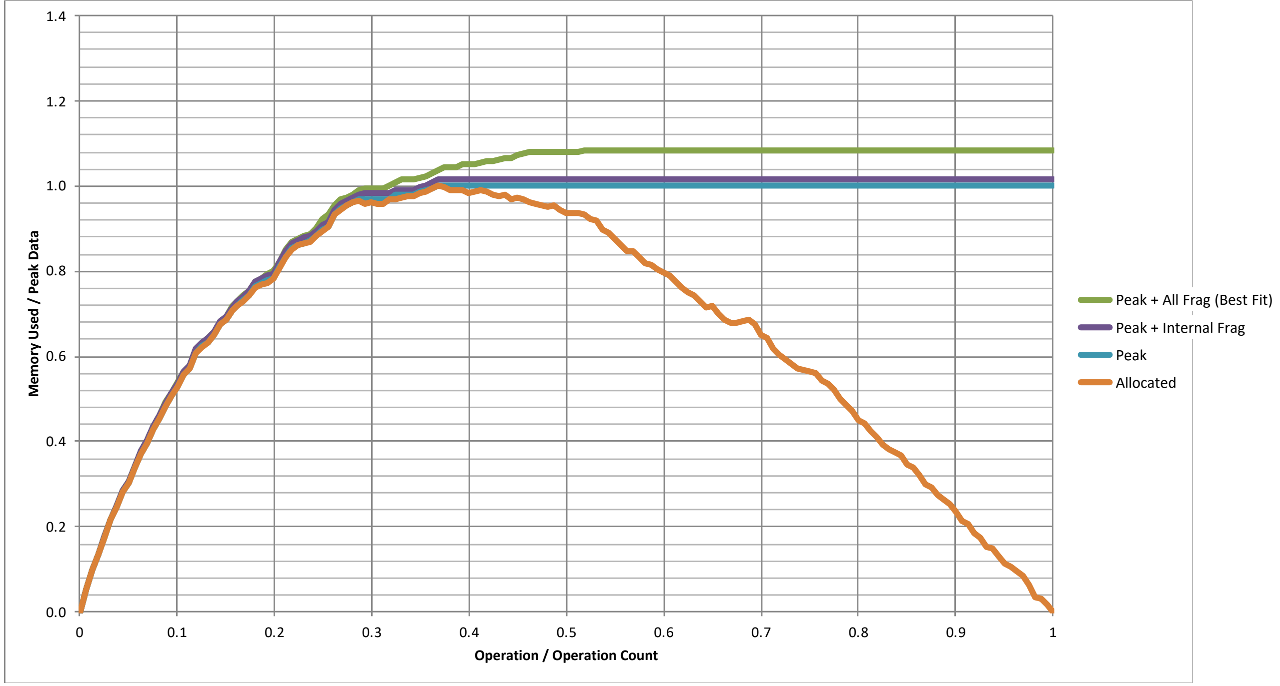
```
p4 = malloc(7*SIZ)
```

诶呀（现在会发生什么？）

Yikes! (what would happen now?)

- 取决于未来请求的模式 **Depends on the pattern of future requests**
  - 因此，难以测量 Thus, difficult to measure

外部碎片的效应
External Fragmentation Effect



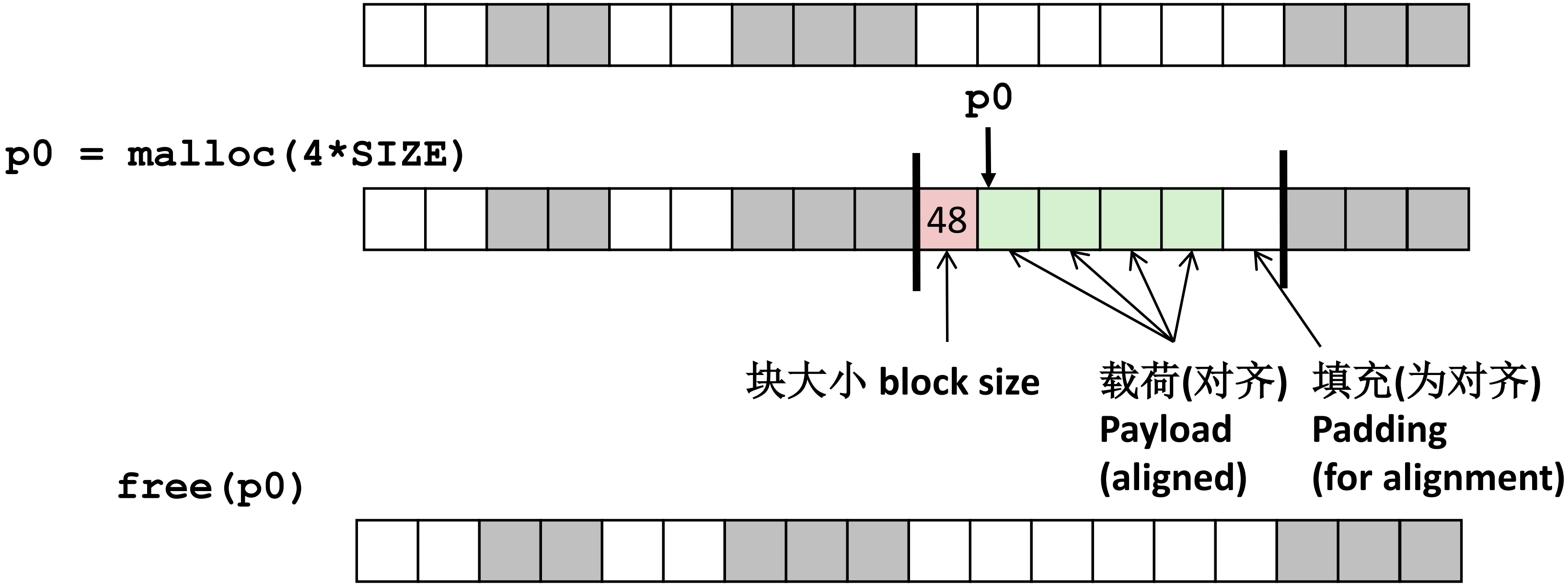
- 绿线： 由于外部碎片导致的额外堆大小 **Green line: additional heap size due to external fragmentation**
- 最佳匹配： 一种分配策略 **Best Fit: One allocation strategy**
  - （稍后讨论） (To be discussed later)
  - 总开销=本基准的8.3% Total overhead = 8.3% on this benchmark

实现问题 Implementation Issues

- 给定一个指针，我们怎么知道要释放多大的空间 **How do we know how much memory to free given just a pointer?**
- 我们怎么跟踪空闲块 **How do we keep track of the free blocks?**
- 当分配的结构大小小于选择的空闲块时怎么办？ **What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?**
- 当有多个块可用时我们应该怎么选？ **How do we pick a block to use for allocation -- many might fit?**
- 如何再次插入空闲块？ **How do we reinsert freed block?**

获取释放大小 Knowing How Much to Free

- 标准方法 **Standard method**
  - 在块之前的字中保存块长度 Keep the length (in bytes) of a block in the word *preceding* the block.
    - 包括头部 Including the header
    - 这个字称为头部域或者头部 This word is often called the ***header field*** or ***header***
  - 每个分配的块需要一个额外的字 Requires an extra word for every allocated block

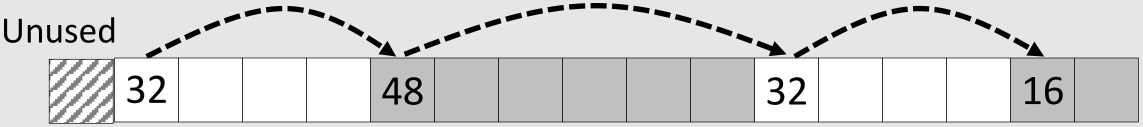


`p0 = malloc(4*SIZE)`

`free(p0)`

跟踪空闲块 Keeping Track of Free Blocks

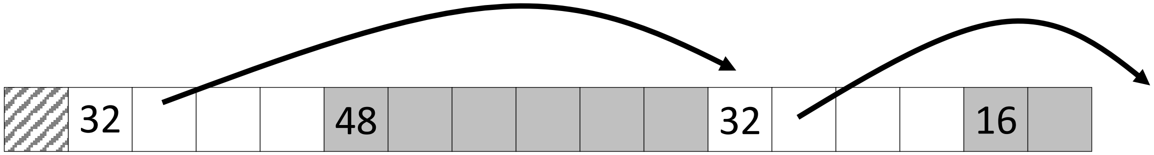
- **方法1：隐式链表-使用长度链接所有块 Method 1: *Implicit list* using length—links all blocks**



需要每个块标记为已分配/空闲

Need to tag each block as allocated/free

- **方法2：空闲块之间使用指针的显式链表 Method 2: *Explicit list* among the free blocks using pointers**



指针需要占空间

Need space for pointers

- **方法3：分离的空闲列表 Method 3: *Segregated free list***
  - 不同大小块使用不同的空闲列表 Different free lists for different size classes
- **方法4：根据大小对块排序 Method 4: *Blocks sorted by size***
  - 可以使用一个平衡树（红黑树），每个空闲块内有指针和做为键值的长度 Can use a balanced tree (e.g., Red-Black tree) with pointers within each free block, and the length used as a key

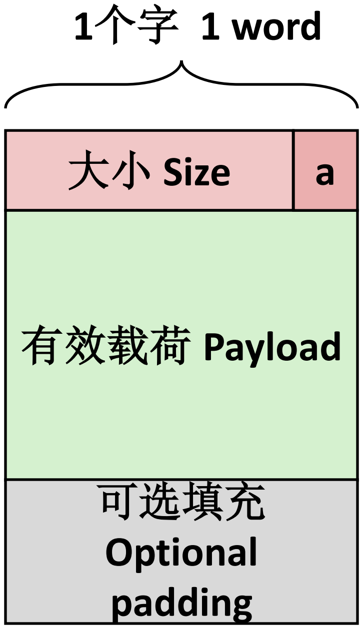
议题 Today

- 基本概念 Basic concepts
- **隐式空闲列表 Implicit free lists**

方法1：隐式空闲链表 Method 1: Implicit Free List

- 对每个块都需要大小和分配的状态 **For each block we need both size and allocation status**
  - 可以放在两个字中:浪费 Could store this information in two words: wasteful!
- 标准技巧 **Standard trick**
  - 如果块是对齐的，则地址低位部分总是0 If blocks are aligned, some low-order address bits are always 0
  - 与其存储0，还不如将其作为已分配/空闲的标志位 Instead of storing an always-0 bit, use it as a allocated/free flag
  - 读块大小那个字时需要将这些位屏蔽掉 When reading size word, must mask out this bit

已分配和空闲块格式 Format of allocated and free blocks

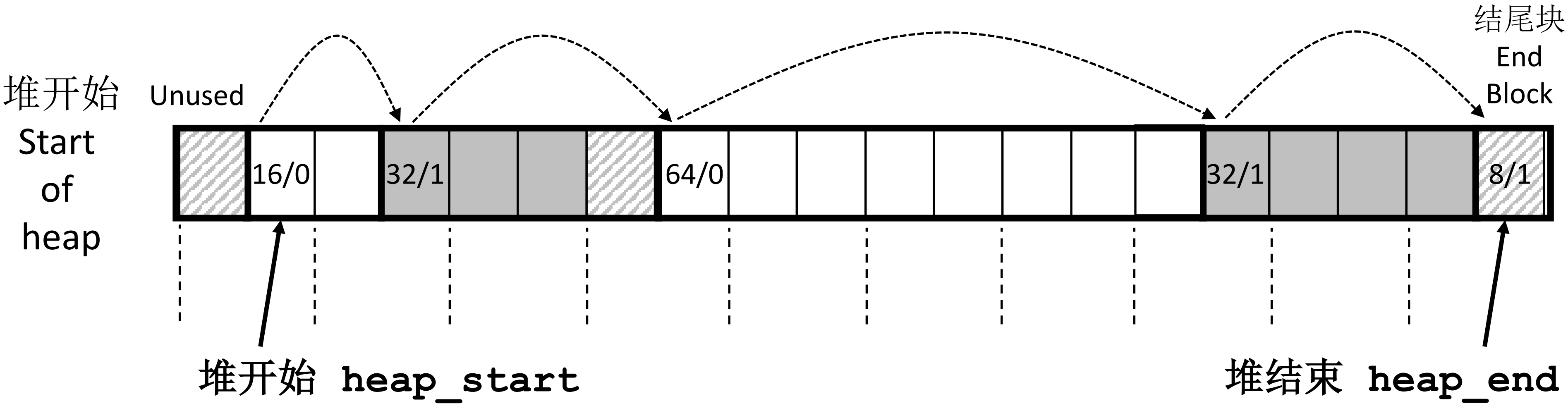


a = 1: Allocated block 分配的块
a = 0: Free block 空闲块

Size: block size 块大小

Payload: application data 载荷：应用数据（仅已分配的块）**(allocated blocks only)**

隐式空闲链表的详细例子
Detailed Implicit Free List Example



双字对齐
Double-word
aligned

已分配块:阴影 **Allocated blocks:** shaded
空闲块：无阴影 **Free blocks:** unshaded
头部：使用字节大小/分配位进行标记，头部不能位于非对齐位置 **Headers:** labeled with “size in words/allocated bit”
Headers are at non-aligned positions
➔ 有效载荷必须对齐 Payloads are aligned

隐式链表：数据结构
Implicit List: Data Structures

- 块声明 **Block declaration**

```
typedef uint64_t word_t;
```

```
typedef struct block
{
    word_t header;
    unsigned char payload[0];          // Zero length array
} block_t;
```

// `block_t *block`

- 从块指针获得有效载荷 **Getting payload from block pointer**

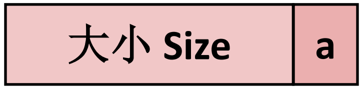
```
return (void *) (block->payload);
```

- 从有效载荷获得头部 **Getting header from payload** // bp points to a payload

```
return (block_t *) ((unsigned char *) bp
                     - offsetof(block_t, payload));
```

C语言函数offsetof(struct,member)返回member在struct中的偏移
C function `offsetof(struct, member)` returns offset of member within struct

隐式链表：访问头部
Implicit List: Header access



- 从头部获得分配位 **Getting allocated bit from header**

```
return header & 0x1;
```

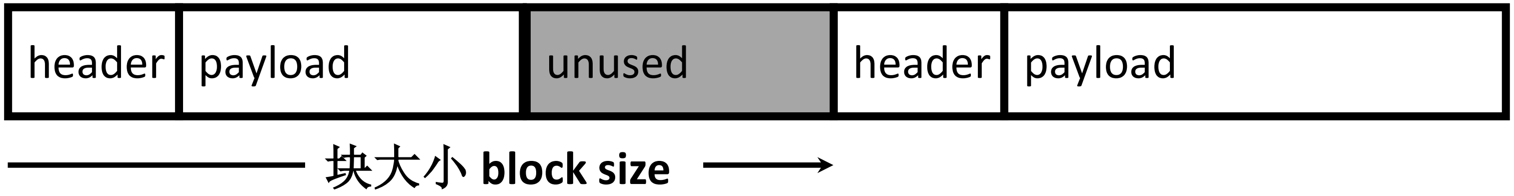
- 从头部获得块大小 **Getting size from header**

```
return header & ~0xfL;
```

