内存管理

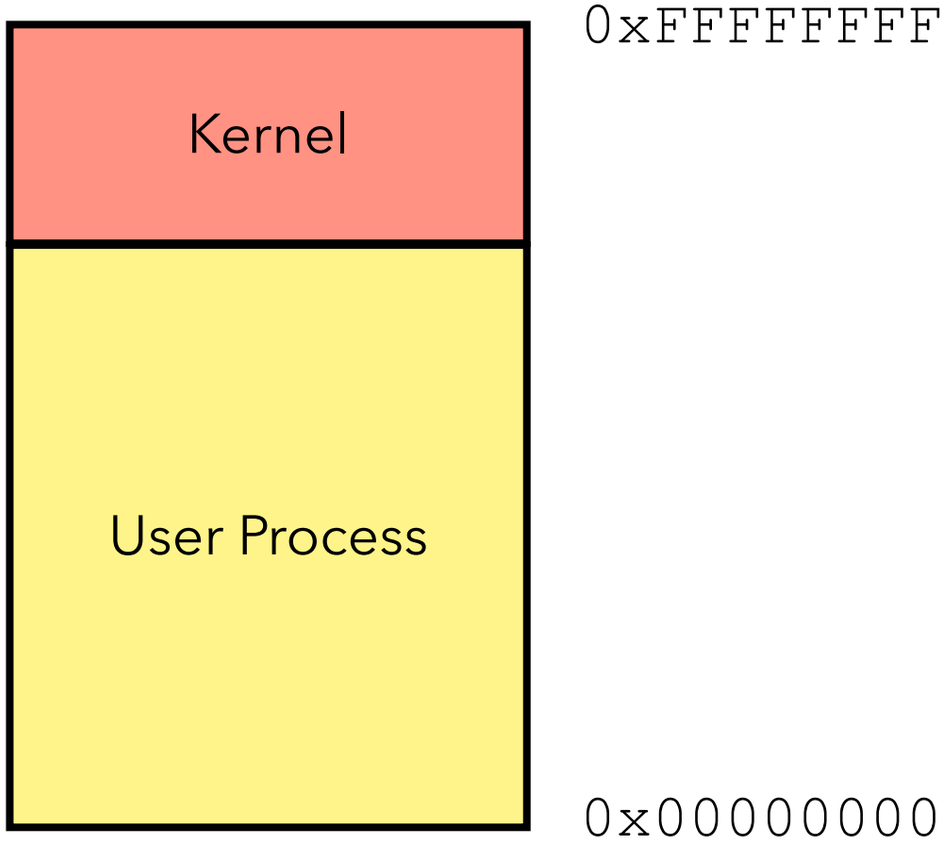
Section 3

2026

内存管理

应用程序需要加载到内存 (一个连续的字节数组) 中来运行

- 在早期的系统中，应用程序和操作系统各自使用物理内存的一部分
  - 通常一次仅运行一个程序，程序总是加载到物理内存的固定位置
  - 应用程序直接使用物理内存地址进行访问



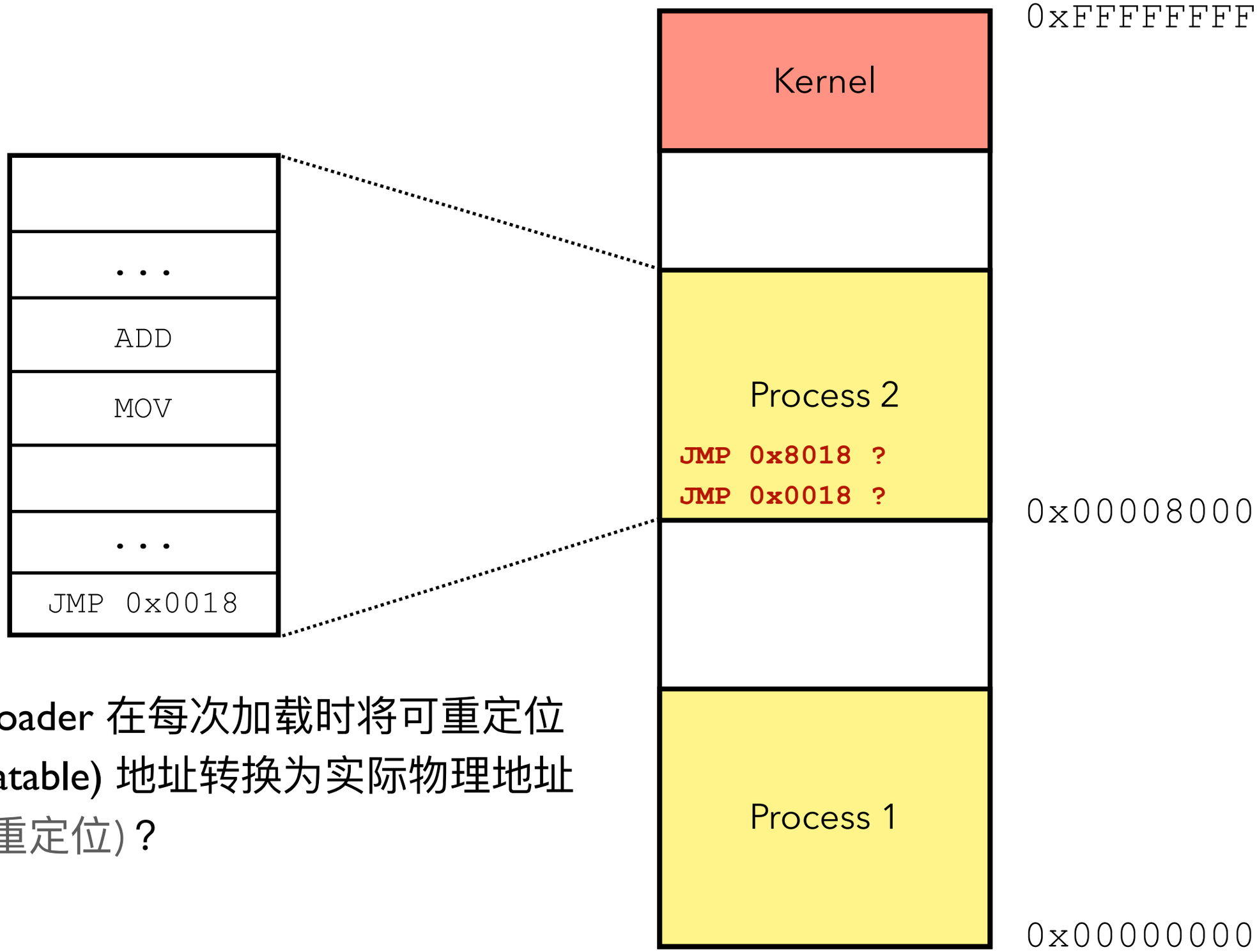
内存管理

应用程序需要加载到内存 (一个连续的字节数组) 中来运行

- 现代操作系统需要加载多个进程到物理内存中，并在其中进行切换
  - 多个进程必须以某种方式共享物理内存
  - 这也就带来了重定位 (relocation) 和保护 (protection) 的问题

内存管理

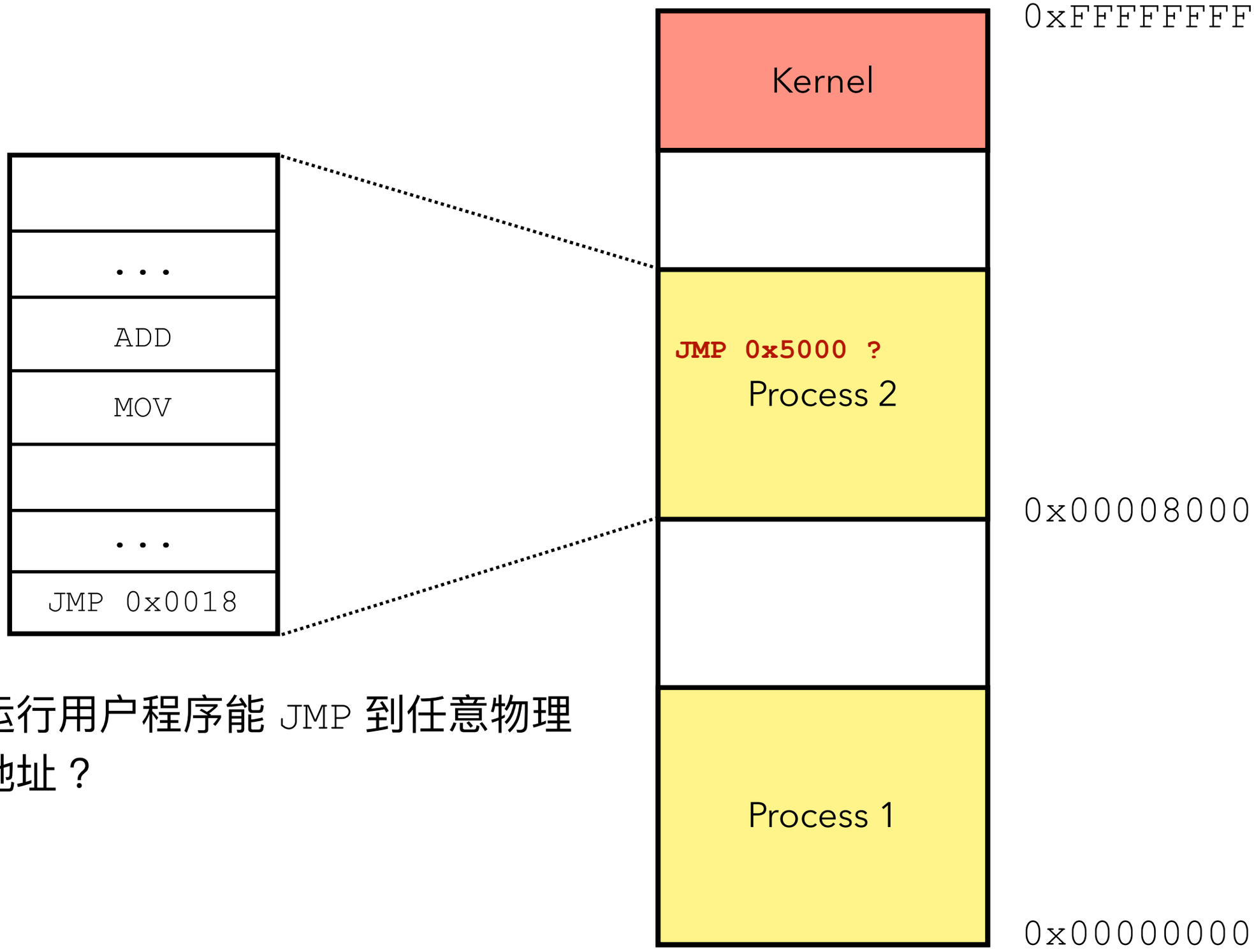
Relocation



依靠 loader 在每次加载时将可重定位 (relocatable) 地址转换为实际物理地址 (静态重定位)？

内存管理

Protection



如果运行用户程序能 JMP 到任意物理内存地址？

内存管理

作为应用程序和硬件之间的中间层，操作系统需要

- 创建某种抽象，使程序员无需关心物理内存的具体细节
- 对物理内存进行管理，以最大化物理内存利用效率
  - 追踪物理内存的使用情况 (哪些部分已分配/空闲)
  - 当进程需要时分配新的内存空间，并在进程不需要时回收
  - 实现不同进程间的隔离和保护
  - 克服物理内存的容量局限

地址空间

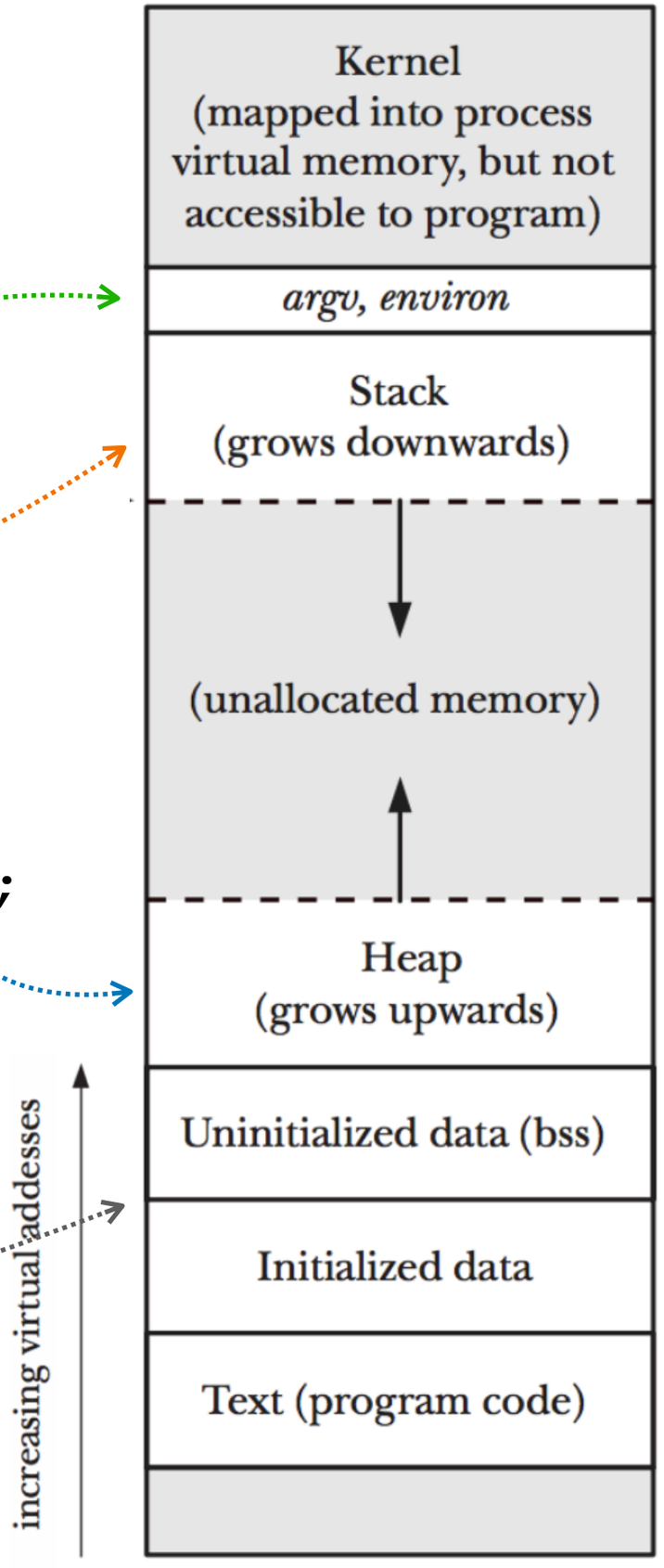
为每个进程提供一个独立于其它进程的地址空间 (Address Space)