- 初始化头部 **Initializing header** `//block_t *block`

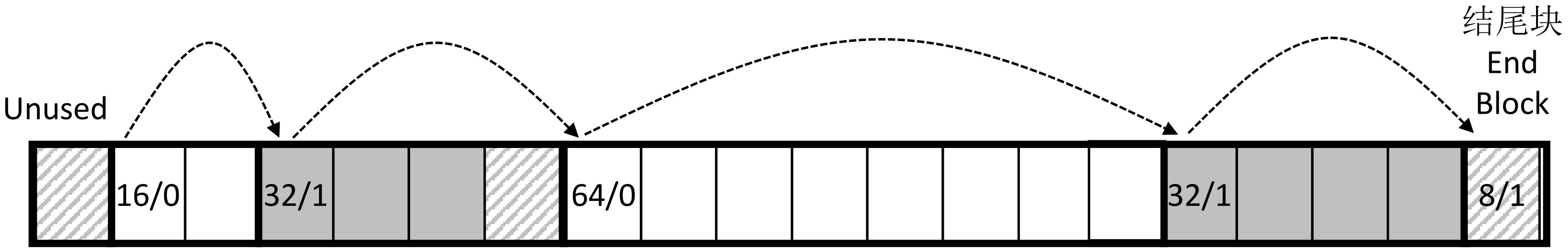
```
block->header = size | alloc;
```

隐式链表：遍历链表
Implicit List: Traversing list



- 查找下一个块 **Find next block**

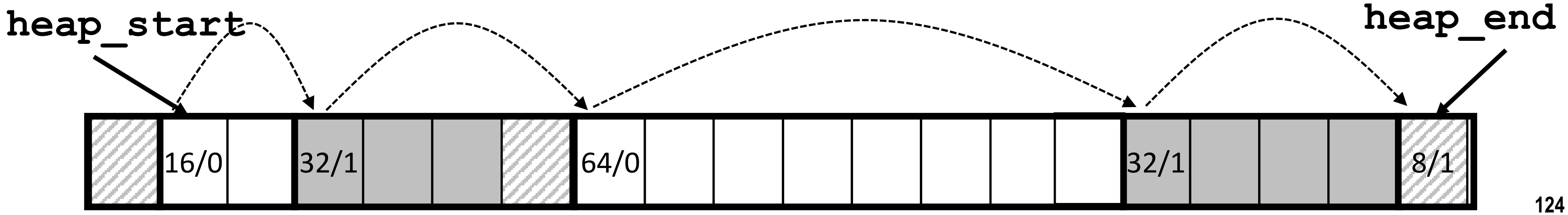
```
static block_t *find_next(block_t *block)
{
    return (block_t *) ((unsigned char *) block
                           + get_size(block));
}
```



隐式链表：查找空闲块
Implicit List: Finding a Free Block

- *首次匹配* ***First fit:***
  - 从链表开始搜索，选择第一个满足条件的空闲块 Search list from beginning, choose ***first*** free block that fits:
  - 查找asize字节的空间（包括头部） Finding space for **`asize`** bytes (including header):

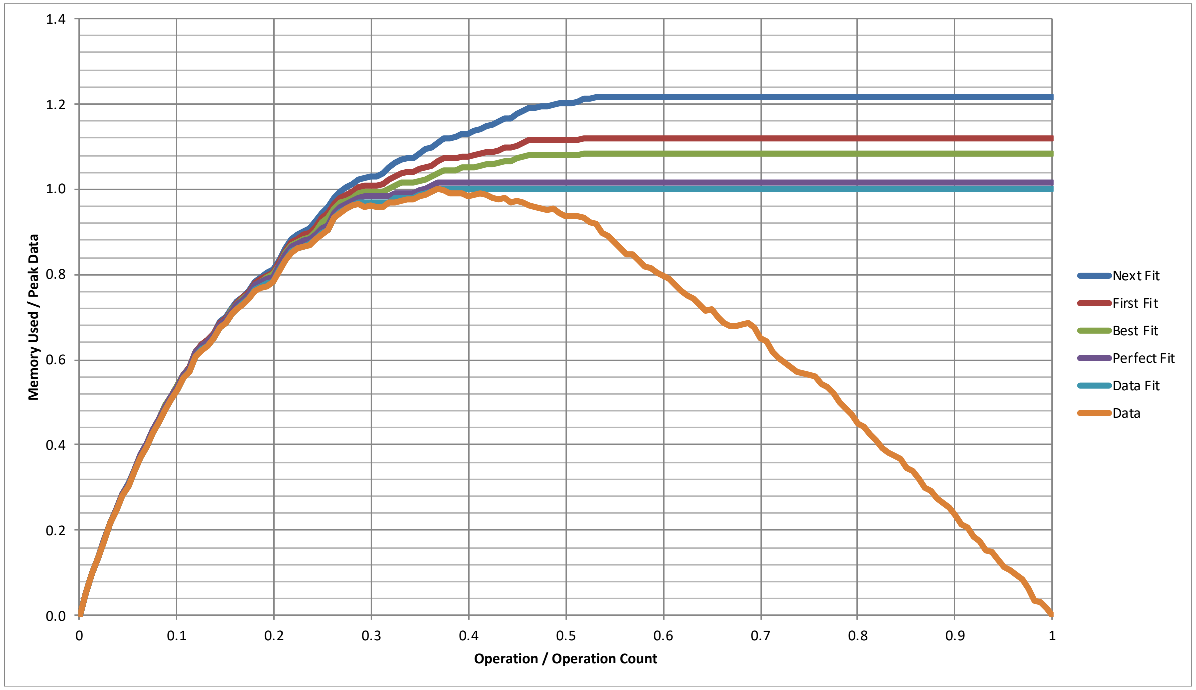
```
static block_t *find_fit(size_t asize)
{
    block_t *block;
    for (block = heap_start; block != heap_end;
         block = find_next(block)) {
    {
      if (!(get_alloc(block))
          && (asize <= get_size(block)))
         return block;
    }
    return NULL; // No fit found
}
```



隐式链表：查找空闲块 Implicit List: Finding a Free Block

- ***首次匹配： First fit:***
  - 从链表开始搜索，选择第一个满足条件的空闲块 Search list from beginning, choose ***first*** free block that fits:
  - 与总块数（分配和释放）成线性时间关系 Can take linear time in total number of blocks (allocated and free)
  - 实际上会在链表开始时造成碎片 In practice it can cause “splinters” at beginning of list
- ***下一次匹配： Next fit:***
  - 与first fit类似，但是从上一次搜索结束的位置开始查找 Like first fit, but search list starting where previous search finished
  - 一般会比first fit块：避免了重扫描无用的块 Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - 部分研究表明更容易造成内存碎片 Some research suggests that fragmentation is worse
- ***最佳匹配： Best fit:***
  - 从链表中选择最佳的空闲块：最小满足需求的块 Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - 保持内存碎片最小化-通常能改进内存利用率 Keeps fragments small—usually improves memory utilization
  - 一般会比first fit慢 Will typically run slower than first fit

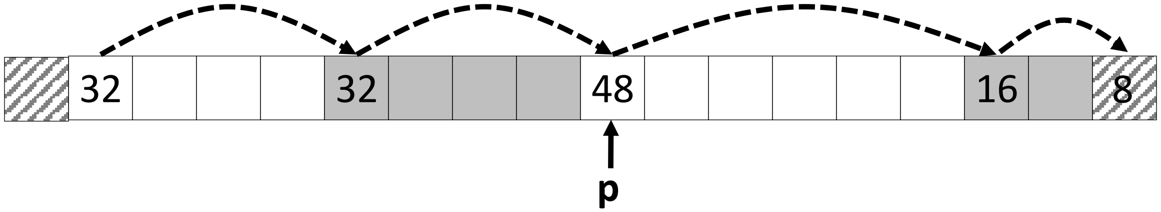
策略比较 Comparing Strategies



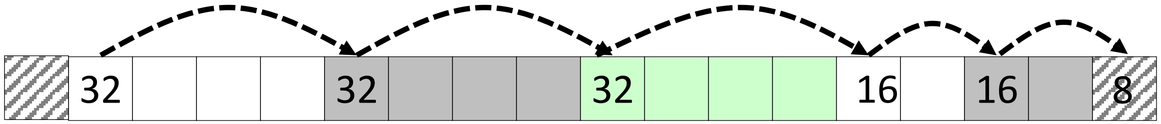
- 总开销（对本基准） **Total Overheads (for this benchmark)**
  - 完美比配 Perfect Fit: 1.6%
  - 最佳匹配 Best Fit: 8.3%
  - 首次匹配 First Fit: 11.9%
  - 下次匹配 Next Fit: 21.6%

隐式链表：从空闲块中分配
Implicit List: Allocating in Free Block

- 从一个空闲块分配：拆分 **Allocating in a free block: *splitting***
  - 由于分配的空间可能会比空闲空间小，因此可能会拆分空闲块
    Since allocated space might be smaller than free space, we might want to split the block

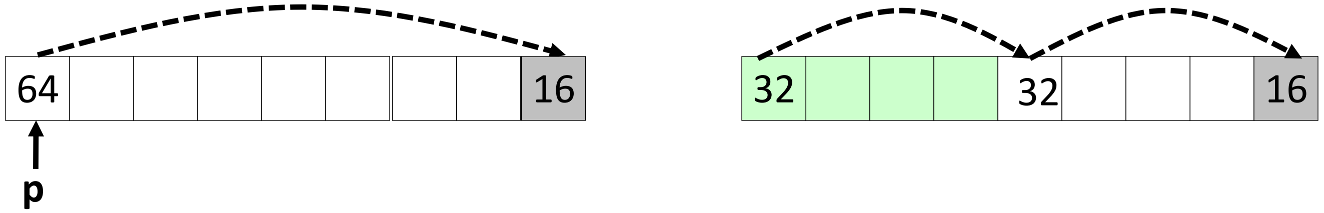


`split_block(p, 32)`



隐式链表：拆分空闲块
Implicit List: Splitting Free Block

`split_block(p, 32)`

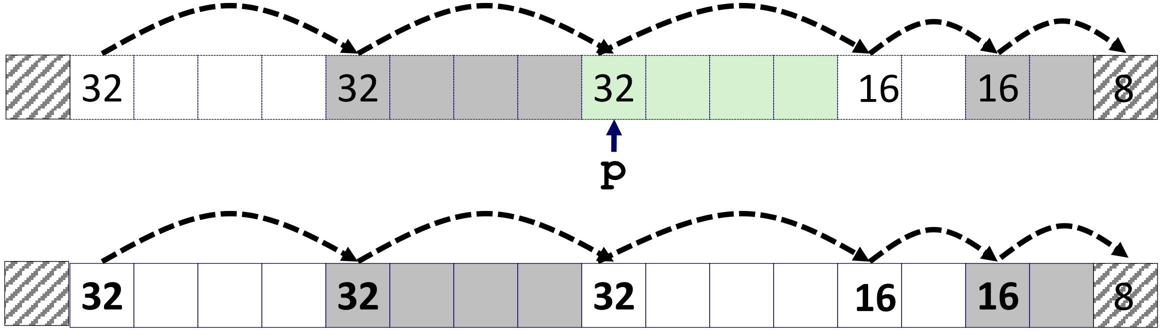


```
// Warning: This code is incomplete

static void split_block(block_t *block, size_t asize){
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
    }
}
```

隐式链表：释放一个块

Implicit List: Freeing a Block

- **最简单的实现 Simplest implementation:**
  - 只需要清除“已分配”标记位 Need only clear the “allocated” flag
  - 但是可能会导致“伪碎片” But can lead to “false fragmentation”



`free(p)`

`malloc(5*SIZ)`

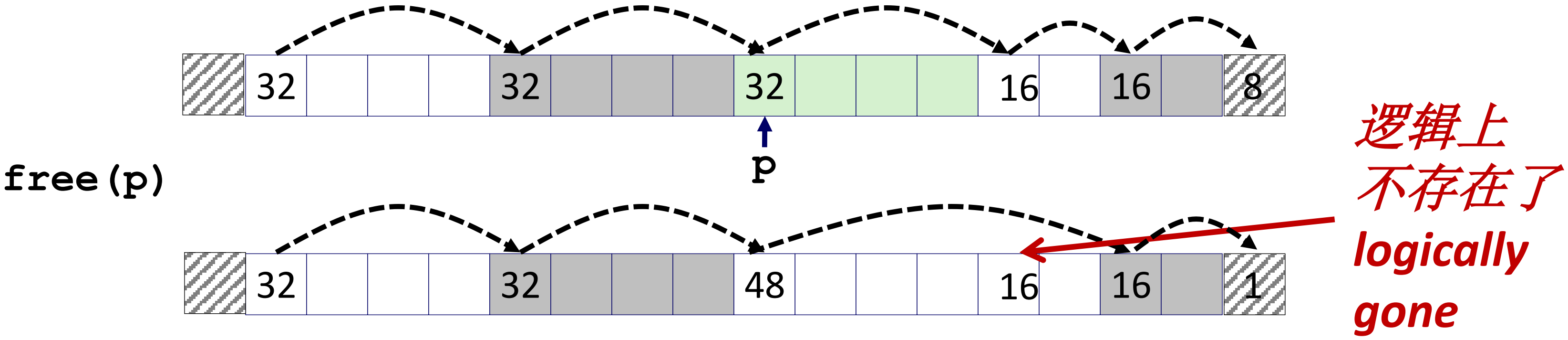
诶呀！ Yikes!

有足够的连续空闲空间，但是分配器找不到

There is enough contiguous free space, but the allocator won’t be able to find it

隐式链表：合并 Implicit List: Coalescing

- 与下一个/前一个空闲块*合并*，如果有空闲块 **Join *(coalesce)* with next/previous blocks, if they are free**
  - 与下一个块合并 Coalescing with next block

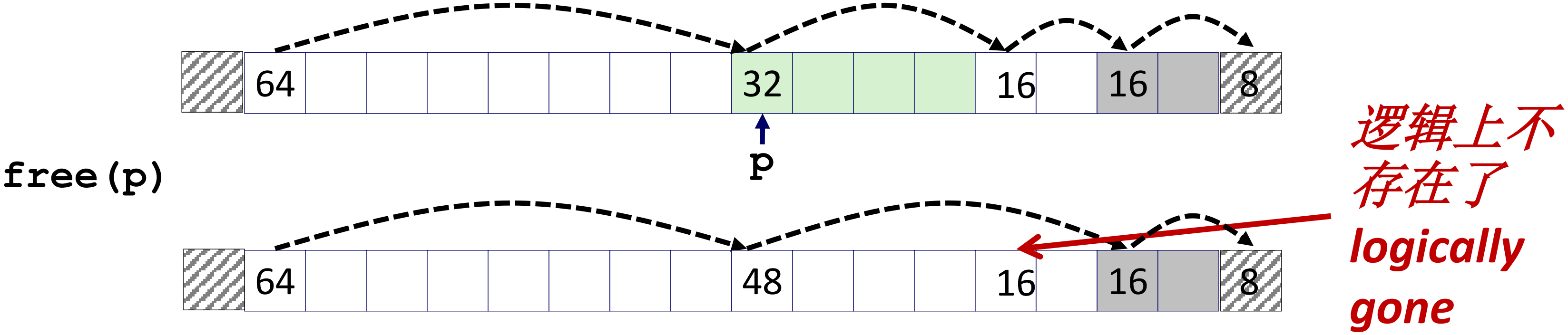


逻辑上不存在了

logically gone

隐式链表：合并 Implicit List: Coalescing

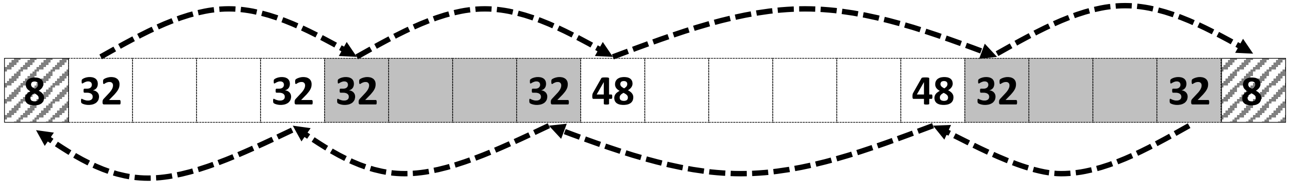
- 与下一个/前一个空闲块*合并*，如果有空闲块 **Join *(coalesce)* with next block, if it is free**
  - 与下一个块合并 Coalescing with next block



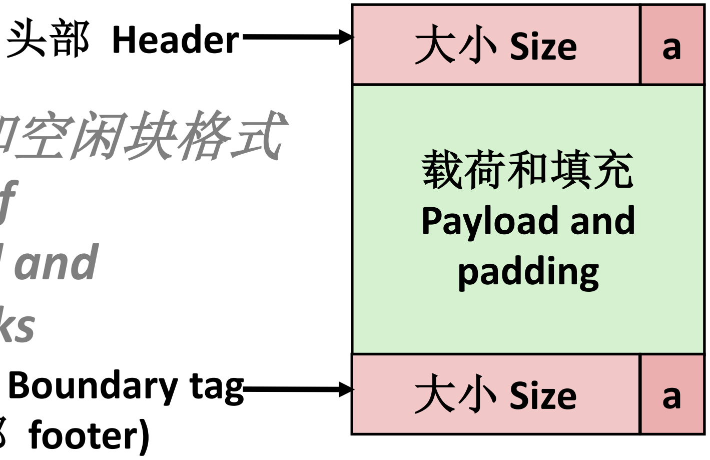
- 但是怎么和前一个块合并？ How do we coalesce with *previous* block?
  - 怎么知道从哪开始？ How do we know where it starts?
  - 怎么能确定是否已经分配出去了？ How can we determine whether its allocated?