- 物理内存的虚拟化
- 进程视角 ➡️ 虚拟地址 (Virtual/Logical Address)
  - 一段大容量、连续且私有的地址空间
  - 无论当前物理内存大小如何、以及当前物理内存被如何使用
- 硬件视角 ➡️ 物理地址 (Physical Address)
  - 一段有限的、多个进程共享的地址空间

地址空间

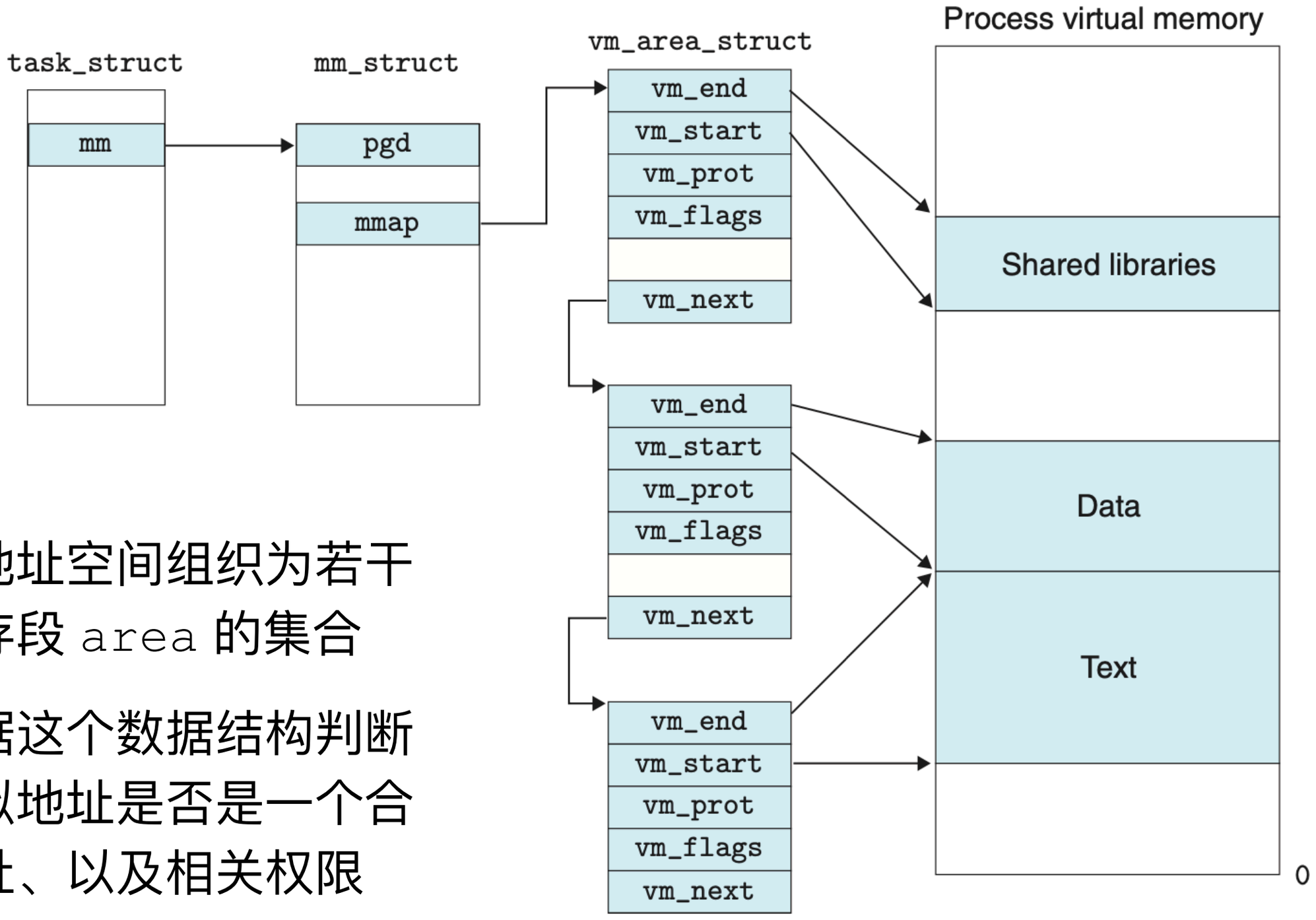
Address Space in Linux

```
int x;
int y = 15;

int main(int argc, char *argv[])
{
  int *value;
  int i;

  value = (int *)malloc(sizeof(int)*5);
  for (i = 0; i < 5; i++)
    value[i] = 10;

  return 0;
}
```



地址空间

mm_struct



- 将进程地址空间组织为若干连续内存段 area 的集合
- 可以根据这个数据结构判断一个虚拟地址是否是一个合法的地址、以及相关权限

地址空间

`mm_struct`

基于进程虚拟地址空间，操作系统为内存管理提供相关的系统调用

- `brk()`: 改变程序间断点 (program break) 的位置
- `mmap()`: 将一个文件或设备映射到进程地址空间中 (通过匿名映射来分配额外内存空间)
- `munmap()`: 撤销地址空间的映射
- `mprotect()`: 修改映射权限

地址空间

Address Space in Linux

可以通过 `pmap` 工具来查看进程的虚拟地址空间

- 观察不同变量和函数的地址
  - `printf()` 位于 `stack` 和 `heap` 中间的区域
  - `malloc()` 并不是要多少空间就分配多少
- 还可以发现内核映射的只读区域 `vdso` (virtual dynamic shared object)，用于在用户态直接调用系统调用

```
00005620c65e6000      4K r---- a.out
00005620c65e7000      4K r-x-- a.out
00005620c65e8000      4K r---- a.out
00005620c65e9000      4K r---- a.out
00005620c65ea000      4K rw--- a.out
00005620c7049000    132K rw---   [ anon ]
00007f2915f9f000     12K rw---   [ anon ]
00007f2915fa2000    160K r---- libc.so.6
00007f2915fca000   1568K r-x-- libc.so.6
00007f2916152000    316K r---- libc.so.6
00007f29161a1000     16K r---- libc.so.6
00007f29161a5000      8K rw--- libc.so.6
00007f29161a7000     52K rw---   [ anon ]
00007f29161bb000      8K rw---   [ anon ]
00007f29161bd000      4K r---- ld-linux-x86-64.so.2
00007f29161be000    172K r-x-- ld-linux-x86-64.so.2
00007f29161e9000     40K r---- ld-linux-x86-64.so.2
00007f29161f3000      8K r---- ld-linux-x86-64.so.2
00007f29161f5000      8K rw--- ld-linux-x86-64.so.2
00007ffc439f5000    132K rw---   [ stack ]
00007ffc43ab8000     16K r----   [ anon ]
00007ffc43abc000      4K r-x--   [ anon ]
```

地址空间

Address Space in Linux

`/proc` 中提供了用于访问进程虚拟地址空间的两个接口

- `/proc/[PID]/maps`: 获取当前映射的内存区域及其访问权限
- `/proc/[PID]/mem`: 进程的虚拟地址空间
  - 虚拟地址空间和文件都可以看作 an array of bytes

地址空间

Address Space in Linux

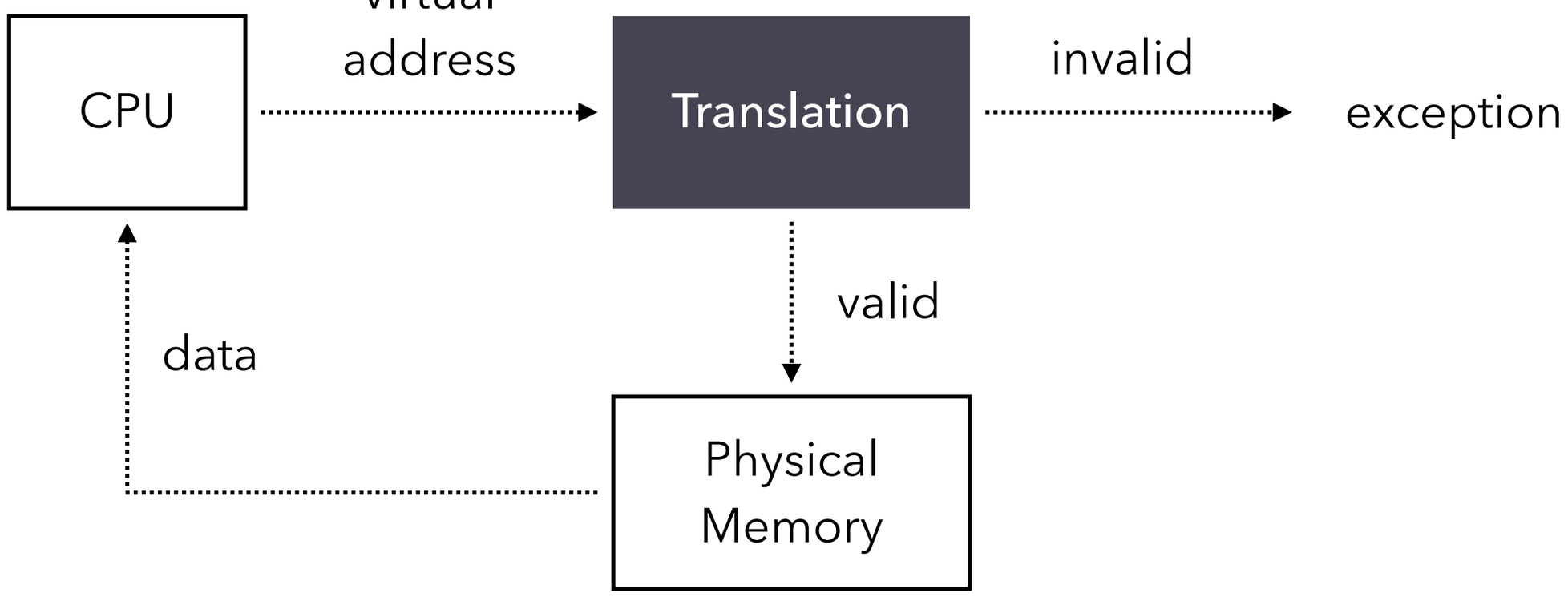
利用 /proc 提供的接口，我们可以尝试写一个程序去修改其它进程的地址空间

```
// 例如，修改 heap 上的一个字符串
int main() {
  printf("pid: %d\n", getpid());
  char *s = strdup("hello");

  while(1) {
    printf("%s: %p\n", s, s);
    sleep(2);
  }
  return 0;
}
```