隐式链表：双向合并
Implicit List: Bidirectional Coalescing

- *边界标记* ***Boundary tags*** [Knuth73]
  - 在空闲块“底部”（结束）的位置复制块大小/已分配字 Replicate size/allocated word at “bottom” (end) of free blocks
  - 以额外的空间换取反向遍历列表功能 Allows us to traverse the “list” backwards, but requires extra space
  - 重要和通用的技术 Important and general technique!



已分配和空闲块格式
Format of allocated and free blocks



头部 **Header**

大小 Size | **a**

载荷和填充
Payload and padding

大小 Size | **a**

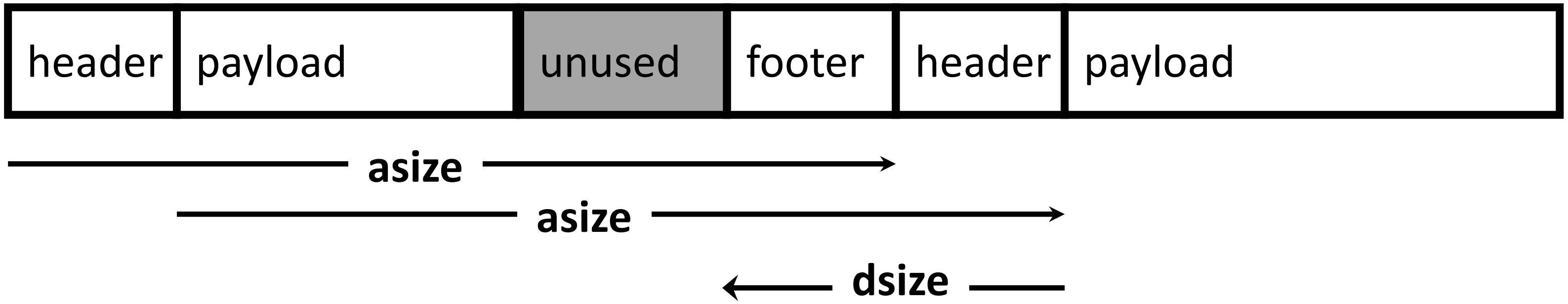
边界标记 **Boundary tag**
(脚部 **footer)**

a = 1: Allocated block 已分配块
a = 0: Free block 空闲块

Size: Total block size 大小：总的块大小

Payload: Application data 有效载荷：应用数据
(allocated blocks only) （仅已分配块）

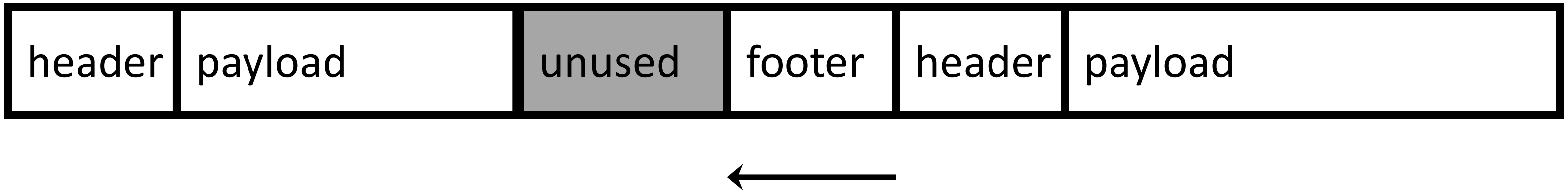
脚部的实现 Implementation with Footers



- 定位当前块的脚部 **Locating footer of current block**

```
const size_t dsize = 2*sizeof(word_t);

static word_t *header_to_footer(block_t *block)
{
    size_t asize = get_size(block);
    return (word_t *) (block->payload + asize - dsize);
}
```

脚部的实现 Implementation with Footers



1个字 1 word

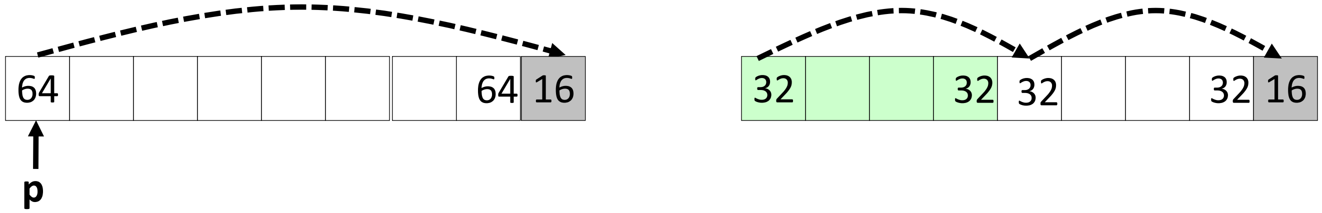
- 定位上一个块的脚部 **Locating footer of previous block**

```
static word_t *find_prev_footer(block_t *block)
{
    return &(block->header) - 1;
}
```

拆分空闲块：完整版本

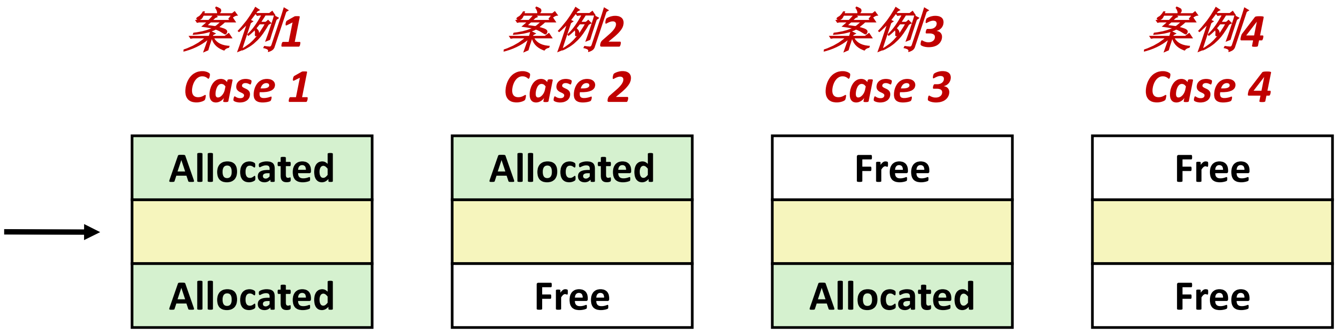
Splitting Free Block: Full Version

`split_block(p, 32)`



```
static void split_block(block_t *block, size_t asize){
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        write_footer(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
        write_footer(block_next, block_size - asize, false);
    }
}
```

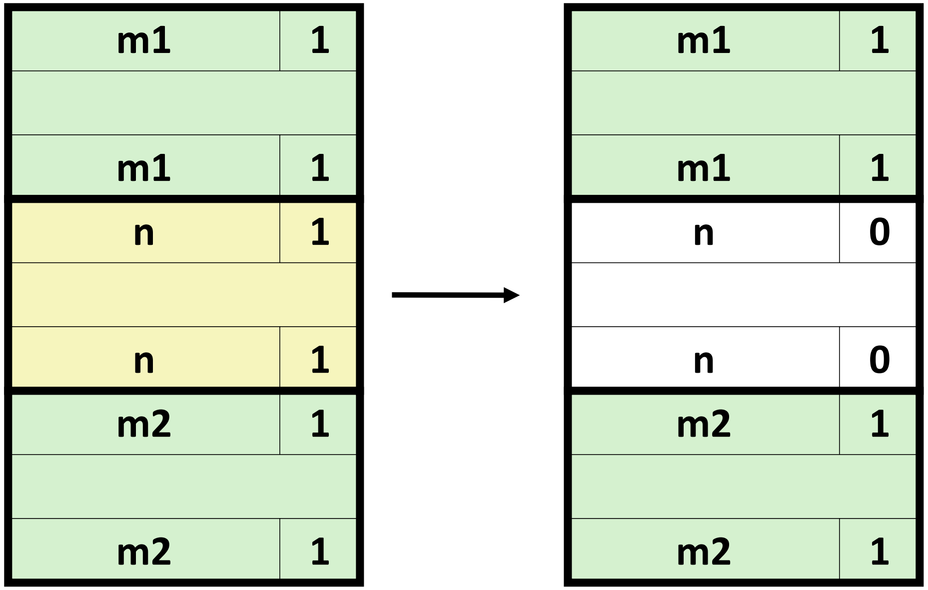
常量时间合并 Constant Time Coalescing



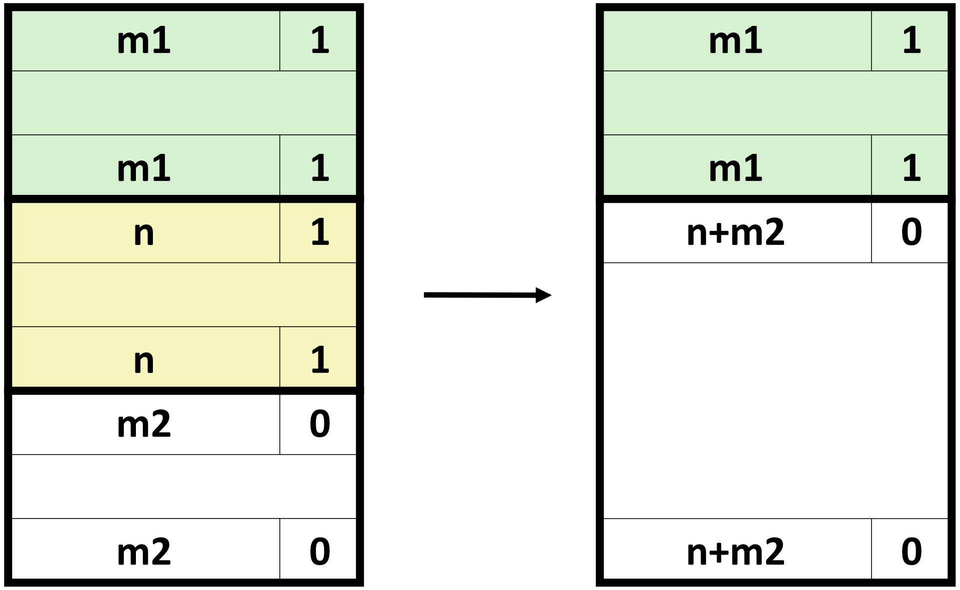
块被释放
Block being freed

案例1 *Case 1*	*案例2* *Case 2*	*案例3* *Case 3*	*案例4* *Case 4*
Allocated	Allocated	Free	Free
Allocated	Free	Allocated	Free

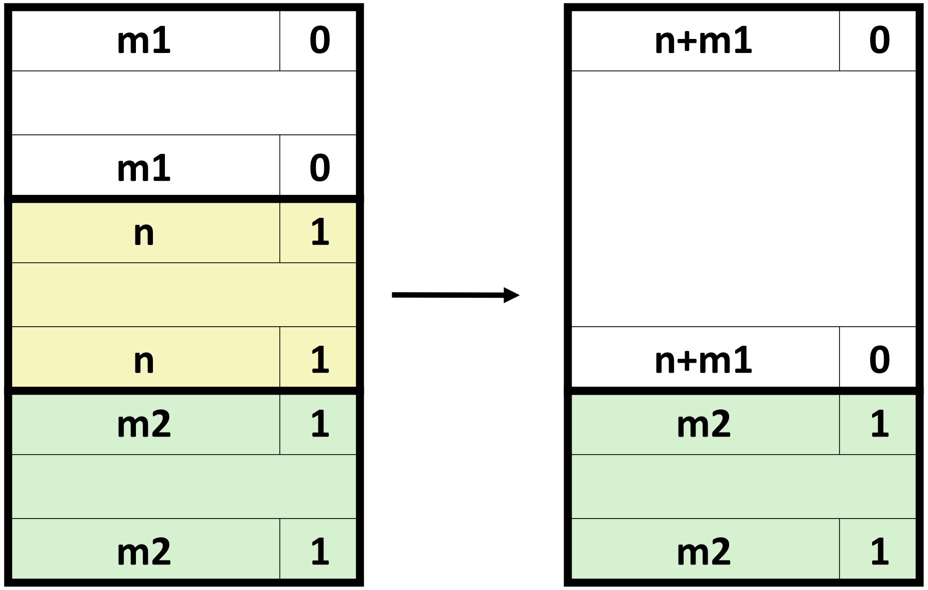
常量时间合并（案例1）
Constant Time Coalescing (Case 1)



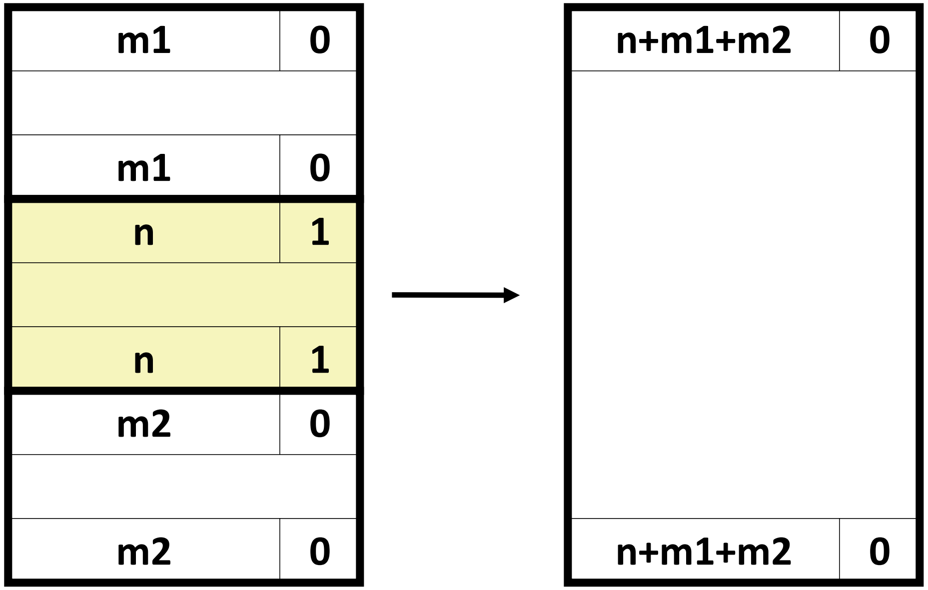
常量时间合并（案例2）
Constant Time Coalescing (Case 2)



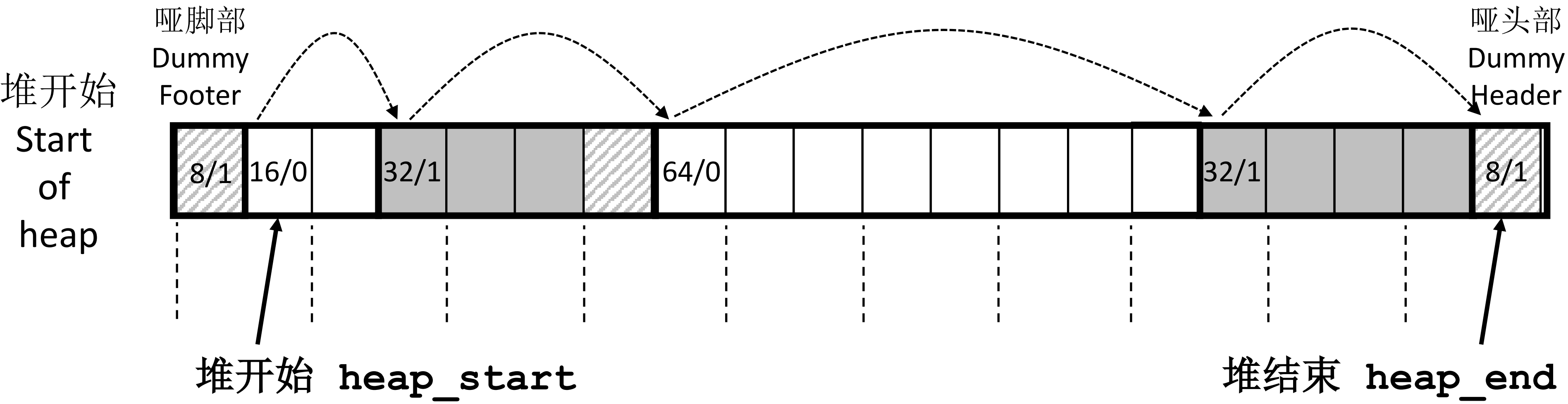
常量时间合并（案例3）
Constant Time Coalescing (Case 3)



常量时间合并（案例4）
Constant Time Coalescing (Case 4)



堆结构 Heap Structure



- 第一个头部之前的哑脚部 **Dummy footer before first header**
  - 标记为已分配 Marked as allocated
  - 当释放第一个块时，防止意外合并 Prevents accidental coalescing when freeing first block
- 最后脚部之后的哑头部 **Dummy header after last footer**
  - 在释放最后一块时，防止意外合并 Prevents accidental coalescing when freeing final block

顶层Malloc代码 Top-Level Malloc Code

```
const size_t dsize = 2*sizeof(word_t);

void *mm_malloc(size_t size)
{
    size_t asize = round_up(size + dsize, dsize);

    block_t *block = find_fit(asize);

    if (block == NULL)
        return NULL;

    size_t block_size = get_size(block);
    write_header(block, block_size, true);
    write_footer(block, block_size, true);

    split_block(block, asize);

    return header_to_payload(block);
}
```

```
round_up(n, m)
      =
m *((n+m-1)/m)
```

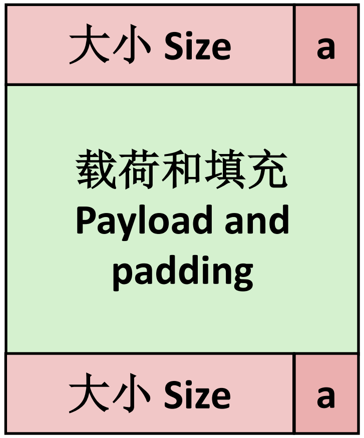
顶层Free代码 Top-Level Free Code

```
void mm_free(void *bp)
{
    block_t *block = payload_to_header(bp);
    size_t size = get_size(block);

    write_header(block, size, false);
    write_footer(block, size, false);

    coalesce_block(block);
}
```

边界标记的缺点
Disadvantages of Boundary Tags

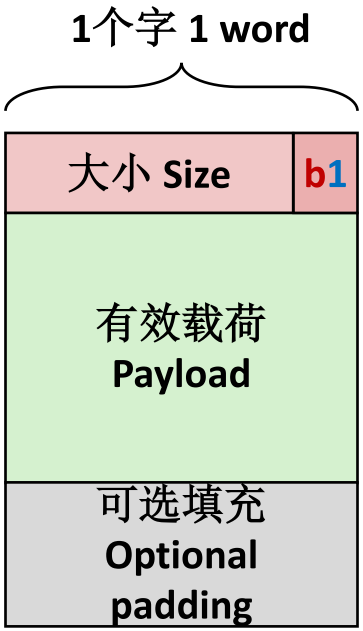
- 内部碎片 **Internal fragmentation**

- 可以进一步优化吗？ **Can it be optimized?**
  - 哪些块需要脚部标记? Which blocks need the footer tag?
  - 这意味着什么? What does that mean?