地址转换

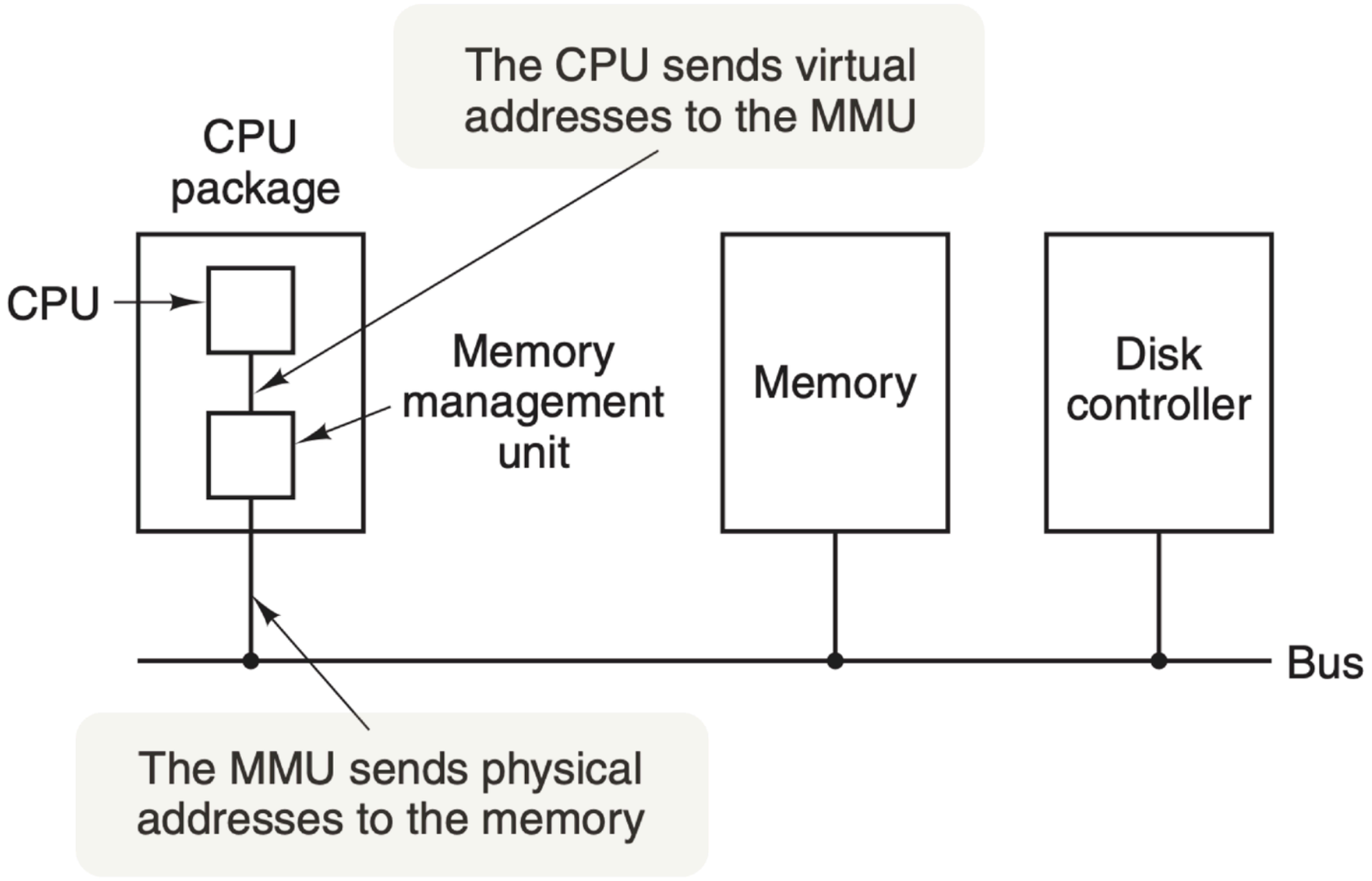
连接虚拟地址和物理地址: $f(pid, virtual\ address) \rightarrow physical\ address$

- 进程只能看到虚拟地址，无法感知物理地址
- 为了实现地址空间抽象，一个必要的机制就是地址转换



地址转换

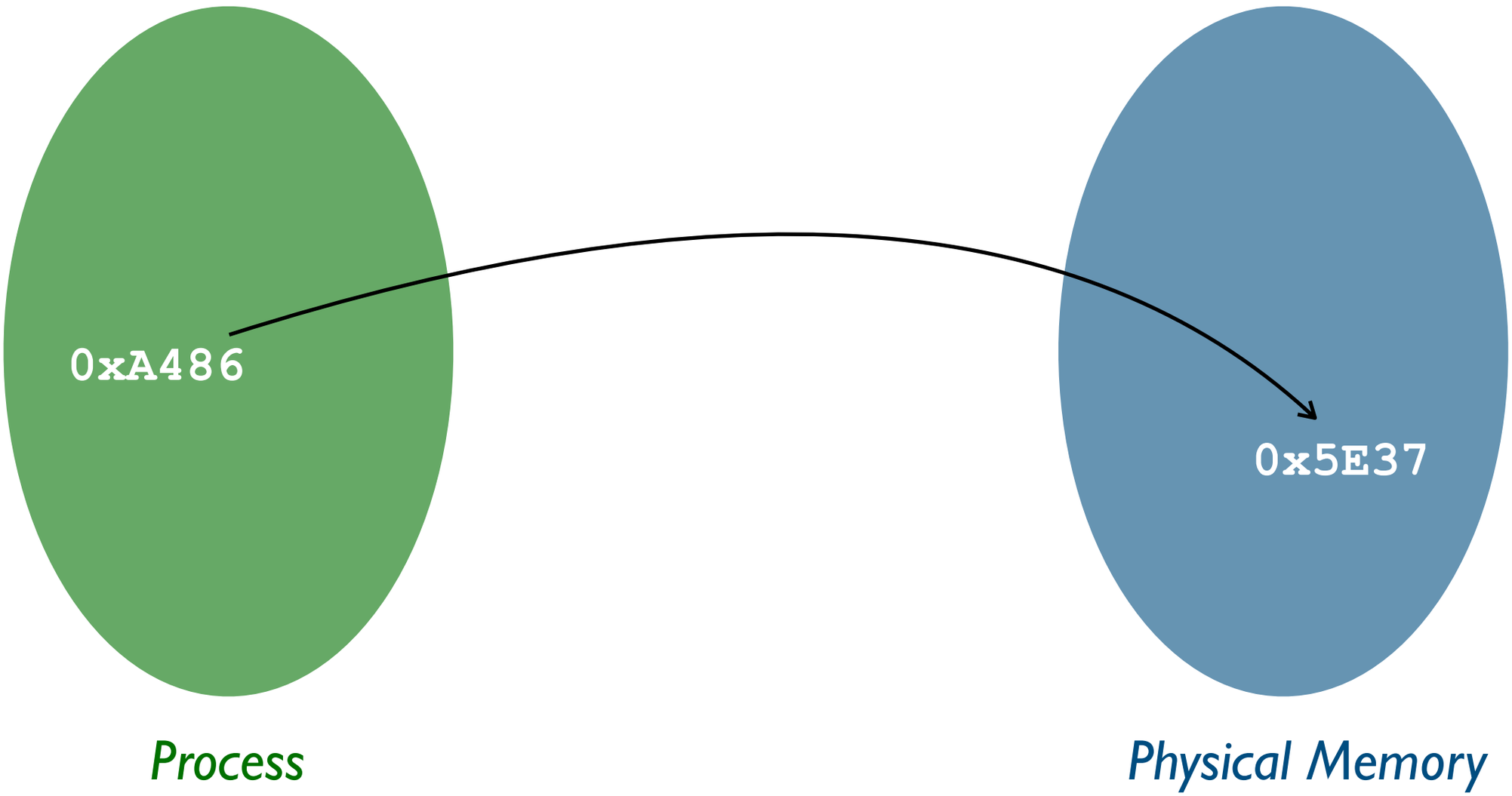
为了提高地址转换效率，通常需要硬件 (Memory Management Unit, MMU) 提供支持



地址转换

Mapping

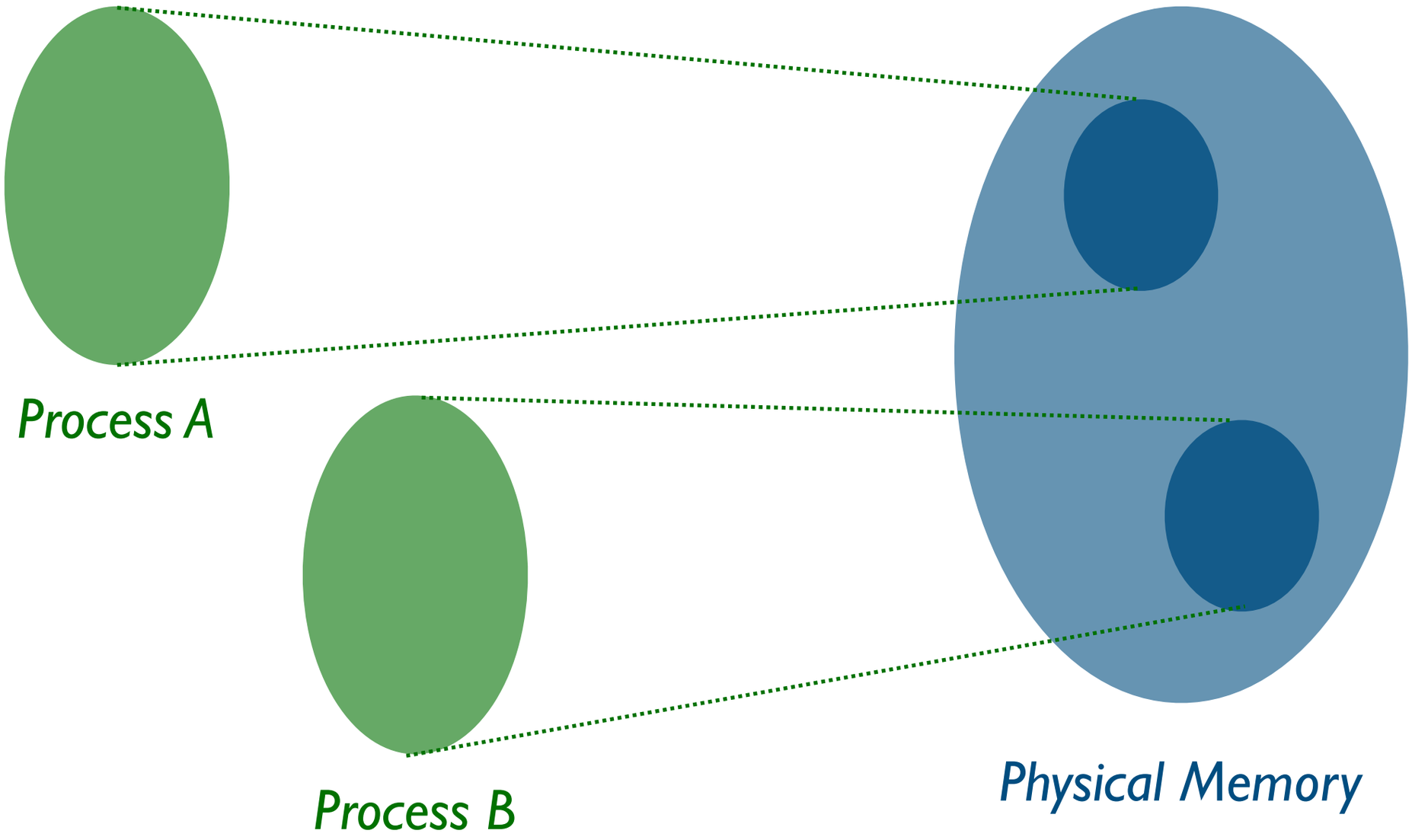
进程的地址转换函数：将每个虚拟地址映射到一个具体的物理地址



地址转换

Protection & Isolation

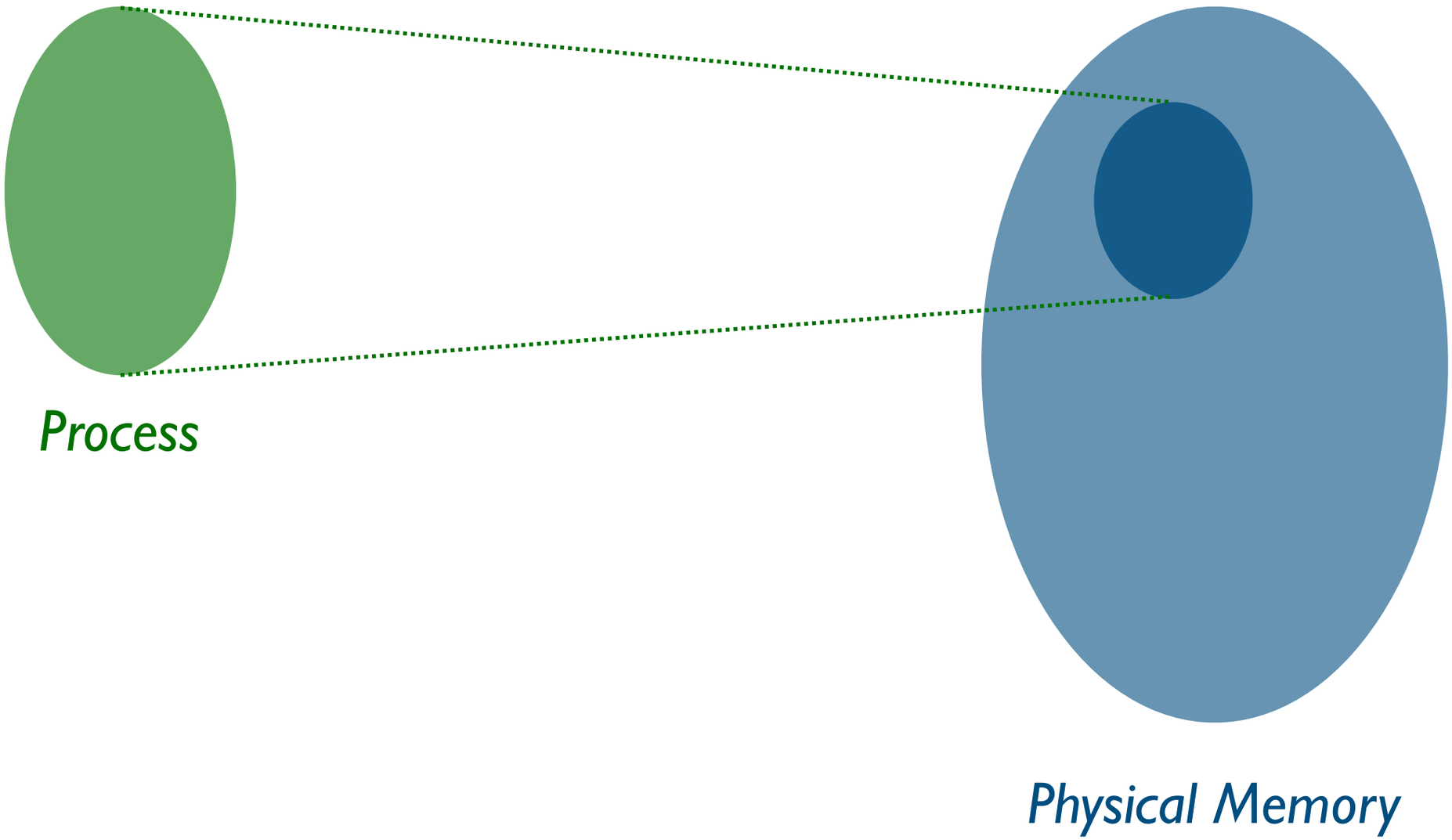
不同进程使用不同的地址转换函数 (值域不相交)：将不同进程的虚拟地址空间映射到互不重叠的物理内存区域 (进程间/进程和 OS 间的隔离)