已分配块没有边界标记
No Boundary Tag for Allocated Blocks

- 仅空闲块需要边界标记 **Boundary tag needed only for free blocks**
- 当块大小是**16**的整倍数，存在**4**个空闲位 **When sizes are multiples of 16, have 4 spare bits**

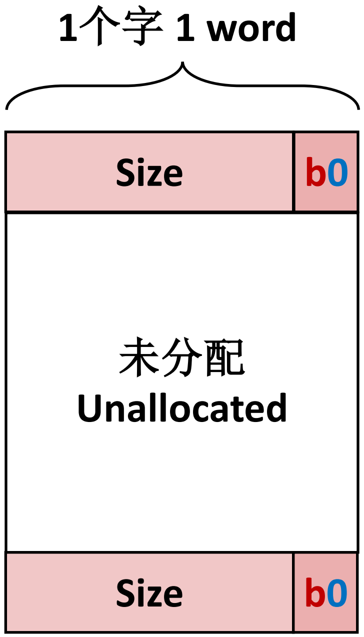


已分配块
Allocated
Block

a = 1: Allocated block 已分配
a = 0: Free block 空闲块
上一个块已分配
b = 1: Previous block is allocated
上一个块是空闲的
b = 0: Previous block is free

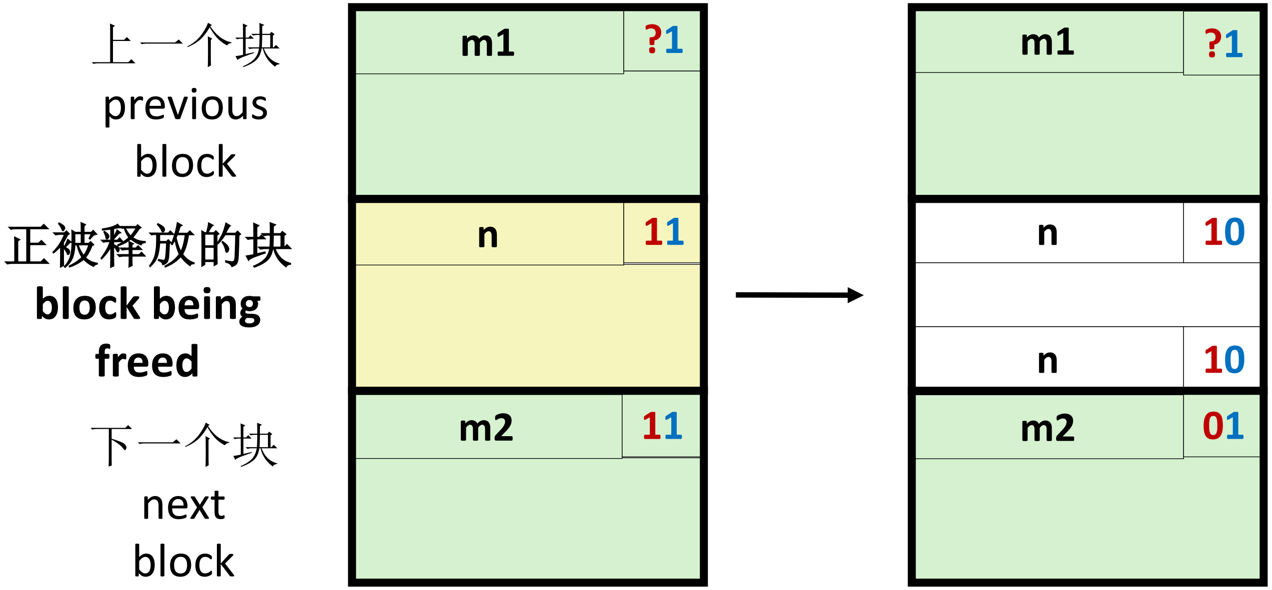
Size: block size 大小：块大小

Payload: application data 有效载荷：应用数据



空闲块
Free
Block

已分配块没有边界标记（案例1）
No Boundary Tag for Allocated Blocks (Case 1)



上一个块
previous
block

正被释放的块
block being
freed

下一个块
next
block

m1 **?1**

n **11**

m2 **11**

→

m1 **?1**

n **10**

n **10**

m2 **01**

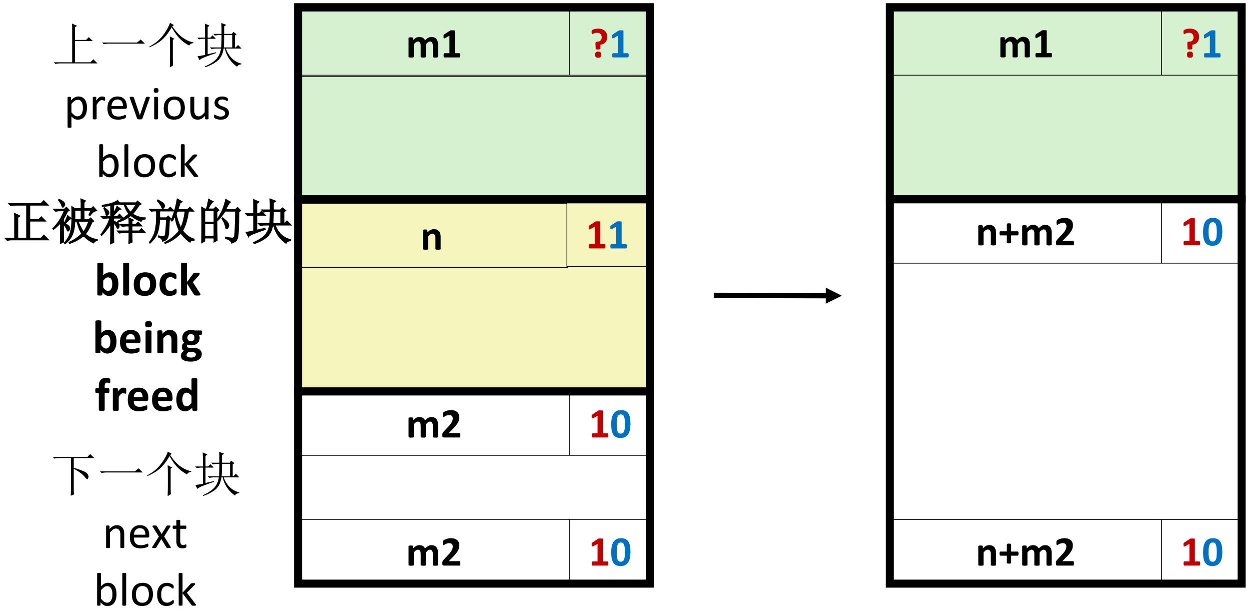
头部：使用2位（由于对齐的原因，这两个地址位始终为零）

Header: Use 2 bits (address bits always zero due to alignment):

上一个分配的块<<1 | 当前分配的块

(previous block allocated)<<1 | (current block allocated)

已分配块没有边界标记（案例2）
No Boundary Tag for Allocated Blocks (Case 2)



上一个块
previous
block

正被释放的块
block being freed

下一个块
next
block

m1 | ?1

n | 11

m2 | 10

m2 | 10

→

m1 | ?1

n+m2 | 10

n+m2 | 10

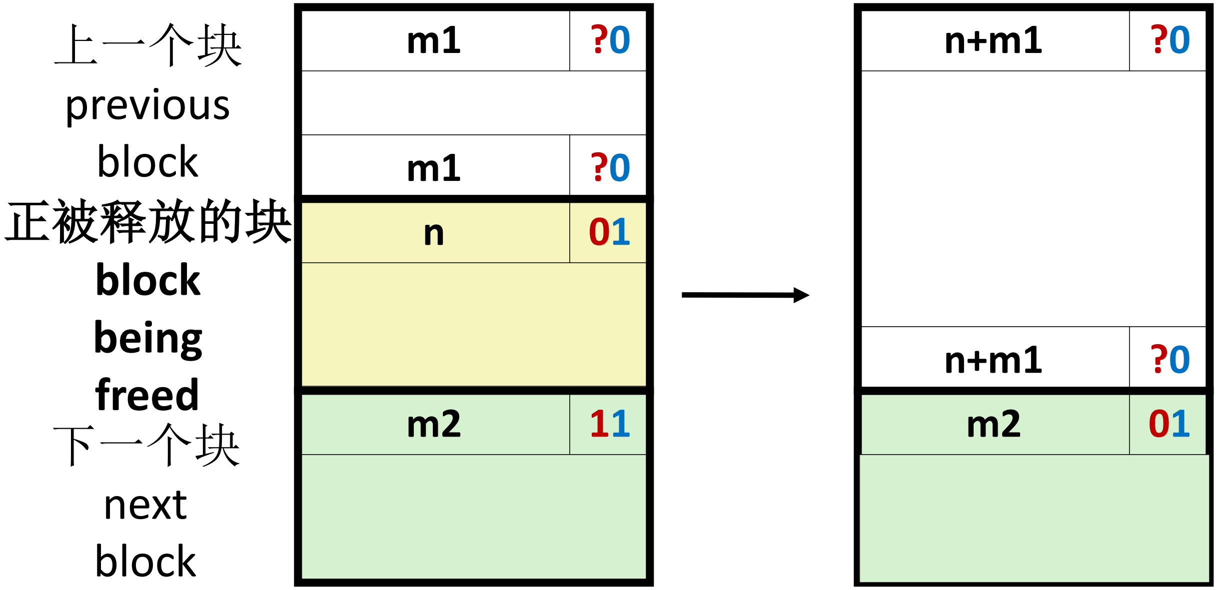
头部：使用2位（由于对齐的原因，这两个地址位始终为零）

Header: Use 2 bits (address bits always zero due to alignment):

上一个分配的块<<1 | 当前分配的块

(previous block allocated)<<1 | (current block allocated)

已分配块没有边界标记（案例3）
No Boundary Tag for Allocated Blocks (Case 3)



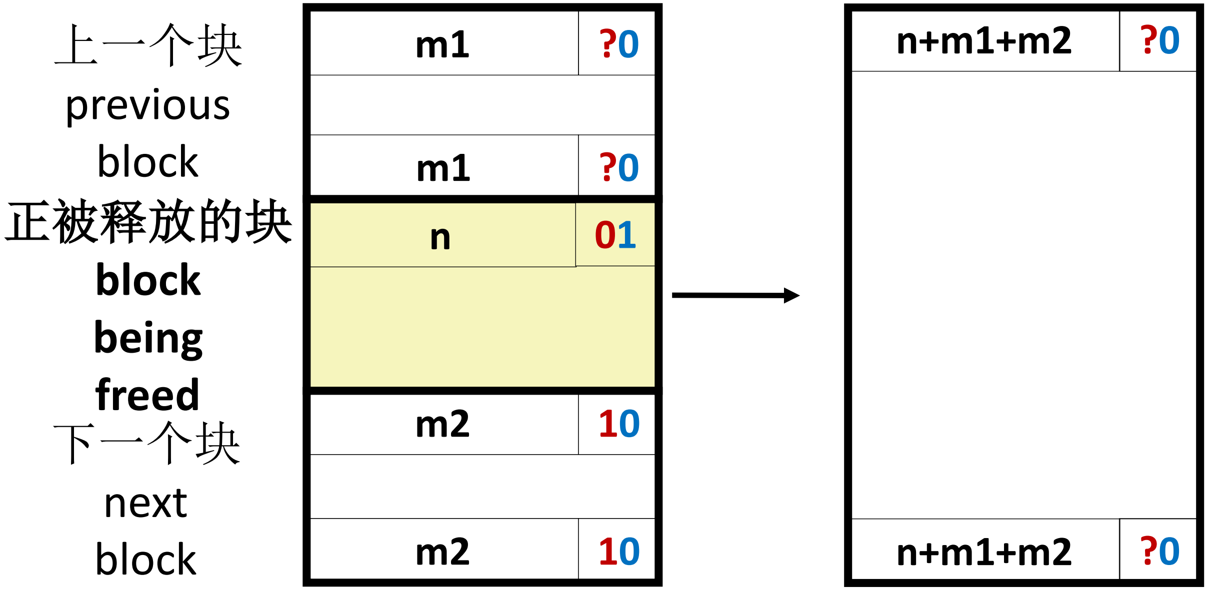
头部：使用2位（由于对齐的原因，这两个地址位始终为零）

Header: Use 2 bits (address bits always zero due to alignment):

上一个分配的块<<1 | 当前分配的块

(previous block allocated)<<1 | (current block allocated)

已分配块没有边界标记（案例4）
No Boundary Tag for Allocated Blocks (Case 4)



头部：使用2位（由于对齐的原因，这两个地址位始终为零）

Header: Use 2 bits (address bits always zero due to alignment):

上一个分配的块<<1 | 当前分配的块

(previous block allocated)<<1 | (current block allocated)

主要分配策略总结
Summary of Key Allocator Policies

- 选择策略 **Placement policy:**
  - 首次匹配、下一次匹配、最佳匹配等 First-fit, next-fit, best-fit, etc.
  - 在更低吞吐率和更少的碎片之间平衡 Trades off lower throughput for less fragmentation
  - ***有趣的观察***：分离的空闲列表与最优选择策略接近，且不用搜索整个链表 ***Interesting observation***: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list

- 拆分策略： **Splitting policy:**
  - 什么时候需要拆分空闲块？ When do we go ahead and split free blocks?
  - 我们可能容忍多少内部碎片？ How much internal fragmentation are we willing to tolerate?