地址转换

Relocation

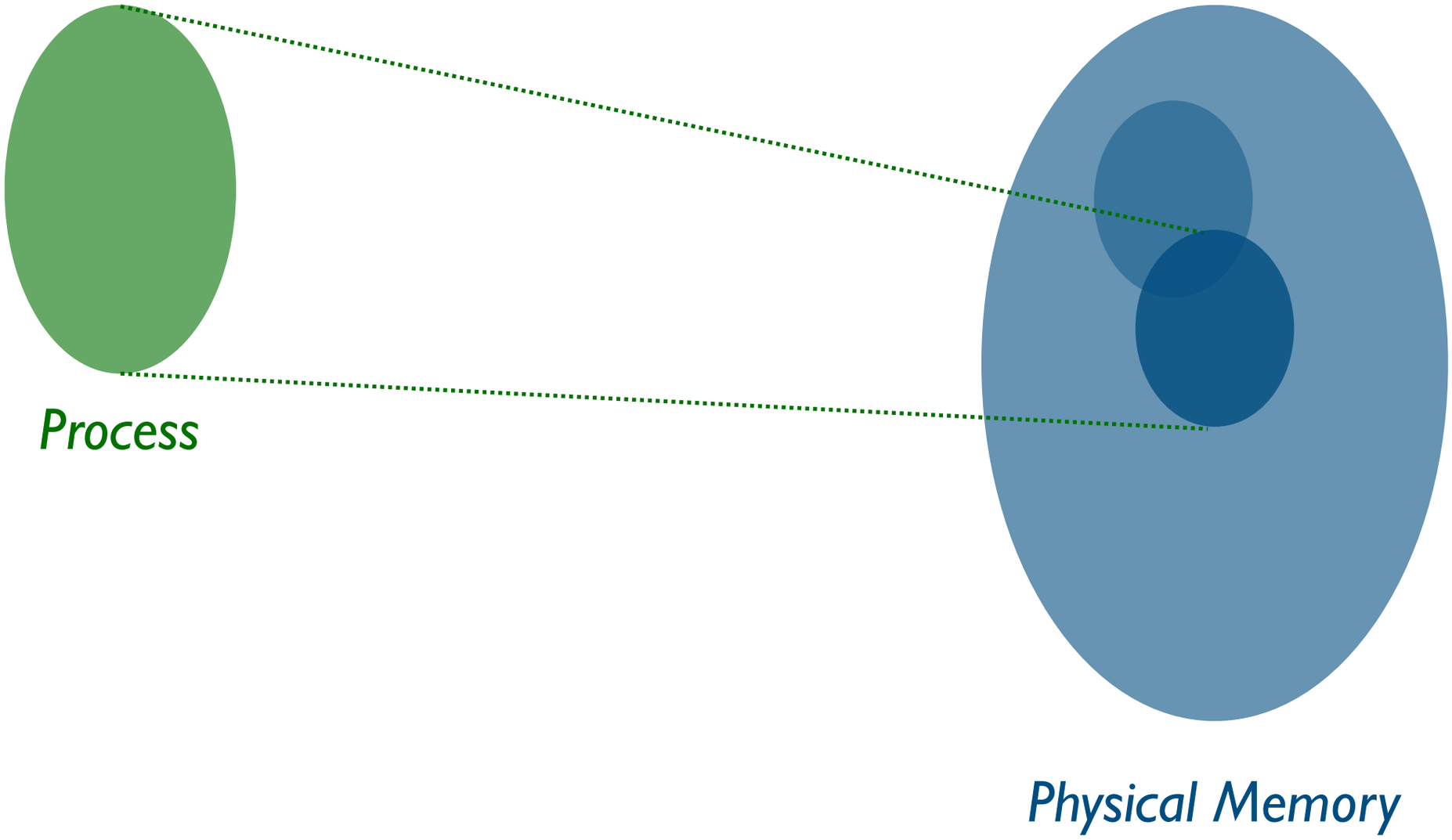
进程使用的地址转换函数可在运行时变化： 允许将进程加载到物理内存的不同地址 (动态重定位)



地址转换

Relocation

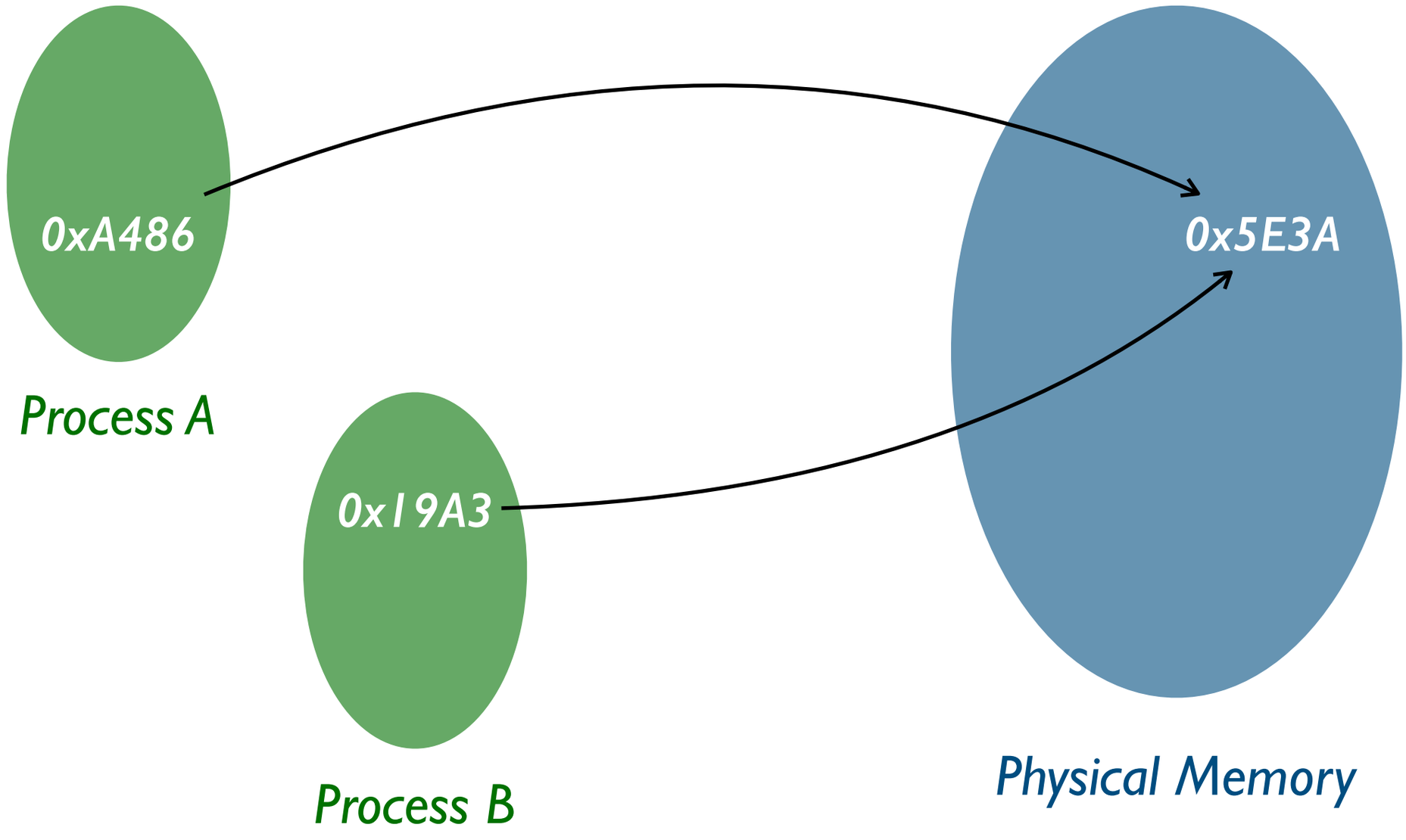
进程使用的地址转换函数可在运行时变化： 允许将进程加载到物理内存的不同地址 (动态重定位)



地址转换

Data Sharing

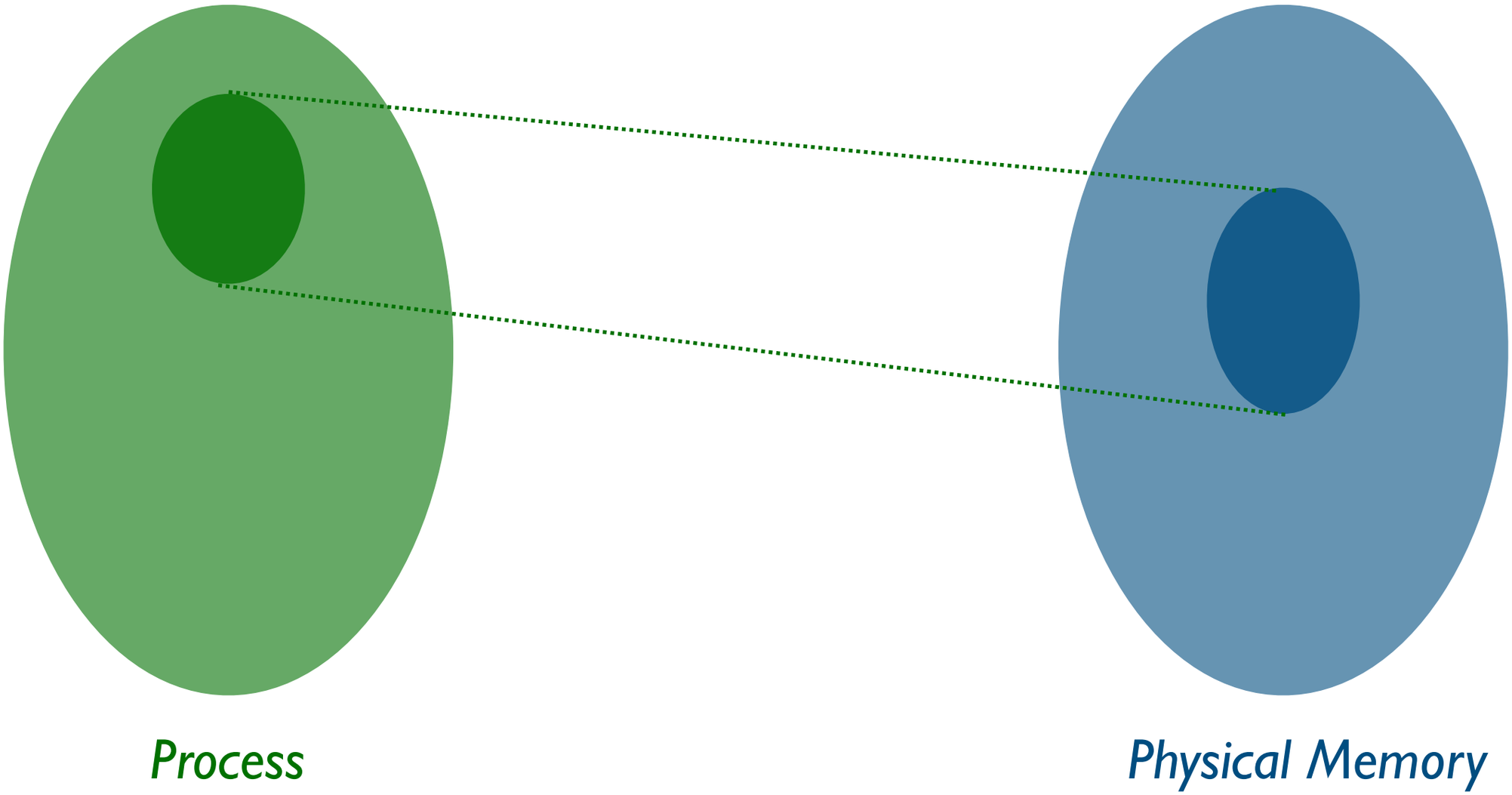
将不同进程的虚拟地址映射到同一个物理地址：进程间共享代码/数据



地址转换

Multiplexing

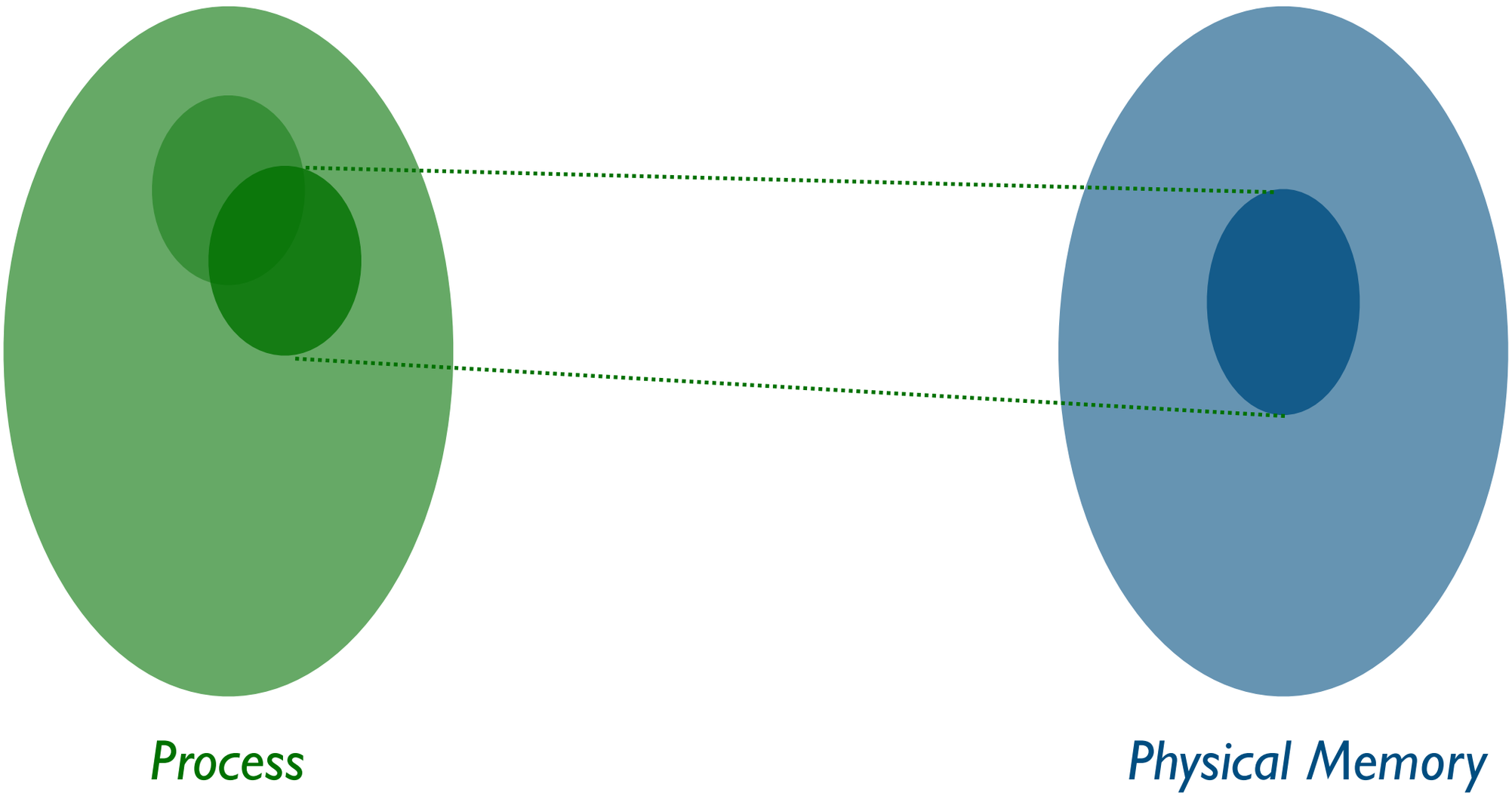
将进程虚拟地址空间的不同部分 (随时间变化) 映射到同一个物理地址区域：创建无限可用内存空间的假象



地址转换

Multiplexing

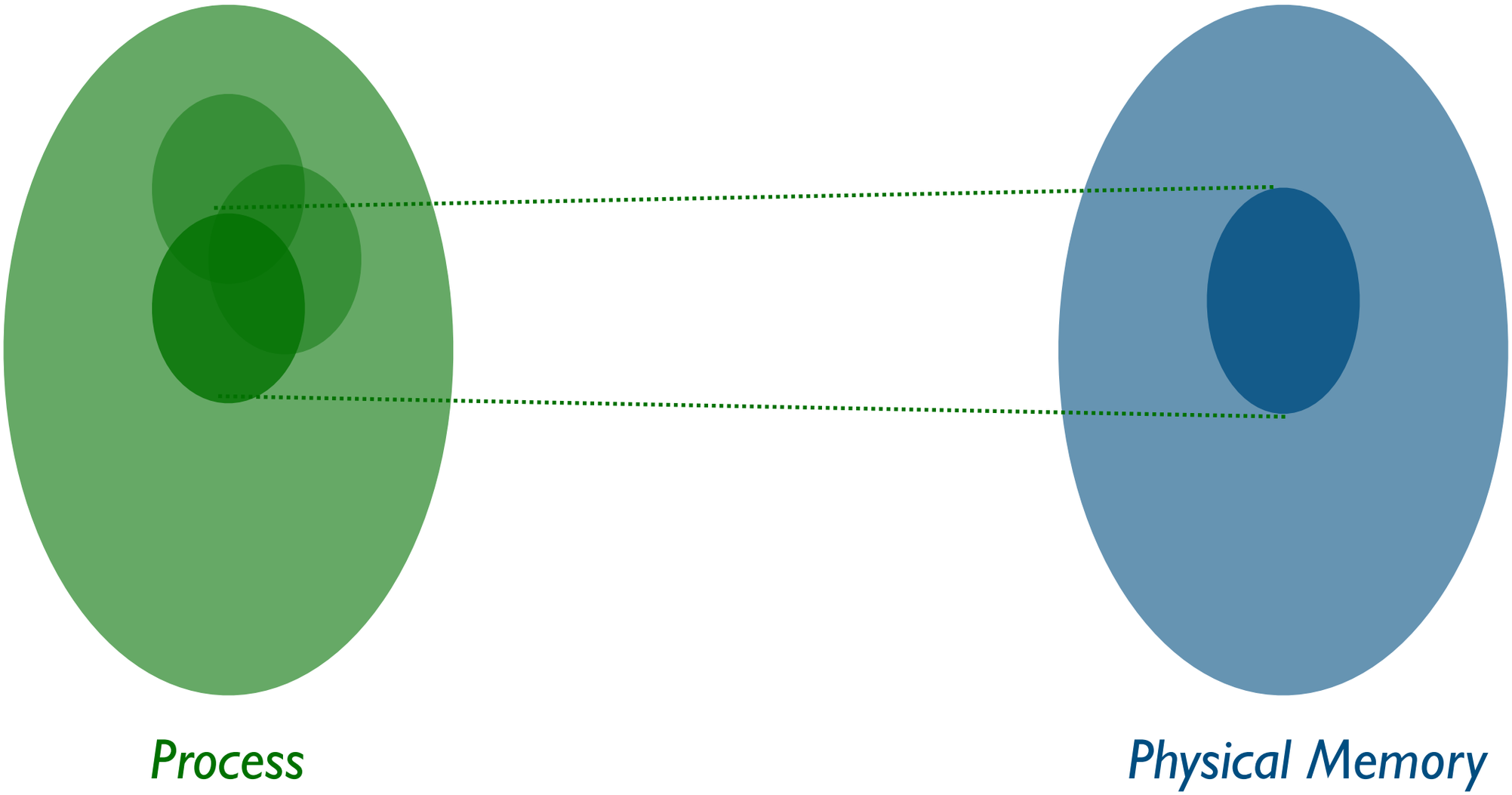
将进程虚拟地址空间的不同部分 (随时间变化) 映射到同一个物理地址区域：创建无限可用内存空间的假象



地址转换

Multiplexing

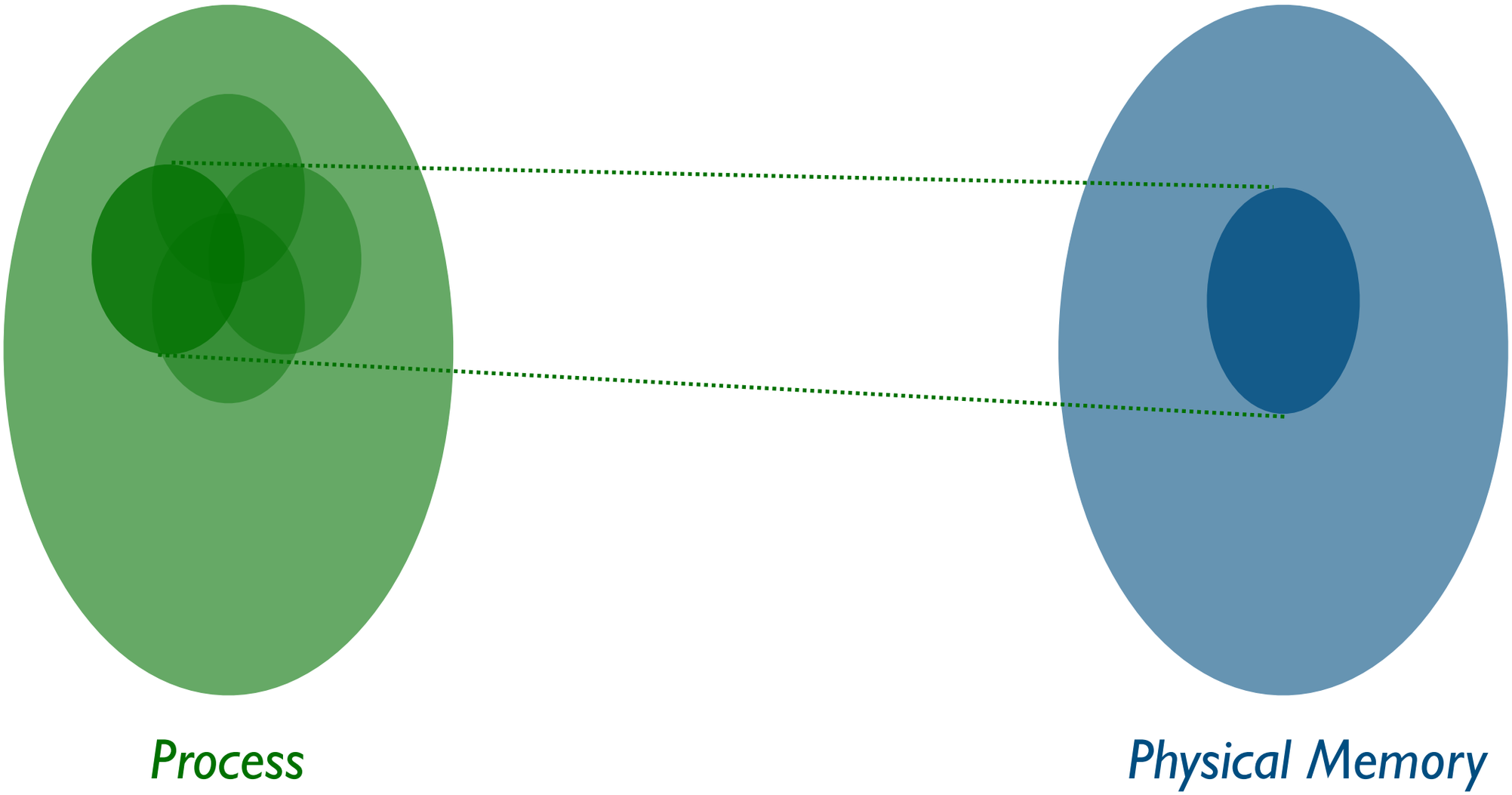
将进程虚拟地址空间的不同部分 (随时间变化) 映射到同一个物理地址区域：创建无限可用内存空间的假象