- 合并策略： **Coalescing policy:**
  - ***立即合并:*** 每次free时合并 ***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***延迟合并:*** 为了提升free的性能，当需要时再合并，例如： ***Deferred coalescing:*** try to improve performance of **`free`** by deferring coalescing until needed. Examples:
    - 由于malloc扫描空闲列表时进行合并 Coalesce as you scan the free list for **`malloc`**
    - 当外部碎片超过某个阈值时进行合并 Coalesce when the amount of external fragmentation reaches some threshold

隐式列表：总结 Implicit Lists: Summary

- 实现：非常简单 **Implementation: very simple**
- 分配开销： **Allocate cost:**
  - 最差是线性时间 linear time worst case
- 释放开销： **Free cost:**
  - 最差常量时间 constant time worst case
  - 甚至包括合并 even with coalescing
- 内存使用 **Memory usage:**
  - 依赖于选择策略 will depend on placement policy
  - 首次匹配、下一次匹配或最佳匹配 First-fit, next-fit or best-fit

- 由于线性时间的分配开销，实际**malloc**和**free**并没有使用 **Not used in practice for `malloc/free` because of linear-time allocation**
  - 在很多特殊目的的应用中使用 used in many special purpose applications

- 然而拆分和基于边界标记的合并的概念对所有的分配器都是适用的 **However, the concepts of splitting and boundary tag coalescing are general to *all* allocators**

第9章 虚拟内存

Dynamic Memory Allocation: Advanced Concepts

动态存储分配:高级概念

100076202： 计算机系统导论

任课教师:

宿红毅 张艳 黎有琦 颜珂

原作者:

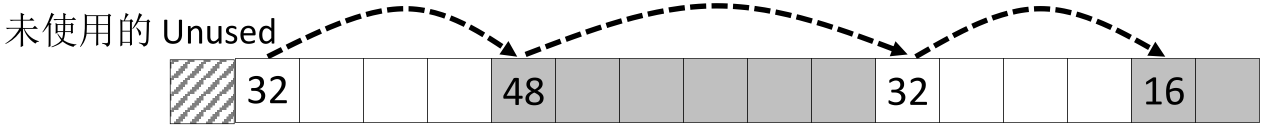
Randal E. Bryant and David R. O’Hallaron

议题 Today

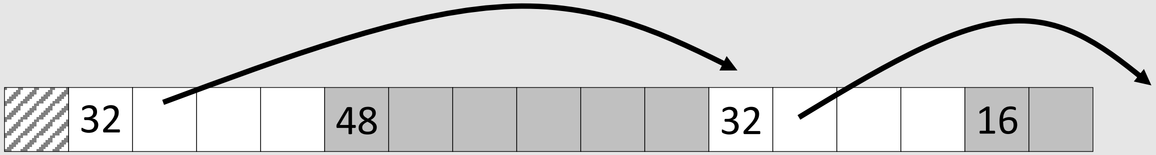
- **显式空闲列表 Explicit free lists**
- 分离的空闲列表 Segregated free lists
- 垃圾收集 Garbage collection
- 内存相关的风险和陷阱 Memory-related perils and pitfalls

跟踪空闲块 Keeping Track of Free Blocks

- 方法1：隐式空闲列表使用长度链接所有块 **Method 1: *Implicit free list* using length—links all blocks**



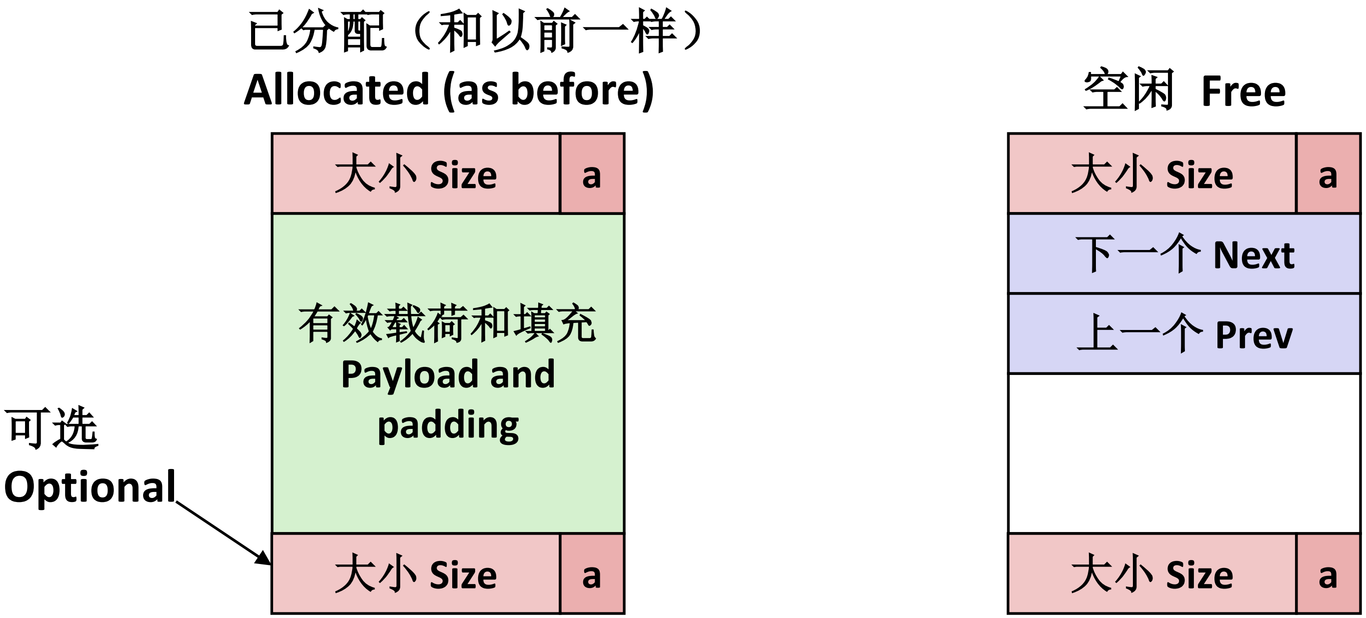
- 方法2：显式空闲列表使用指针串接空闲块 **Method 2: *Explicit free list* among the free blocks using pointers**



- 方法3：分离的空闲列表 **Method 3: *Segregated free list***
  - 不同大小的块使用不同的列表管理 Different free lists for different size classes

- 方法4：根据大小排序块 **Method 4: *Blocks sorted by size***
  - 使用平衡红黑树，每个空闲块内包含指针和用作键值的长度 Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

显式空闲列表 Explicit Free Lists

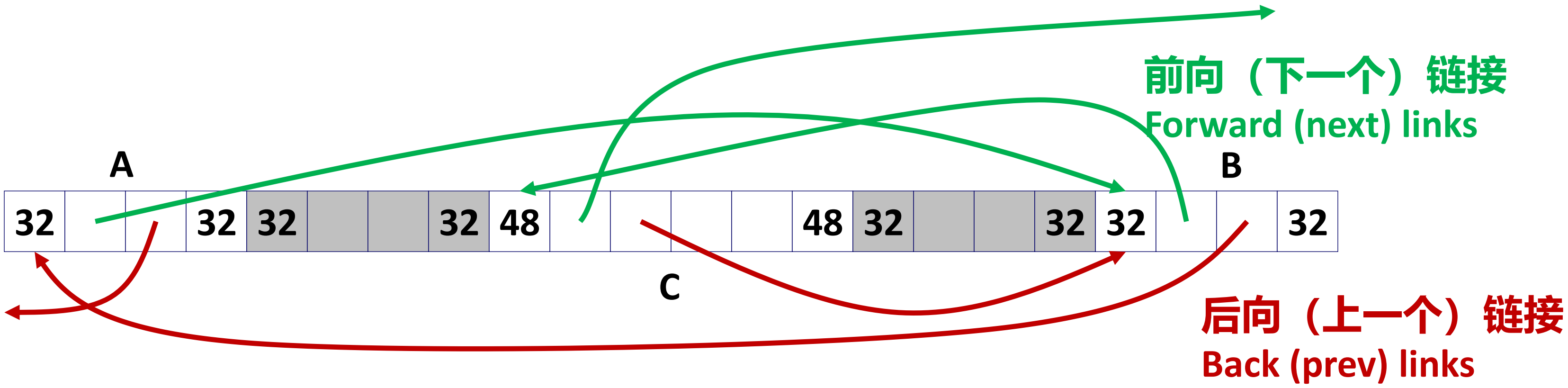


- 维护空闲块列表，而不是所有块 **Maintain list(s) of *free* blocks, not *all* blocks**
  - 下一个空闲块可能在任一地方 The “next” free block could be anywhere
    - 所以需要存储前向/后向指针，不只是大小 So we need to store forward/back pointers, not just sizes
  - 仍然需要使用边界标记进行合并 Still need boundary tags for coalescing
    - 根据内存顺序发现邻接块 To find adjacent blocks according to memory order
  - 幸运的是我们只需要跟踪空闲块，所以可以使用有效载荷区域 Luckily we track only free blocks, so we can use payload area

显式空闲列表 Explicit Free Lists

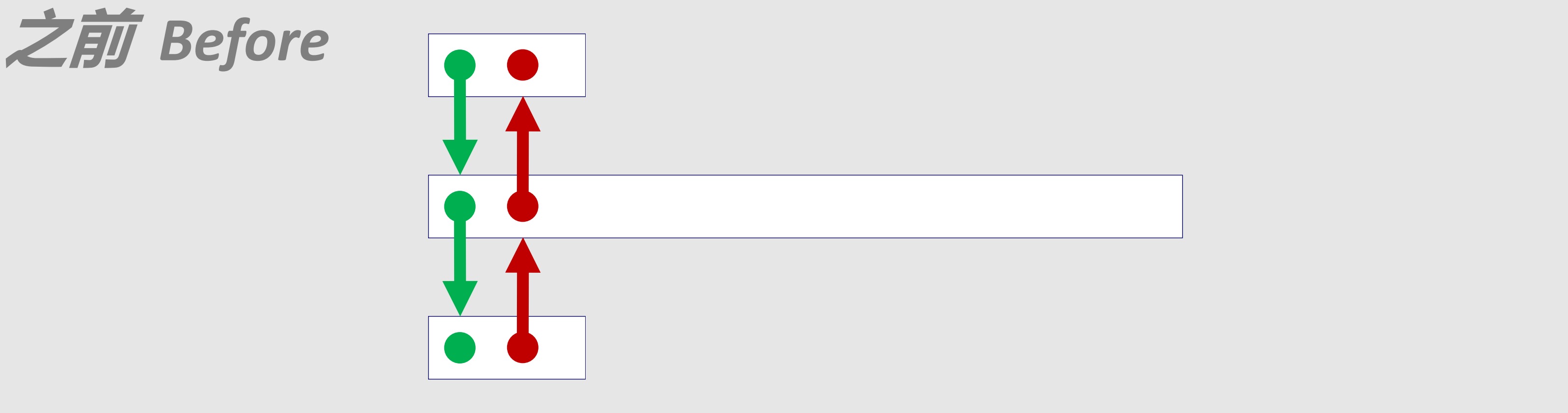
- 逻辑上 **Logically:**

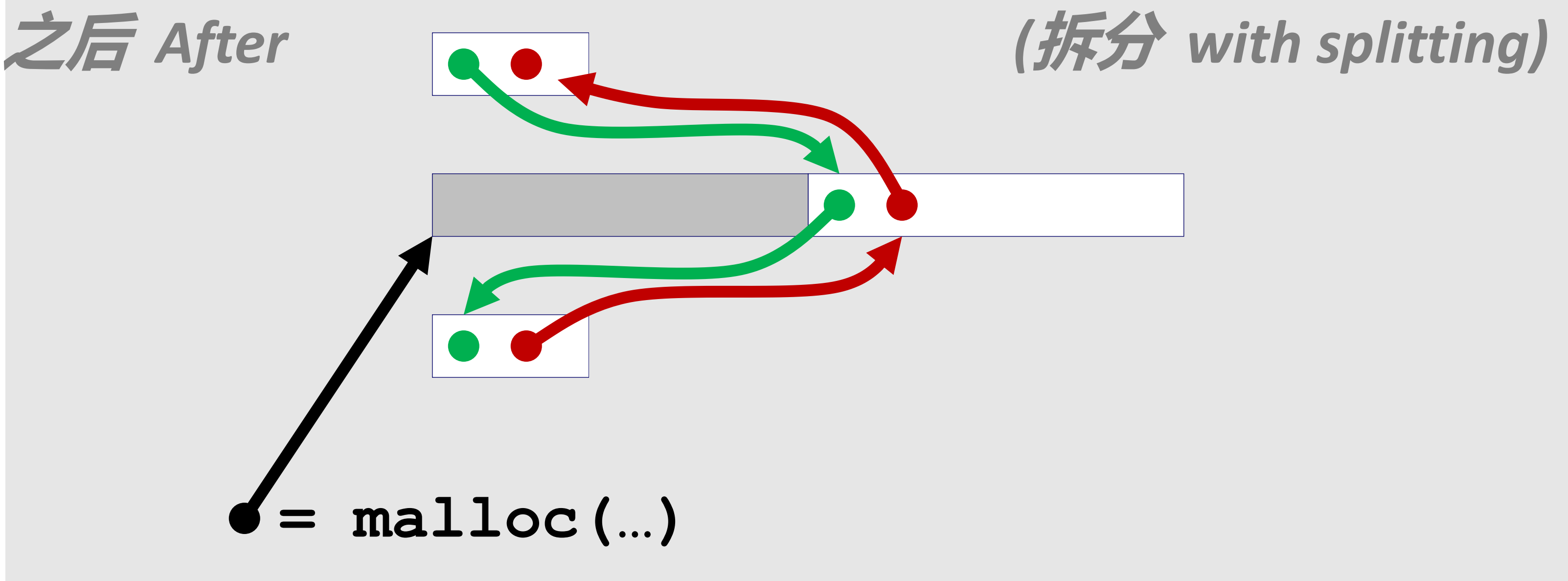
- 物理上：块可能是任意顺序 **Physically: blocks can be in any order**



从显式空闲列表分配
Allocating From Explicit Free Lists

概念图 conceptual graphic



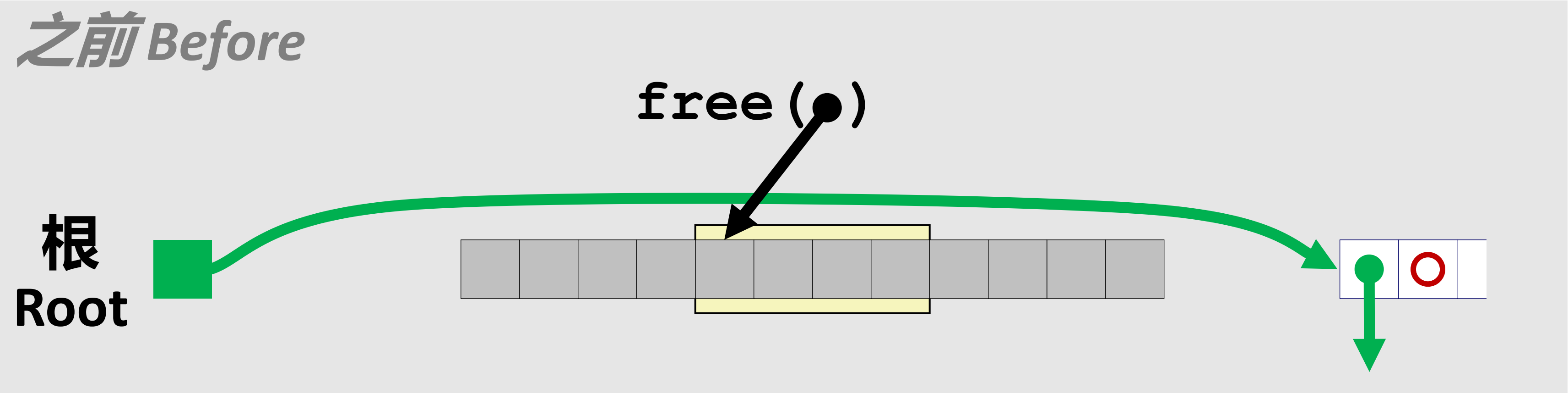


释放空闲块到显式空闲列表
Freeing With Explicit Free Lists

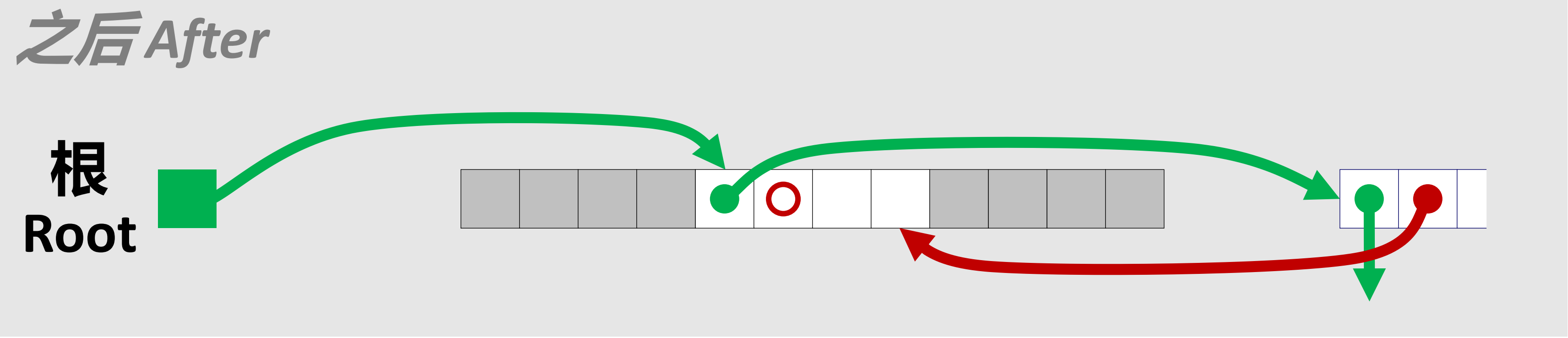
- ***插入策略：*** 在空闲列表的什么位置插入一个新的空闲块？ ***Insertion policy*: Where in the free list do you put a newly freed block?**
- 后进先出策略 **LIFO (last-in-first-out) policy**
  - 在空闲列表的开始插入空闲块 Insert freed block at the beginning of the free list
  - ***优点：*** 简单并且常数时间完成 ***Pro:*** simple and constant time
  - ***缺点：*** 研究表明比地址排序导致更多的碎片 ***Con:*** studies suggest fragmentation is worse than address ordered
- 地址排序策略 **Address-ordered policy**
  - 插入空闲块以便空闲列表块始终按地址排序 Insert freed blocks so that free list blocks are always in address order:

    addr(prev) < addr(curr) < addr(next)
  - ***缺点：*** 需要搜索 ***Con:*** requires search
  - ***优点：*** 研究表明比LIFO有低的内存碎片 ***Pro:*** studies suggest fragmentation is lower than LIFO