地址转换

Multiplexing

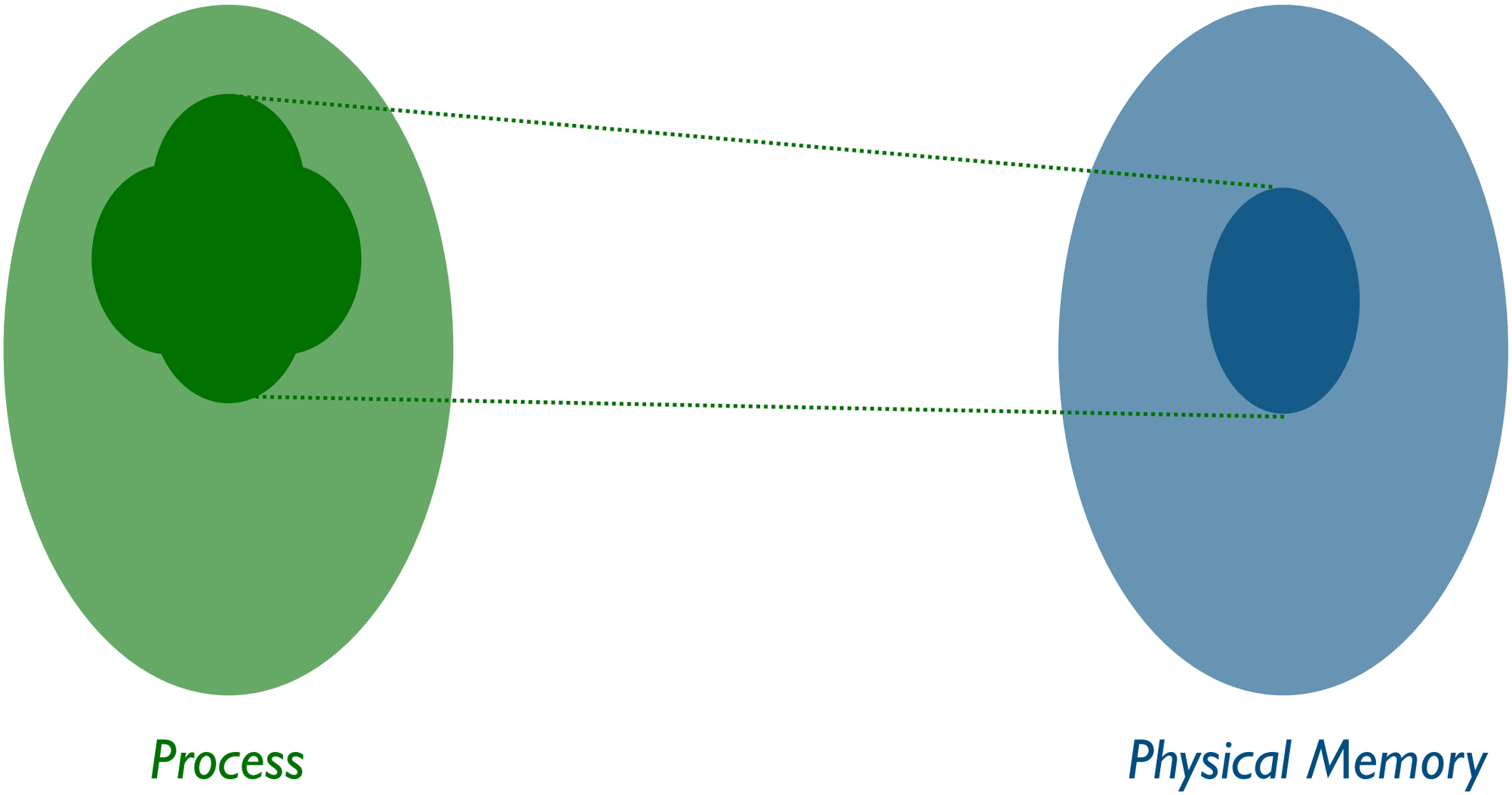
将进程虚拟地址空间的不同部分 (随时间变化) 映射到同一个物理地址区域：创建无限可用内存空间的假象



地址转换

Multiplexing

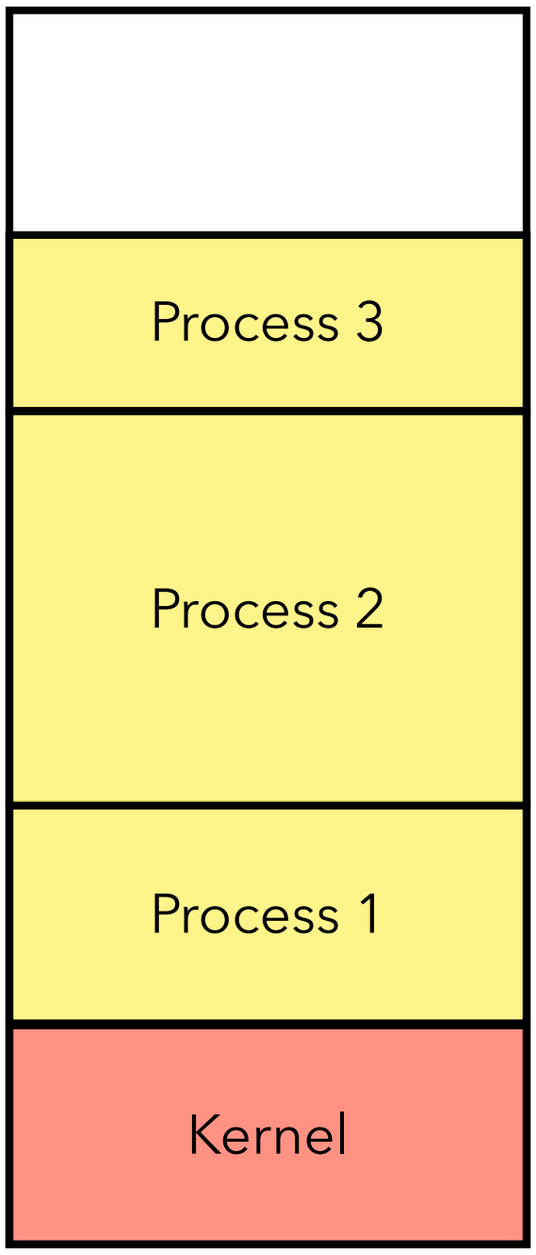
将进程虚拟地址空间的不同部分 (随时间变化) 映射到同一个物理地址区域：创建无限可用内存空间的假象



连续内存空间管理

连续内存空间管理

将连续的虚拟地址空间映射到连续的物理地址空间

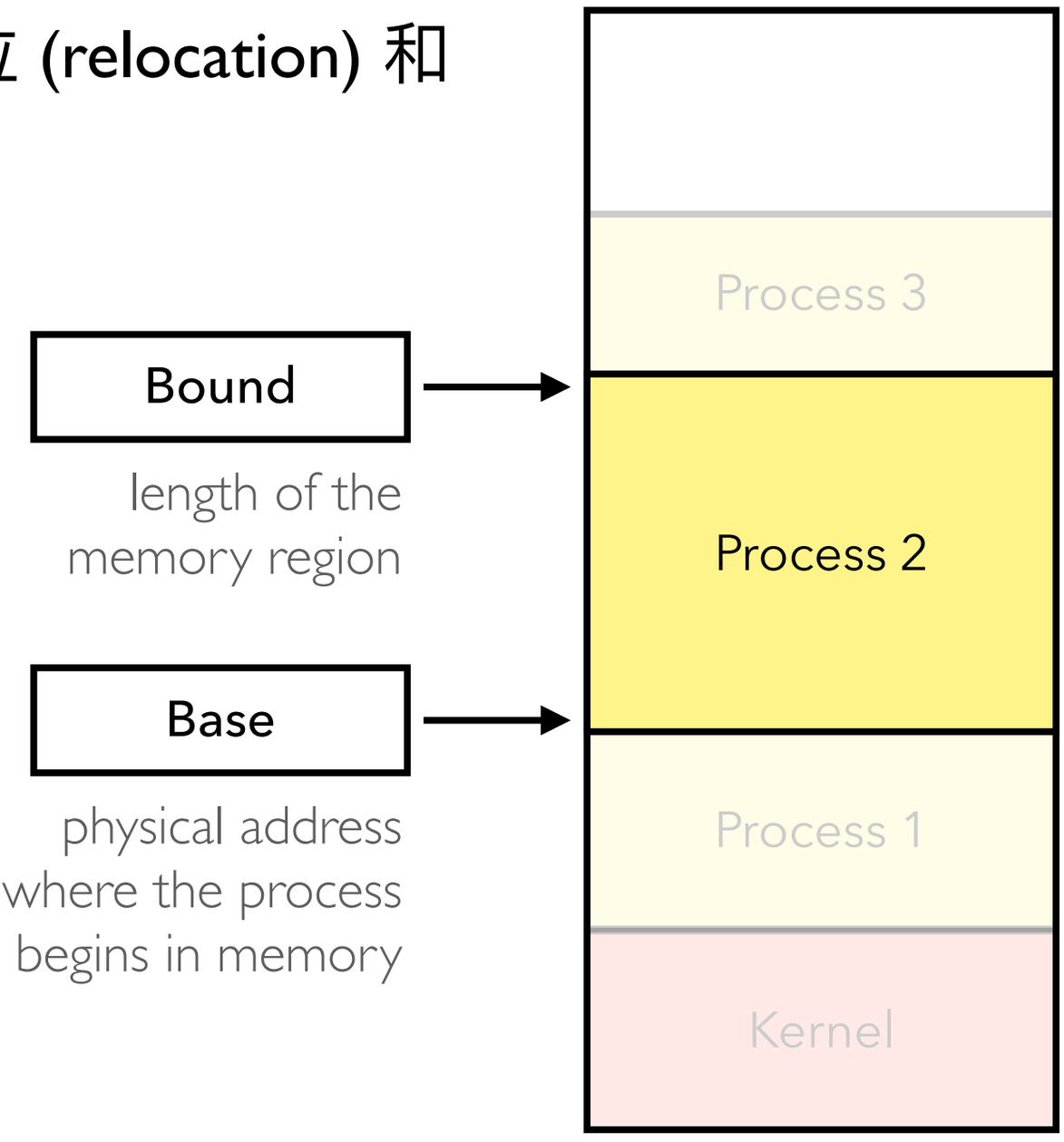


连续内存空间管理

Address Translation

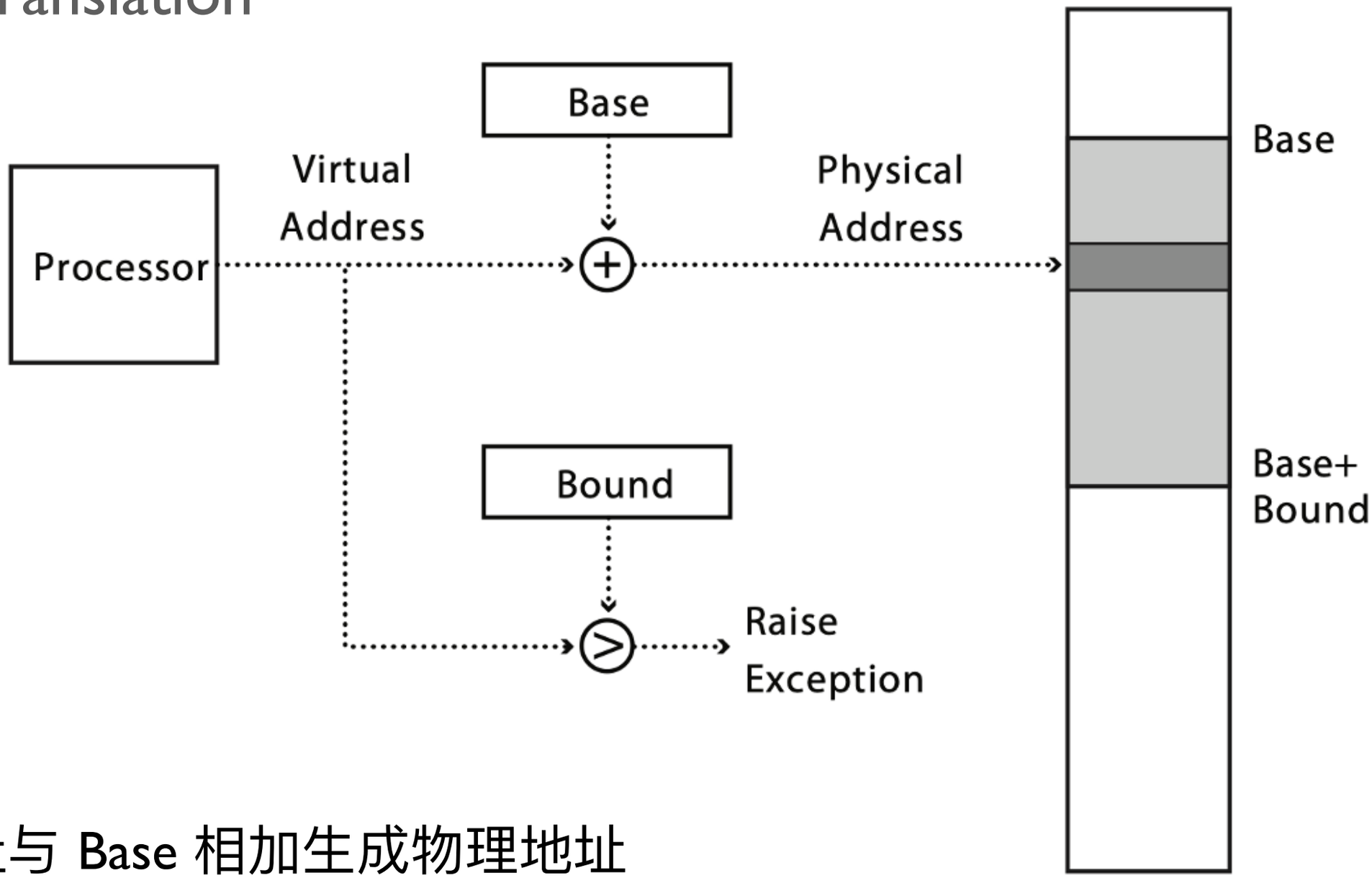
利用 Base and Bound (Limit) 两个特殊寄存器 (MMU) 来定义地址空间