基于LIFO策略的释放（案例1）
Freeing With a LIFO Policy (Case 1)



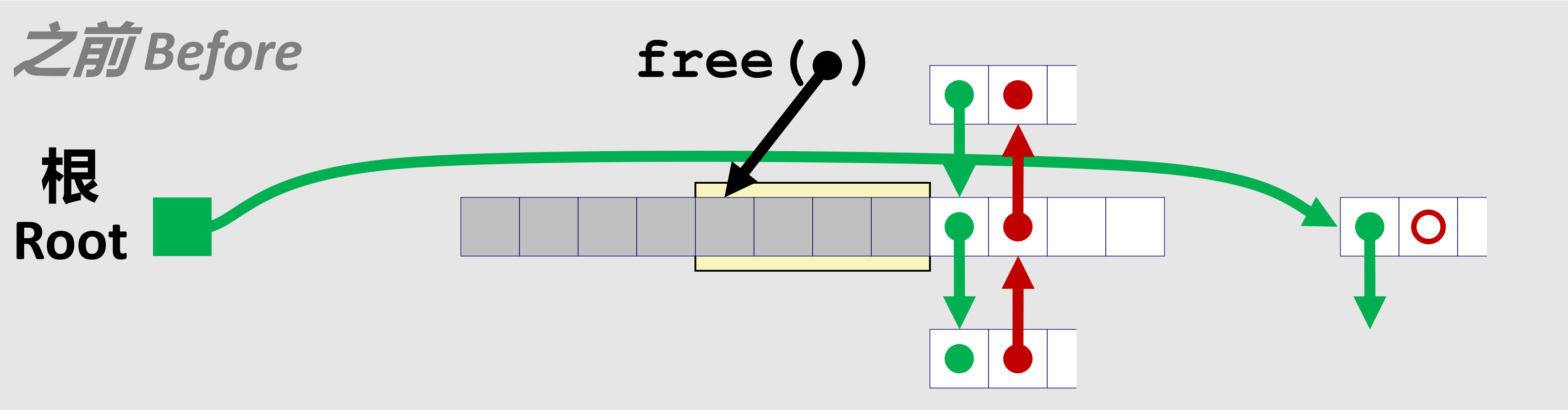
- 将空闲块插入到列表头 Insert the freed block at the root of the list



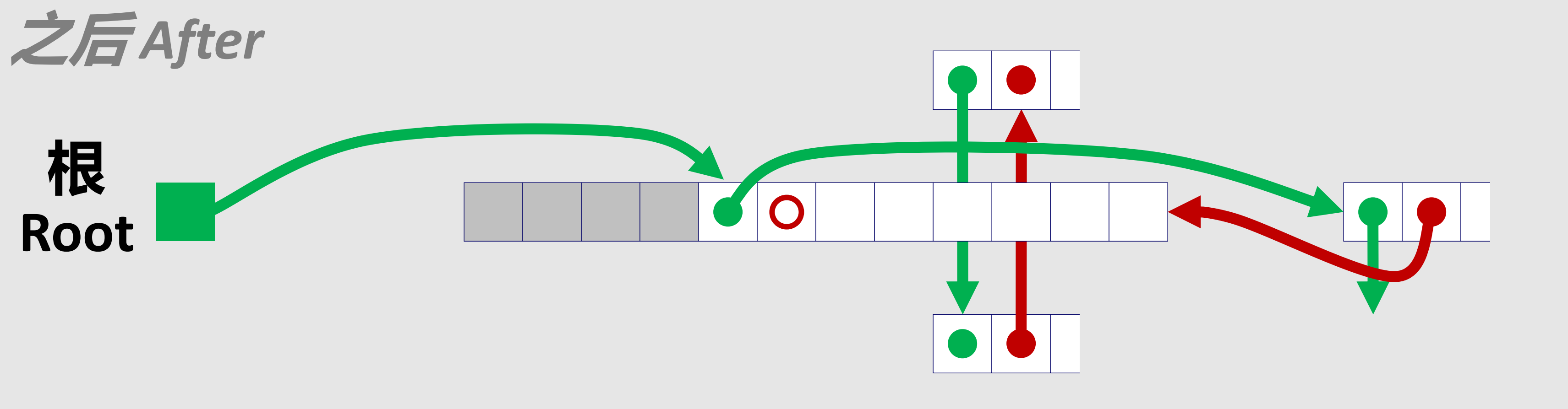
基于LIFO策略的释放（案例2）
Freeing With a LIFO Policy (Case 2)

已分配 Allocated 🟨 空闲 Free

概念图 conceptual graphic

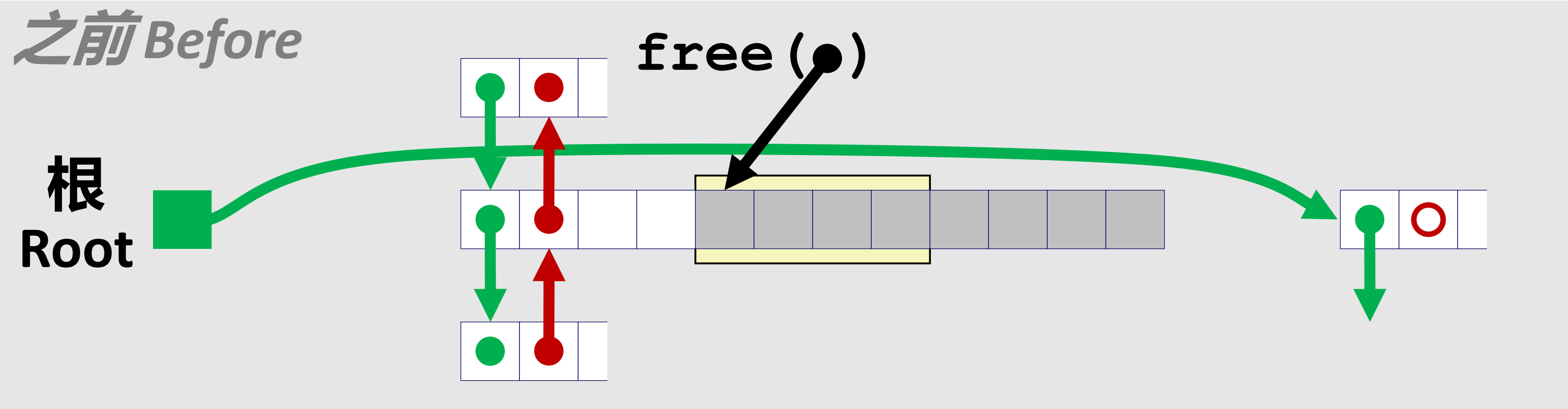


- 拼接出后续块，合并两个块并在列表头插入新块 **Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list**

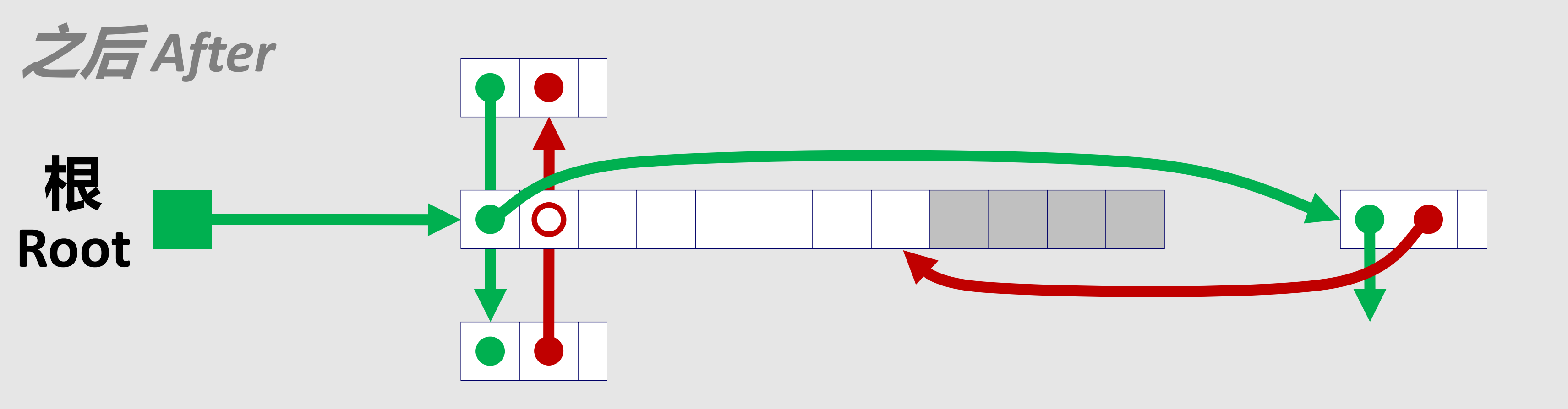


基于LIFO策略的释放（案例3）
Freeing With a LIFO Policy (Case 3)

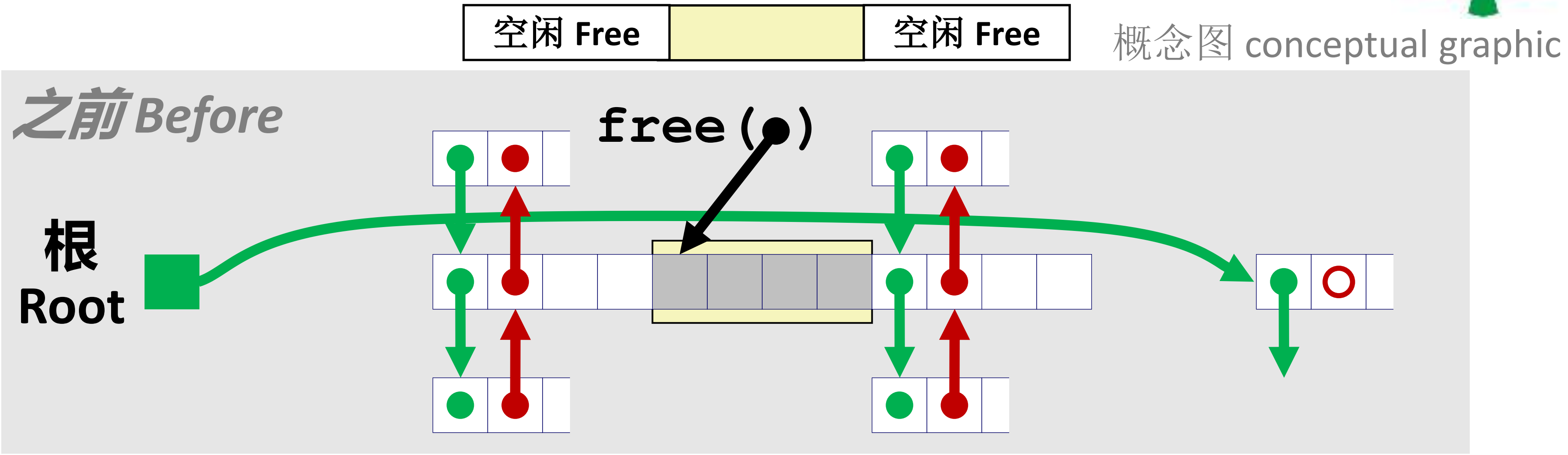
空闲 Free 🟨 已分配Allocated 概念图 conceptual graphic



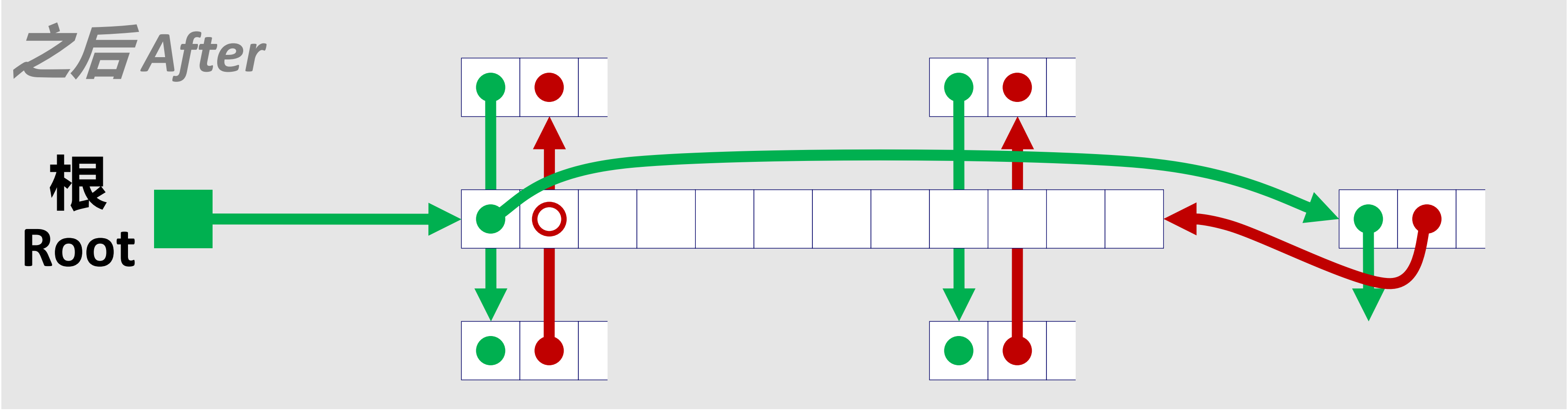
- 拼接出前驱块，合并两个块并在列表头插入新块 Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



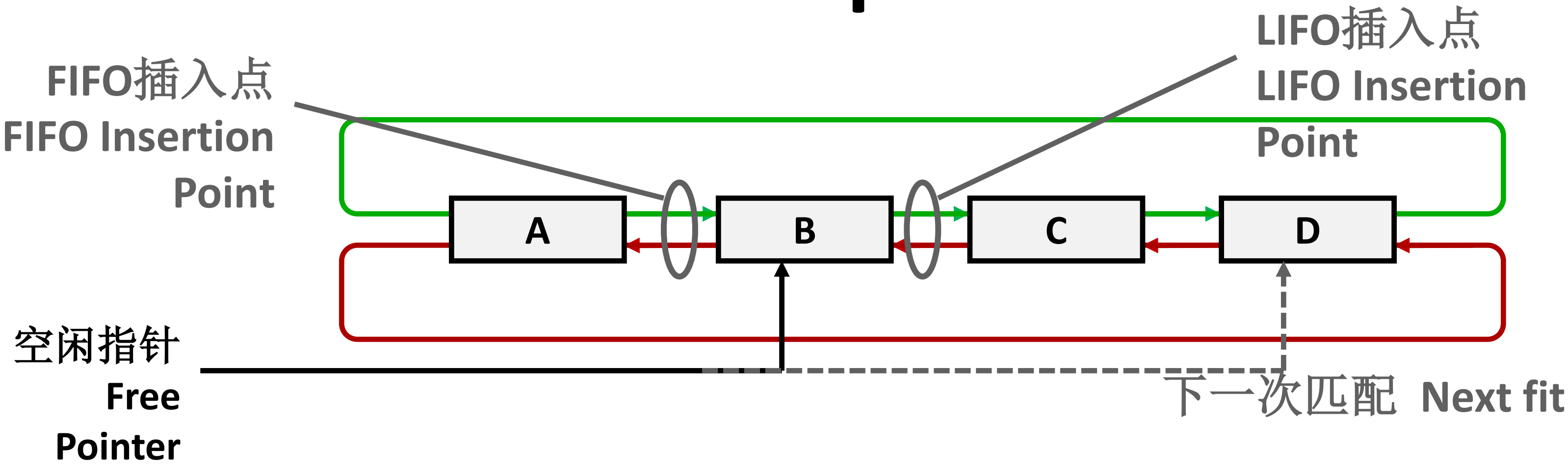
基于LIFO策略的释放（案例4）
Freeing With a LIFO Policy (Case 4)



- 拼接出前驱和后继块，合并三个块并在列表头插入新块 **Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list**



一些建议：实现技巧
Some Advice: An Implementation Trick



- 使用循环双向链表 **Use circular, doubly-linked list**
- 用单一数据结构支持多种方法 **Support multiple approaches with single data structure**
- 首次匹配对下一次匹配 **First-fit vs. next-fit**
  - 要么保持空闲指针固定，要么随搜索列表移动 Either keep free pointer fixed or move as search list
- 后进先出对先进先出 **LIFO vs. FIFO**
  - 插入做为下一个块（LIFO）或做为上一个块 Insert as next block (LIFO), or previous block (FIFO)

显式列表总结 Explicit List Summary

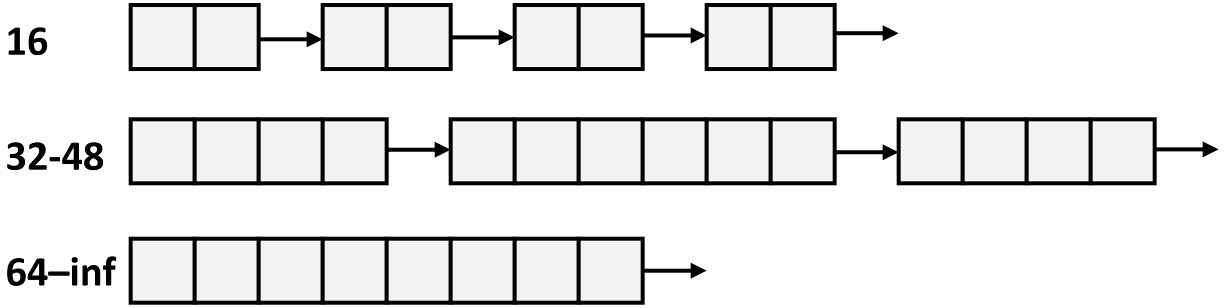
- **与隐式列表相比 Comparison to implicit list:**
  - 分配时间与空闲块的数量成线性时间，而不是所有的块 Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - 当内存大部分被占用的时候快很多 ***Much faster*** when most of the memory is full
  - 由于需要从列表中删除和向链表中插入块，分配和释放稍微复杂一些 Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - 链接需要一些额外的空间（每个块需要2个额外的字） Some extra space for the links (2 extra words needed for each block)
    - 会增加内部碎片吗？ Does this increase internal fragmentation?
- **链表通常是和分离的空闲列表一起使用的 Most common use of linked lists is in conjunction with segregated free lists**
  - 保持多个不同大小类的列表，或者为不同类型的对象设置不同的列表 Keep multiple linked lists of different size classes, or possibly for different types of objects

议题 Today

- 显示空闲列表 Explicit free lists
- **分离的空闲列表 Segregated free lists**
- 垃圾收集 Garbage collection
- 内存相关的风险和陷阱 Memory-related perils and pitfalls

分离空闲列表（Seglist）分配器
Segregated List (Seglist) Allocators

- 每个不同大小类块有自己的空闲列表 Each *size class* of blocks has its own free list



- 通常比较小的块有自己单独的类 Often have separate classes for each small size
- 对于比较大的块：每个2的指数区间有一个类 For larger sizes: One class for each size $[2^i + 1, 2^{i+1}]$

分离空闲列表分配器 Seglist Allocator

- 空闲列表数组中的每个元素对应某个大小类 **Given an array of free lists, each one for some size class**

- 分配大小为n的块时： **To allocate a block of size *n*:**
  - 搜索对应的空闲列表，其中的块大小m> n Search appropriate free list for block of size *m > n*
  - 如果找到一个合适的块： If an appropriate block is found:
    - 拆分块并将碎片挂接到对应的列表（可选） Split block and place fragment on appropriate list (optional)
  - 如果没找到，则尝试下一个更大的列表 If no block is found, try next larger class
  - 重复以上步骤直到找到一个块 Repeat until block is found

- 如果没找到： **If no block is found:**
  - 从OS申请更多的堆内存（使用sbrk()） Request additional heap memory from OS (using **`sbrk()`**)
  - 从新申请的内存分配大小为n字节的块 Allocate block of *n* bytes from this new memory
  - 将剩下的当做一个空闲块放到最大的类表中 Place remainder as a single free block in largest size class.

Seglist分配器(续) Seglist Allocator (cont.)

- **释放一个块： To free a block:**
  - 合并并放到合适的列表中 Coalesce and place on appropriate list

- **seglist分配器相对非seglist分配器的优点（均采用首次匹配） Advantages of seglist allocators vs. non-seglist allocators (both with first-fit)**
  - 高吞吐率 Higher throughput
    - 对于2的指数次方的大小类是log时间复杂度 log time for power-of-two size classes
  - 更好的内存利用率 Better memory utilization
    - 分离空闲列表中的首次匹配搜索近似于整个堆上的最佳匹配搜索 First-fit search of segregated free list approximates a best-fit search of entire heap.
    - 极端案例：如果每个块有自己的大小类，则等价于最佳匹配 Extreme case: Giving each block its own size class is equivalent to best-fit.

内存分配器的更多资料 More Info on Allocators

- “计算机编程的艺术” **D. Knuth, “*The Art of Computer Programming*”, vol 1, 3rd edition, Addison Wesley, 1997**
  - 关于动态内存分配的经典参考 The classic reference on dynamic storage allocation
- “动态存储分配：调查与评论” **Wilson et al, “*Dynamic Storage Allocation: A Survey and Critical Review*”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.**
  - 综合调查 Comprehensive survey
  - 访问CS:APP学生网站 Available from CS:APP student site (csapp.cs.cmu.edu)

议题 Today

- 显示空闲列表 Explicit free lists
- 分离的空闲列表 Segregated free lists
- **垃圾收集 Garbage collection**
- 内存相关的风险和陷阱 Memory-related perils and pitfalls

隐式内存管理：垃圾收集
Implicit Memory Management: Garbage Collection

- *垃圾收集*：自动回收堆中分配的内存块-应用程序不用负责释放 ***Garbage collection:*** **automatic reclamation of heap-allocated storage—application never has to free**

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- **许多动态语言的共同特性 Common in many dynamic languages:**
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica

- **C和C++存在变种（保守的垃圾收集） Variants ("conservative" garbage collectors) exist for C and C++**
  - 然而，不一定收集所有垃圾 However, cannot necessarily collect all garbage

垃圾收集 Garbage Collection

- 内存管理器如何知道内存什么时候可以被释放？ **How does the memory manager know when memory can be freed?**
  - 通常我们是不知道将来会用到哪些，因为程序执行是有路径分支的 In general we cannot know what is going to be used in the future since it depends on conditionals
  - 但是如果某些块没有指针指向则可以确定是不会用的 But we can tell that certain blocks cannot be used if there are no pointers to them

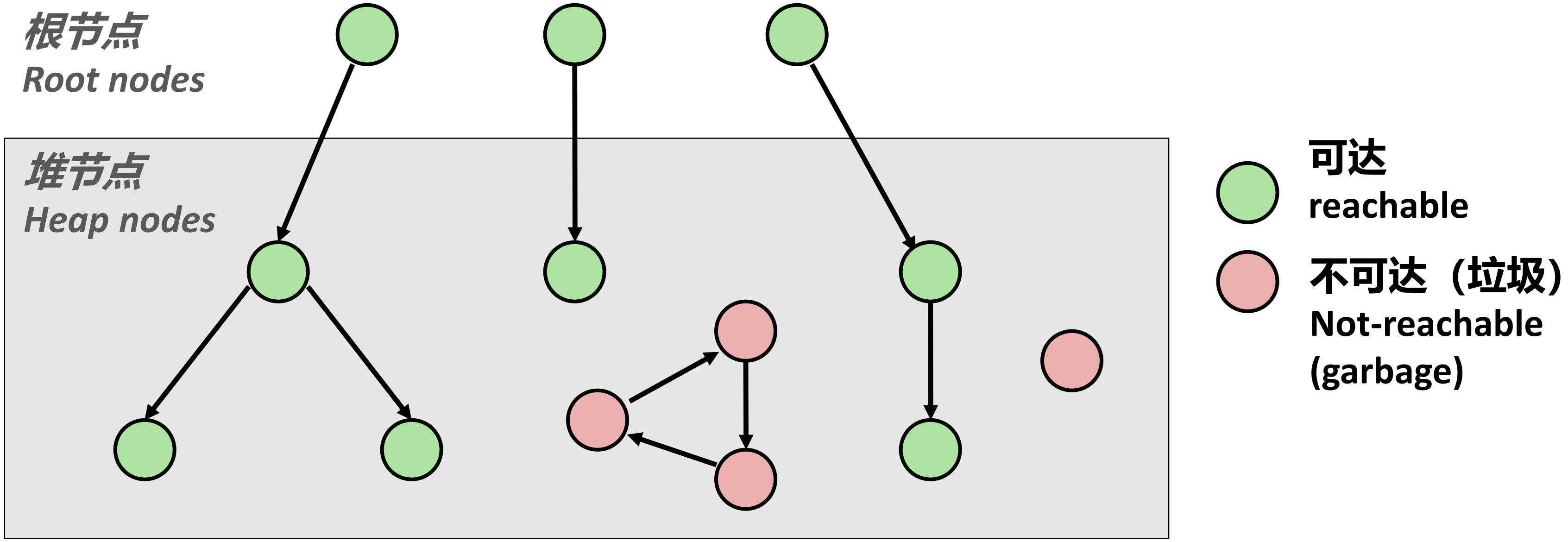
- 关于指针的一些假设 **Must make certain assumptions about pointers**
  - 内存管理器能够区分指针和非指针 Memory manager can distinguish pointers from non-pointers
  - 所有的指针指向块的开始地址 All pointers point to the start of a block
  - 不能隐藏指针 Cannot hide pointers
(例如，强制转为int，再转回来 e.g., by coercing them to an **`int`**, and then back again)

经典垃圾收集算法 Classical GC Algorithms

- 标记清除收集算法 **Mark-and-sweep collection (McCarthy, 1960)**
  - 不需要移动内存块（除非需要平移压紧占用部分） Does not move blocks (unless you also “compact”)
- 引用计数算法 **Reference counting (Collins, 1960)**
  - 不需要移动内存块（不讨论） Does not move blocks (not discussed)
- 拷贝收集算法 **Copying collection（不讨论） (Minsky, 1963)**
  - 需要移动内存块 Moves blocks (not discussed)
- 按代垃圾收集算法 **Generational Collectors (Lieberman and Hewitt, 1983)**
  - 基于生命周期的收集 Collection based on lifetimes
    - 大部分内存块很快变为垃圾 Most allocations become garbage very soon
    - 主要聚焦在最近分配的区域内开展回收工作 So focus reclamation work on zones of memory recently allocated
- 更详细信息参见：“垃圾收集：自动动态内存算法” **For more information:**
  Jones and Lin, “*Garbage Collection: Algorithms for Automatic Dynamic Memory*”, John Wiley & Sons, 1996.

将内存当做一个图 Memory as a Graph

- 我们将内存看做一个有向图 **We view memory as a directed graph**
  - 每个块是图中的一个节点 Each block is a node in the graph
  - 每个指针是图中的一条边 Each pointer is an edge in the graph
  - 不在堆中但是持有指向堆中指针的位置称为根节点（例如，寄存器，栈中元素，以及全局变量） Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)

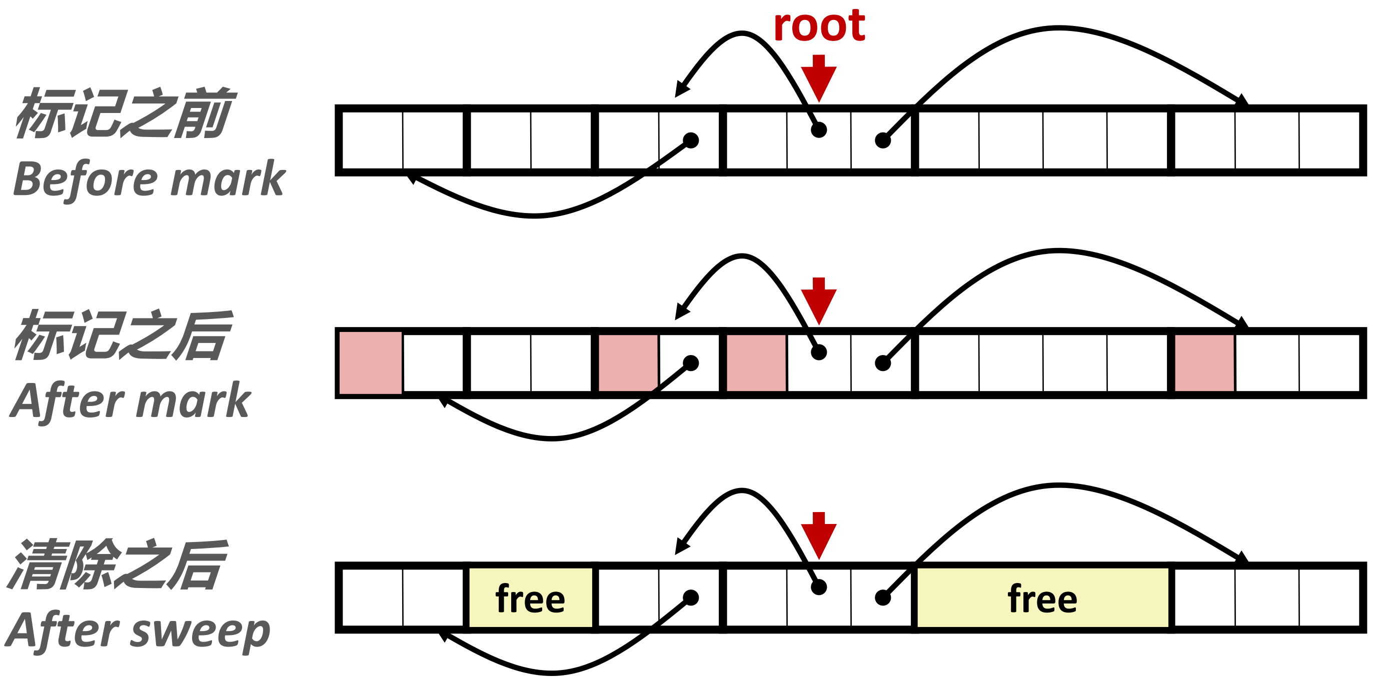


如果有从根节点到某个节点的路径则这个节点是可达的 A node (block) is *reachable* if there is a path from any root to that node.

不可达的都是垃圾（应用程序不再需要） Non-reachable nodes are *garbage* (cannot be needed by the application)

标记清除收集算法 Mark and Sweep Collecting

- **可以基于malloc/free包构建 Can build on top of malloc/free package**
  - 一直使用malloc直到空间不够用 Allocate using `malloc` until you “run out of space”
- **当内存不够用 When out of space:**
  - 在每个块的头部使用额外的标记位 Use extra ***mark bit*** in the head of each block
  - ***Mark:*** 从根节点开始并对所有可达节点设置标记位 Start at roots and set mark bit on each reachable block
  - ***Sweep:*** 扫描所有的块并释放未标记的块 Scan all blocks and free blocks that are not marked

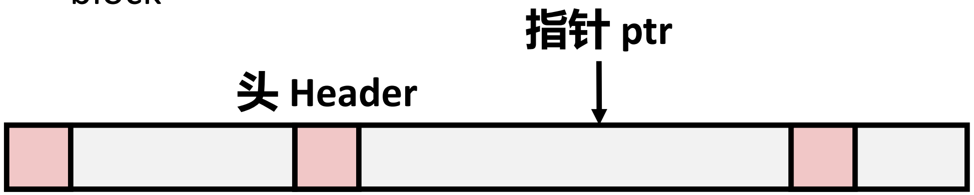


注意：这里的箭头表示引用关系，不是空闲列表指针 Note: arrows here denote memory refs, not free list ptrs.