- 非常简单，但已经提供了重定位 (relocation) 和保护 (protection) 的能力
- 对 Base 和 Bound 的修改应属于特权操作
- 在进程上下文切换时，需要切换这两个寄存器的值



连续内存空间管理

Address Translation



- 虚拟地址与 Base 相加生成物理地址
- 虚拟地址和 Bound 进行比较以判断是否超出了内存地址访问界限
  (Bound 也可以用于存储物理地址的界限，取决于地址转换电路的设计)

内存分配

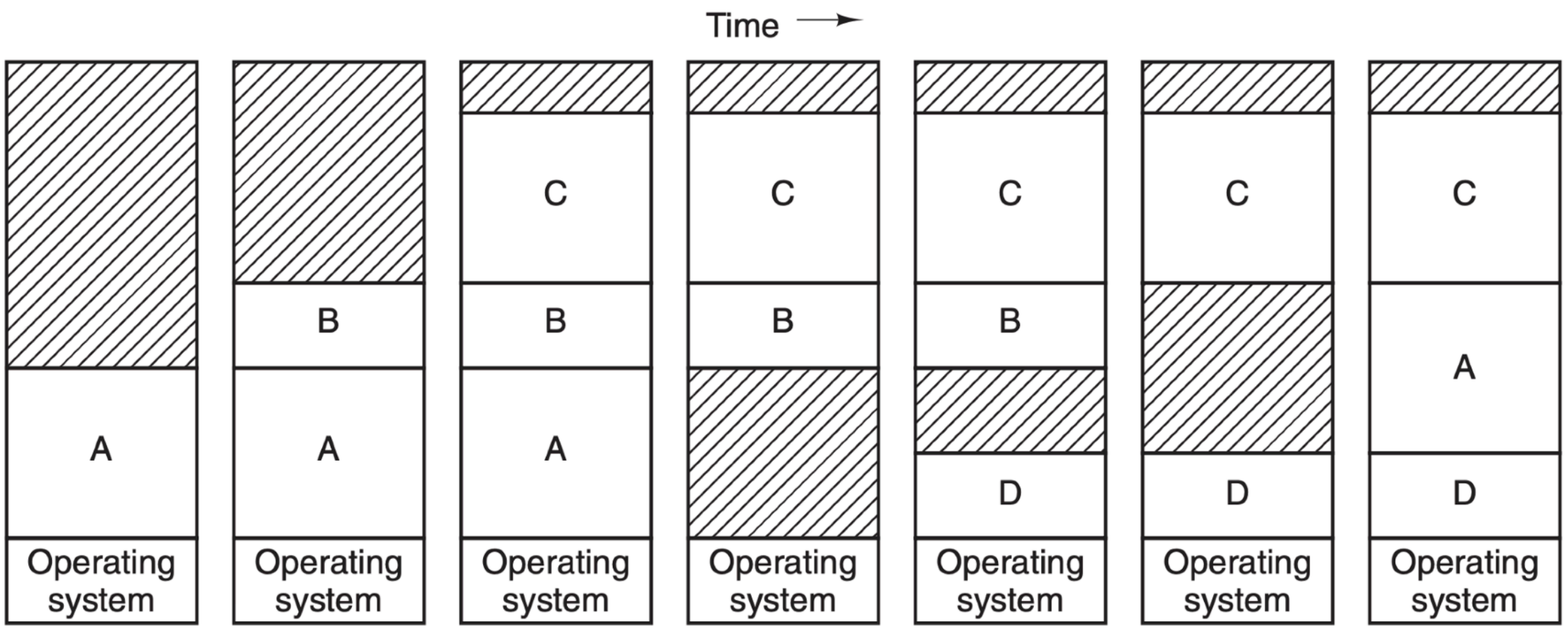
当创建一个进程 (或收到内存分配请求) 时如何分配内存空间?

- 任何存储空间管理系统都需要关注的一个问题
  - 例如，`malloc()`
- 一种基本的思想是 Multiple-Partition Allocation
  - 将物理内存划分成多个连续、且互不重叠的可分配分区
  - 可以使用 Fixed 或 Variable 的方式来进行划分
    - Fixed: 每个分区大小固定 (便于管理)
    - Variable: 每个分区大小不定

动态内存分配

动态按需分配一段连续的内存空间

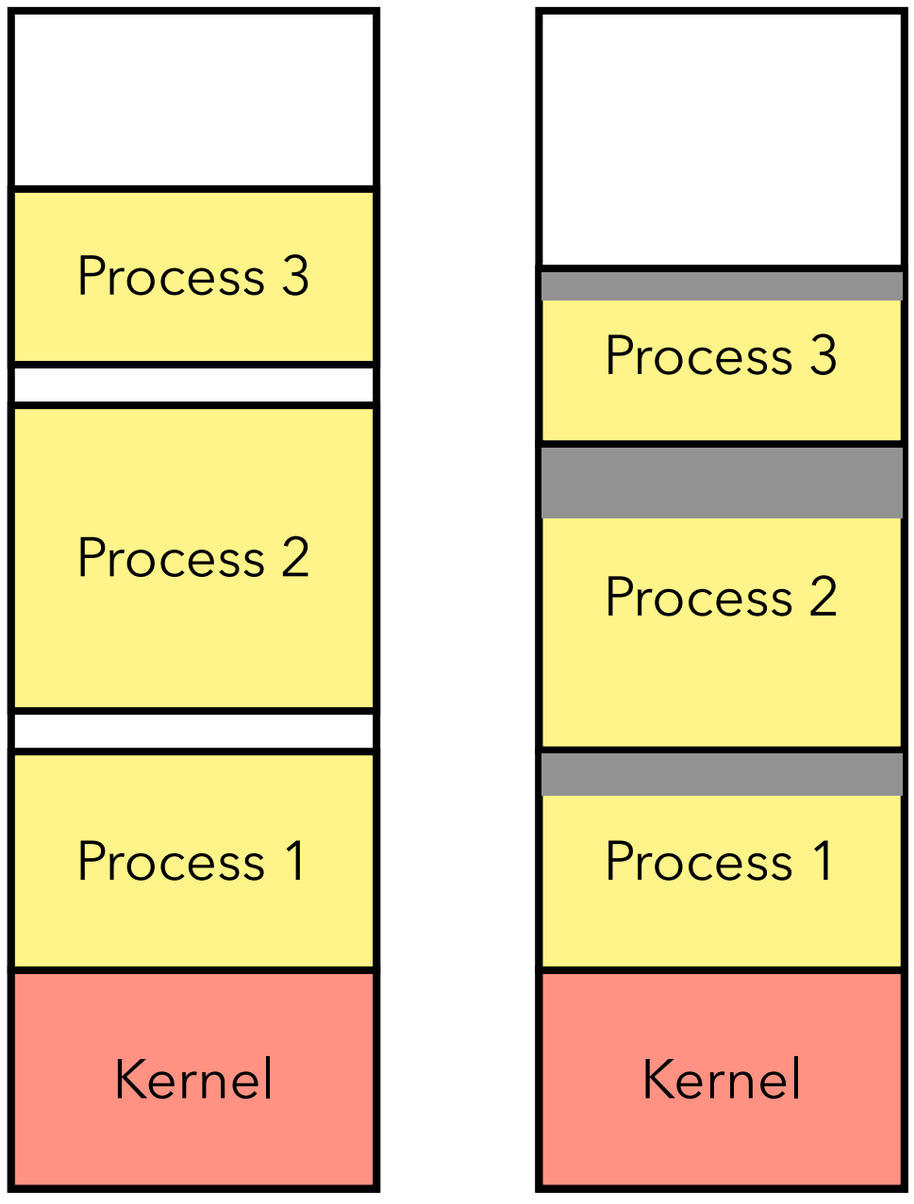
- 在内存分配时，从当前一个足够大的可用分区 (holes) 中分割出一块空间进行分配 (splitting)
- 在释放内存时，将相邻的空闲空间进行合并 (coalescing)



动态内存分配

在内存分配过程中会产生无法被分配的未使用内存 (Fragmentation)

- 外部碎片 (holes): 分区之间浪费的小内存空间 (过于小而无法分配给新的请求)
  - 可以通过移动分区来将空闲空间整合到一起 (compaction) 来降低外部碎片 (very costly)
- 内部碎片: 分区中浪费的未被使用的内存空间 (由于实际所需空间小于分区大小)

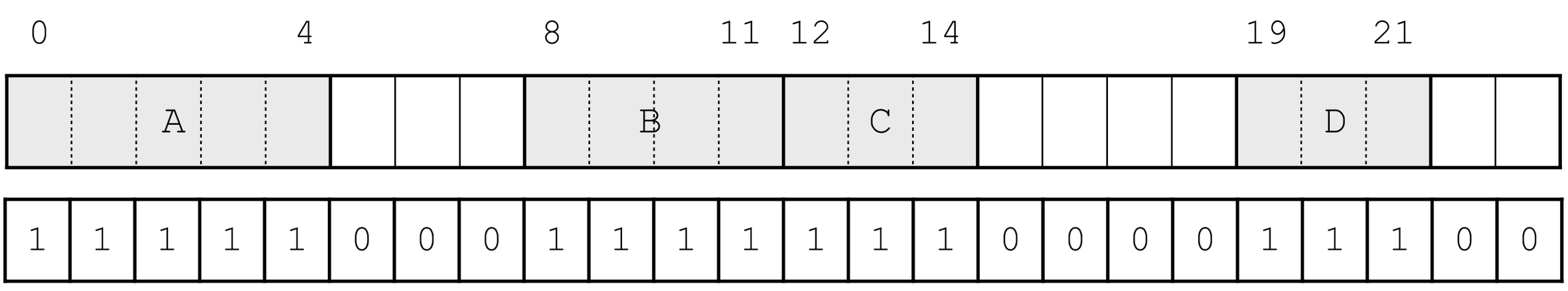


空闲空间管理

Bitmap

为了实现动态内存分配，需要对空闲空间进行管理

- 将内存空间被划分为一系列固定大小的可分配单元
- 用一个固定大小的 Bitmap 来追踪每个单元是否已分配

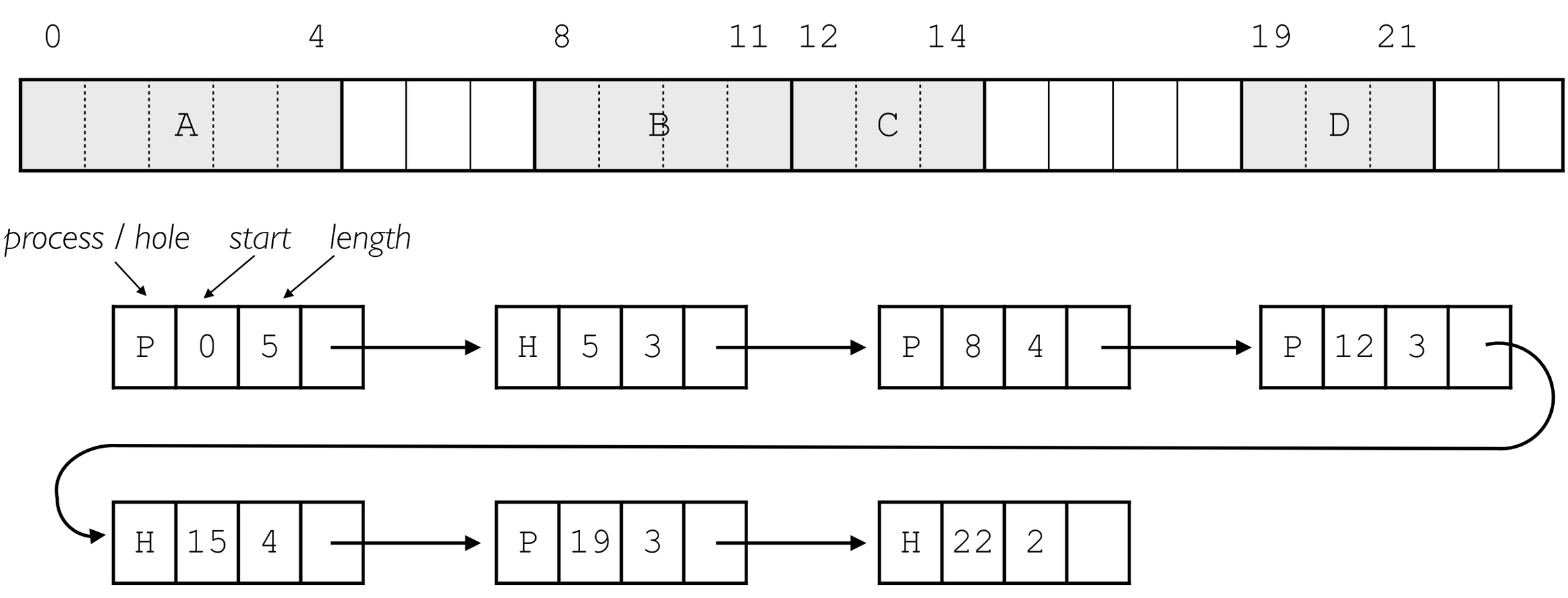


空闲空间管理

Linked List (Free List)