🟥 **已设置标记位 Mark bit set**

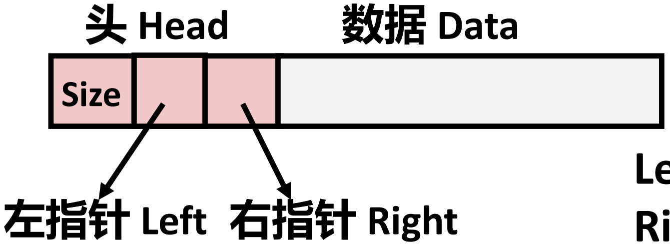
C语言中保守的标记-清除算法
Conservative Mark & Sweep in C

- **C程序的一个保守垃圾收集器 A “conservative garbage collector” for C programs**
  - `is_ptr()` 用来判断一个字是否是指向一个已经分配的内存块的指针 **`is_ptr()`** determines if a word is a pointer by checking if it points to an allocated block of memory
  - 但是，C指针可以指向块中间的位置 But, in C pointers can point to the middle of a block



假设中间的指针可以用于到达块的任何地方，但不能到其他块
Assumes ptr in middle can be used to reach anywhere in the block, but no other block

- **所以要如何找到块的开始？ So how to find the beginning of the block?**
  - 可以使用一个平衡二叉树跟踪所有已经分配的块(key是块开始地址) Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - 平衡二叉树的指针可以存在head中(使用两个额外的字) Balanced-tree pointers can be stored in header (use two additional words)



Left: smaller addresses 较小地址
Right: larger addresses 较大地址

一个简单实现的前提假设
Assumptions For a Simple Implementation

- 应用 **Application**
  - **`new(n)`**：返回指向新块的指针，所有的域清除 returns pointer to new block with all locations cleared
  - **`read(b,i)`**：将块b中位置i的内容读到寄存器 read location **`i`** of block **`b`** into register
  - **`write(b,i,v)`**： 将v写入块b中的位置i write **`v`** into location **`i`** of block **`b`**

- 每个块有一个头部字 **Each block will have a header word**
  - 对b可以使用b[-1]寻址 addressed as **`b[-1]`**, for a block **`b`**
  - 在不同的垃圾收集器里面有不同的用途 Used for different purposes in different collectors
- 垃圾收集器使用的操作 **Instructions used by the Garbage Collector**
  - **`is_ptr(p)`**：确定p是否是一个指针 determines whether **`p`** is a pointer
  - **`length(b)`**: 返回b的长度，不包括头部 returns the length of block **`b`**, not including the header
  - **`get_roots()`**: 返回所有块的根 returns all the roots

标记和清除（续） Mark and Sweep (cont.)

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                    //   in the block
   return;
}
```

清除阶段通过长度找到下一个块 Sweep using lengths to find next

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

标记和清除伪代码

Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
    if (!is_ptr(p)) return;
    if (markBitSet(p)) return;
    setMarkBit(p);
    for (i=0; i < length(p); i++)
      mark(p[i]);
    return;
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;         // if not pointer -> do nothing
   if (markBitSet(p)) return;      // if already marked -> do nothing
   setMarkBit(p);                  // set the mark bit
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;         // if not pointer -> do nothing
   if (markBitSet(p)) return;      // if already marked -> do nothing
   setMarkBit(p);                  // set the mark bit
   for (i=0; i < length(p); i++)   // for each word in p’s block
     mark(p[i]);
   return;
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p’s block
     mark(p[i]);                    //  make recursive call
   return;
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;         // if not pointer -> do nothing
   if (markBitSet(p)) return;      // if already marked -> do nothing
   setMarkBit(p);                  // set the mark bit
   for (i=0; i < length(p); i++)   // for each word in p’s block
     mark(p[i]);                   //  make recursive call
   return;
}
```

清除阶段通过长度找到下一个块 Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {               // for entire heap
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)   // for each word in p’s block
     mark(p[i]);                    //  make recursive call
   return;
}
```

清除阶段通过长度找到下一个块 Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p’s block
     mark(p[i]);                    //  make recursive call
   return;
}
```

清除阶段通过长度找到下一个块 Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)  // for each word in p’s block
     mark(p[i]);                  //  make recursive call
   return;
}
```

清除阶段通过长度找到下一个块 Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {              // for entire heap
      if markBitSet(p)            // did we reach this block?
         clearMarkBit();          //  yes -> so just clear mark bit
      else if (allocateBitSet(p)) // never reached: is it allocated?
         free(p);
      p += length(p+1);
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p’s block
     mark(p[i]);                    //  make recursive call
   return;
}
```

清除阶段通过长度找到下一个块 Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p))   // never reached: is it allocated?
         free(p);                   //  yes -> its garbage, free it
      p += length(p+1);
}
```

标记和清除伪代码
Mark and Sweep Pseudocode

通过内存图的深度优先遍历标记 Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)  // for each word in p’s block
     mark(p[i]);                  //  make recursive call
   return;
}
```

清除阶段通过长度找到下一个块 Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {              // for entire heap
      if markBitSet(p)            // did we reach this block?
         clearMarkBit();          //  yes -> so just clear mark bit
      else if (allocateBitSet(p)) // never reached: is it allocated?
         free(p);                 //  yes -> its garbage, free it
      p += length(p+1);           // goto next block
}
```

C指针声明：测试一下你自己
C Pointer Declarations: Test Yourself!

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Source: K&R Sec 5.12

C指针声明：测试一下你自己
C Pointer Declarations: Test Yourself!

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int

Source: K&R Sec 5.12

分析：Parsing: `int (*(*f())[13])()`

`int (*(*f())[13])()`	f
`int (*(*f())[13])()`	f is a function
`int (*(*f())[13])()`	f is a function that returns a ptr
`int (*(*f())[13])()`	f is a function that returns a ptr to an array of 13
`int (*(*f())[13])()`	f is a function that returns a ptr to an array of 13 ptrs
`int (*(*f())[13])()`	f is a function that returns a ptr to an array of 13 ptrs to functions returning an int

议题 Today

- 显示空闲列表 Explicit free lists
- 分离的空闲列表 Segregated free lists
- 垃圾收集 Garbage collection
- **内存相关的风险和陷阱 Memory-related perils and pitfalls**

内存相关的风险和陷阱
Memory-Related Perils and Pitfalls

- 解引（间接引用）问题指针 **Dereferencing bad pointers**
- 使用未初始化内存 **Reading uninitialized memory**
- 覆盖内存 **Overwriting memory**
- 引用不存在的变量 **Referencing nonexistent variables**
- 重复释放内存块 **Freeing blocks multiple times**
- 引用释放的内存 **Referencing freed blocks**
- 释放内存失败 **Failing to free blocks**

解引（间接引用）问题指针
Dereferencing Bad Pointers

- 经典的scanf bug The classic `scanf` bug

```
int val;

...

scanf(“%d”, val);
```

使用未初始化变量 Reading Uninitialized Memory

- 假设堆数据初始化为0 Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

- 使用calloc可以避免 Can avoid by using `calloc`

覆盖内存 Overwriting Memory

- 分配了可能错误大小的对象 **Allocating the (possibly) wrong sized object**

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

- 你能发现这个**bug**吗？ **Can you spot the bug?**

覆盖内存 Overwriting Memory

- 错位错误 **Off-by-one error**

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

```
char *p;

p = malloc(strlen(s));
strcpy(p,s);
```

覆盖内存 Overwriting Memory

- 没有检查最大字符串长度 **Not checking the max string size**

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- 经典缓冲区溢出攻击的基础 **Basis for classic buffer overflow attacks**

覆盖内存 Overwriting Memory

- 指针运算理解错误 Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

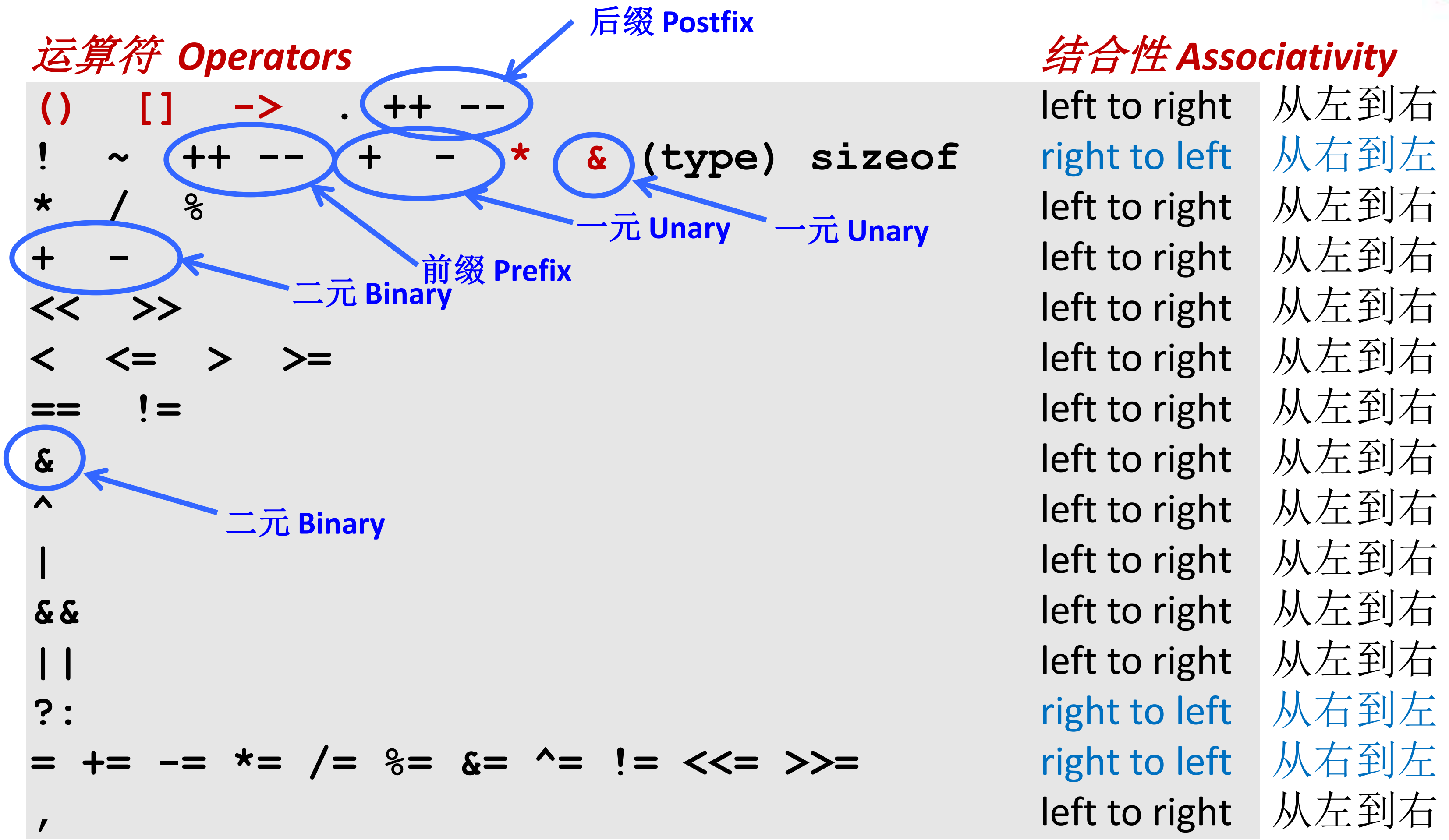
覆盖内存 Overwriting Memory

- 引用了一个指针，而不是其指向的对象 **Referencing a pointer instead of the object it points to**

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

- 减的是什么？ **What gets decremented?**
  - (见下页幻灯片) / (See next slide)

C语言运算符 C operators



<table>
<tr><th>运算符 Operators</th><th>结合性 Associativity</th><th></th></tr>
<tr><td>() [] -> . ++ --</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>! ~ ++ -- + - * & (type) sizeof</td><td>right to left</td><td>从右到左</td></tr>
<tr><td>* / %</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>+ -</td><td>left to right</td><td>从左到右</td></tr>
<tr><td><< >></td><td>left to right</td><td>从左到右</td></tr>
<tr><td>< <= > >=</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>== !=</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>&</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>^</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>|</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>&&</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>||</td><td>left to right</td><td>从左到右</td></tr>
<tr><td>?:</td><td>right to left</td><td>从右到左</td></tr>
<tr><td>= += -= *= /= %= &= ^= != <<= >>=</td><td>right to left</td><td>从右到左</td></tr>
<tr><td>,</td><td>left to right</td><td>从左到右</td></tr>
</table>

- **->, ( ) , and [ ] have high precedence ->、()和[]有最高优先级, with * and & just below *、&有次高优先级**
- **一元+、-和*比二元形式有更高优先级 Unary +, –, and * have higher precedence than binary forms**

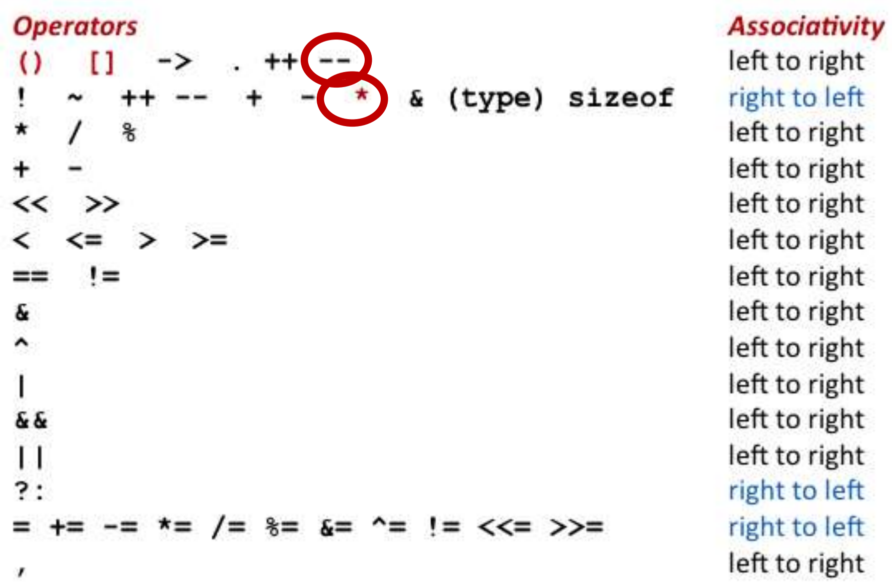
来源：Source: K&R page 53, updated

覆盖内存 Overwriting Memory

- 引用了一个指针，而不是其指向的对象 **Referencing a pointer instead of the object it points to**

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

- 下面效果相同 **Same effect as**
  - `size--;`
- 应重写为 **Rewrite as**
  - `(*size)--;`



Operators	*Associativity*
() [] -> . ++ --	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= != <<= >>=	right to left
,	left to right

引用不存在的变量 Referencing Nonexistent Variables

- 忘记函数返回之后局部变量不可用 **Forgetting that local variables disappear when a function returns**

```
int *foo () {
   int val;

   return &val;
}
```

多次重复释放块 Freeing Blocks Multiple Times

- 很危险！ **Nasty!**

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

引用已经释放的块 Referencing Freed Blocks

- 令人讨厌！ **Evil!**

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

没有释放内存块（内存泄漏）
Failing to Free Blocks (Memory Leaks)

- 慢性长期的问题 **Slow, long-term killer!**

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

没有释放内存块（内存泄漏）
Failing to Free Blocks (Memory Leaks)

- 只是释放了数据结构的一部分 **Freeing only part of a data structure**

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
     ...
   free(head);
   return;
}
```

应对内存Bug Dealing With Memory Bugs

- **调试器：gdb   Debugger: `gdb`**
  - 能够方便找出问题指针解引 Good for finding bad pointer dereferences
  - 难以探测其他内存问题 Hard to detect the other memory bugs
- **数据结构一致性检查 Data structure consistency checker**
  - 静默运行，出错时打印信息 Runs silently, prints message only on error
  - 用作错误归零的探针 Use as a probe to zero in on error
- **二进制翻译：valgrind   Binary translator: `valgrind`**
  - 强大的调试和分析技术 Powerful debugging and analysis technique
  - 重写可执行目标文件的代码段 Rewrites text section of executable object file
  - 运行时检查每个单独的引用 Checks each individual reference at runtime
    - 问题指针、覆盖、越界访问 Bad pointers, overwrites, refs outside of allocated block
- **glibc malloc 包含了检查代码 glibc malloc contains checking code**
  - `setenv MALLOC_CHECK_ 3`