为了实现动态内存分配，需要对空闲空间进行管理

- 维护一个由已分配和可用分区构成的 Linked List
- 如何存储这个 Linked List (variable size)？

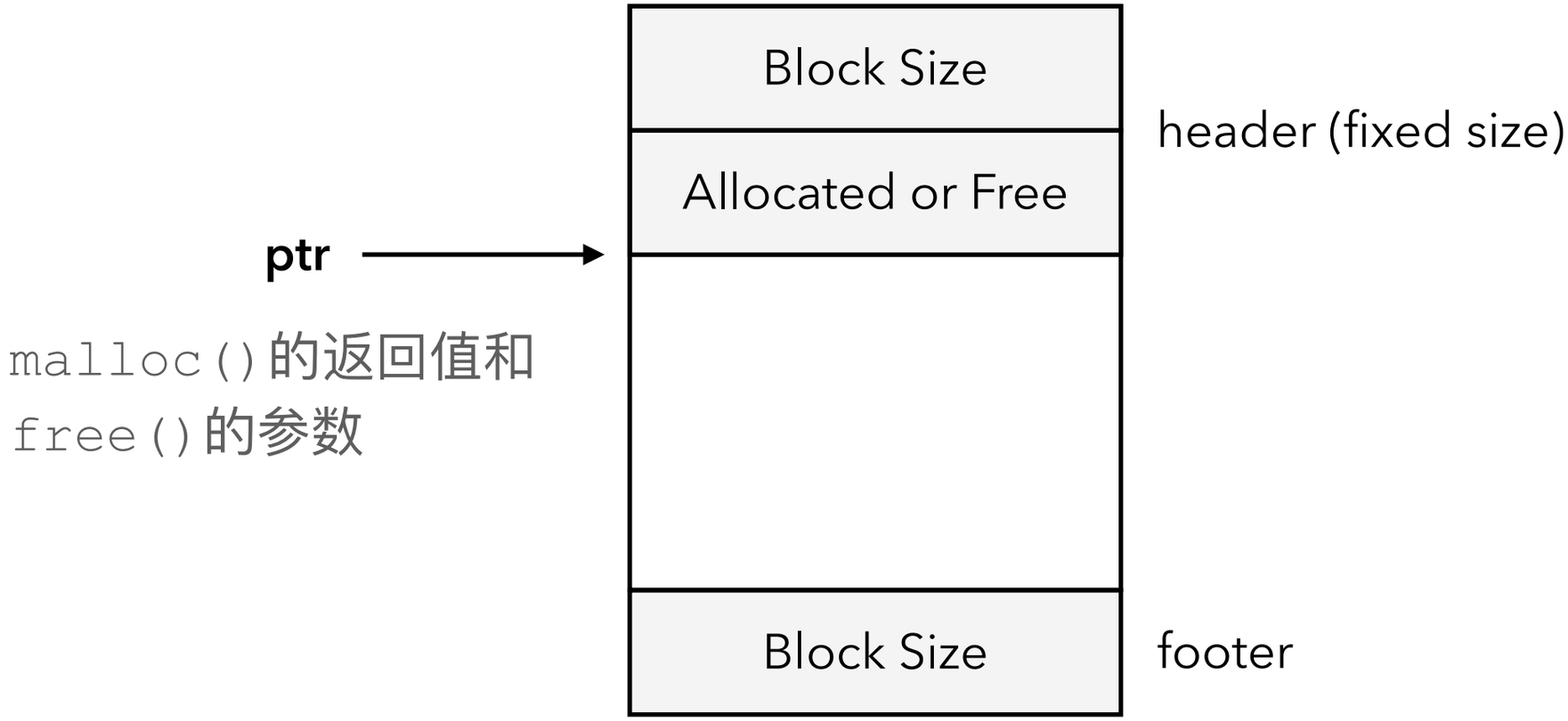


空闲空间管理

Linked List (Free List)

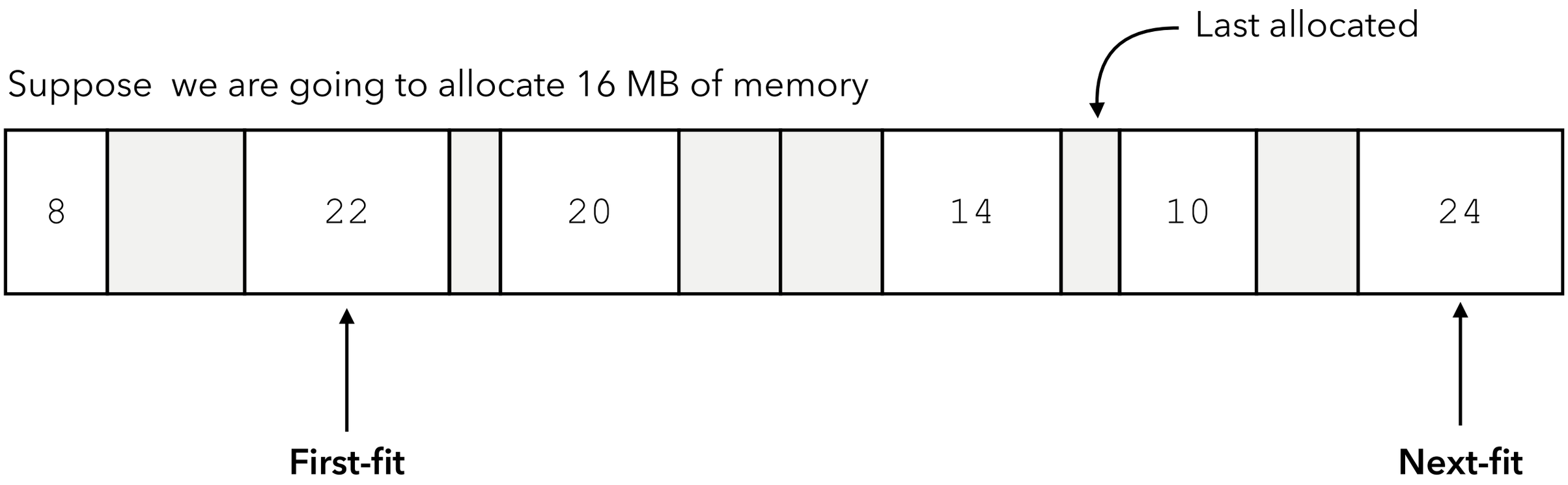
为了实现动态内存分配，需要对空闲空间进行管理

- 可以将每个分区的 metadata 也存储在相应分区中 (Implicit Free List)
- 在空闲空间合并时利用 Footer (Boundary Tag) 快速找到上一个分区



分配策略

当接收到大小为 *N* 的内存空间请求时，需要决定分配哪一块空闲空间
(Dynamic Memory Allocation Problem)

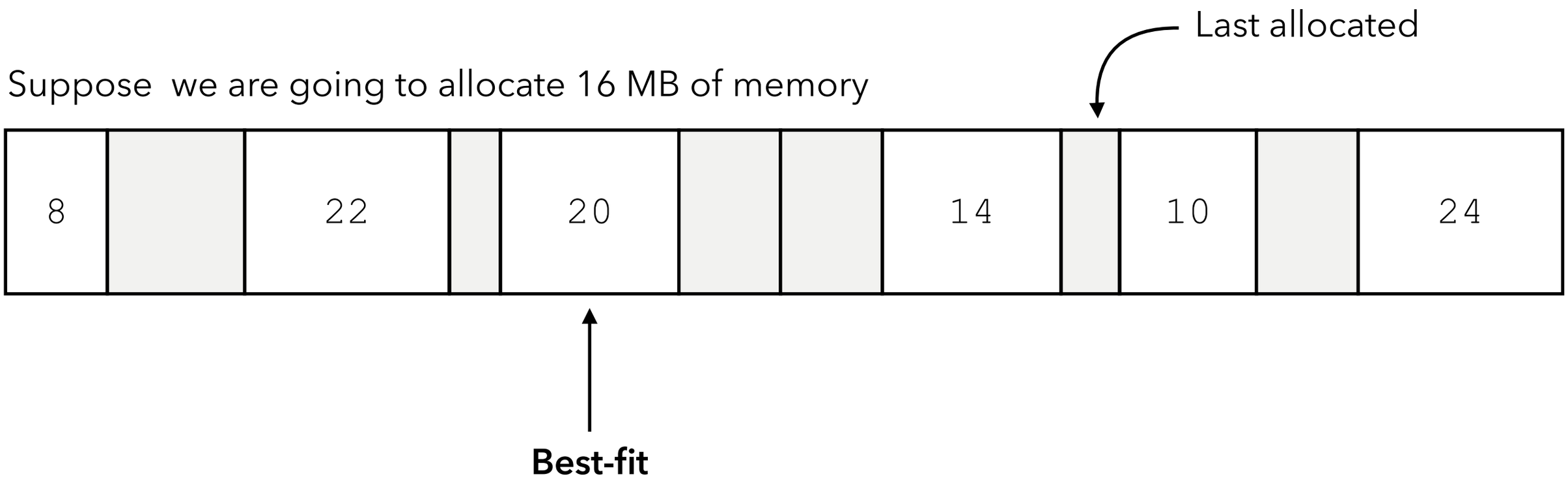


从第一个足够大的空闲分区中进行分配
(尽可能减少搜索时间)

从上一次分配的位置开始寻找
(尽可能均匀地分配)

分配策略

当接收到大小为 *N* 的内存空间请求时，需要决定分配哪一块空闲空间
(Dynamic Memory Allocation Problem)



从满足请求大小的最小空闲分区中进行分配
(需要搜索整个列表；可能会由于产生很多 tiny holes 而更浪费内存空间)

分配策略

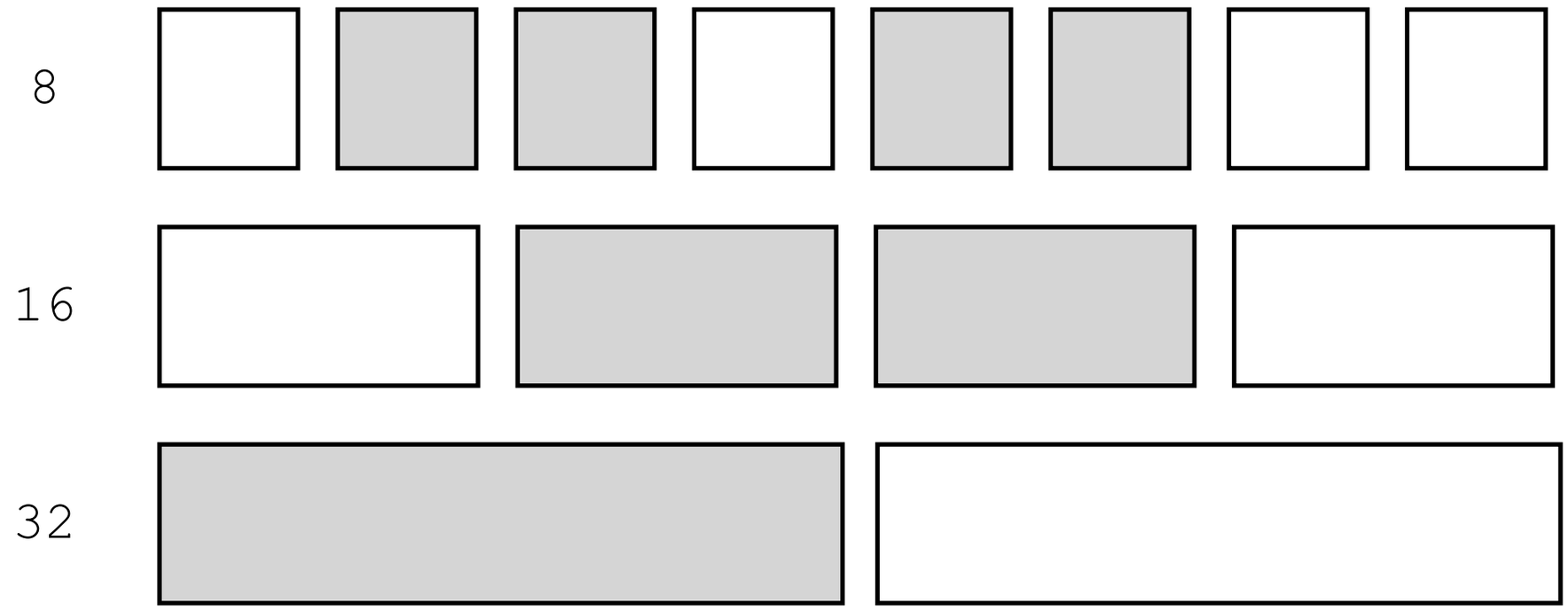
当接收到大小为 *N* 的内存空间请求时，需要决定分配哪一块空闲空间 (Dynamic Memory Allocation Problem)

- 理想情况是尽可能降低碎片，同时算法执行速度要快
- 但是空间分配和释放请求的到来可能是任意的
  - 任何特定策略在一些特殊情况下可能都会表现得很糟糕
  - 通常面向特定 workload 进行优化

分配策略

例如，考虑到操作系统内核通常频繁请求特定大小的内存空间

- 为常见的请求大小分别维护一个空闲分区列表 (Quick-Fit)
  - 能快速找到并分配一个特定大小的分区
  - 释放空间时将其重新插入特定的空闲分区列表



分配策略 *

Buddy System Allocation

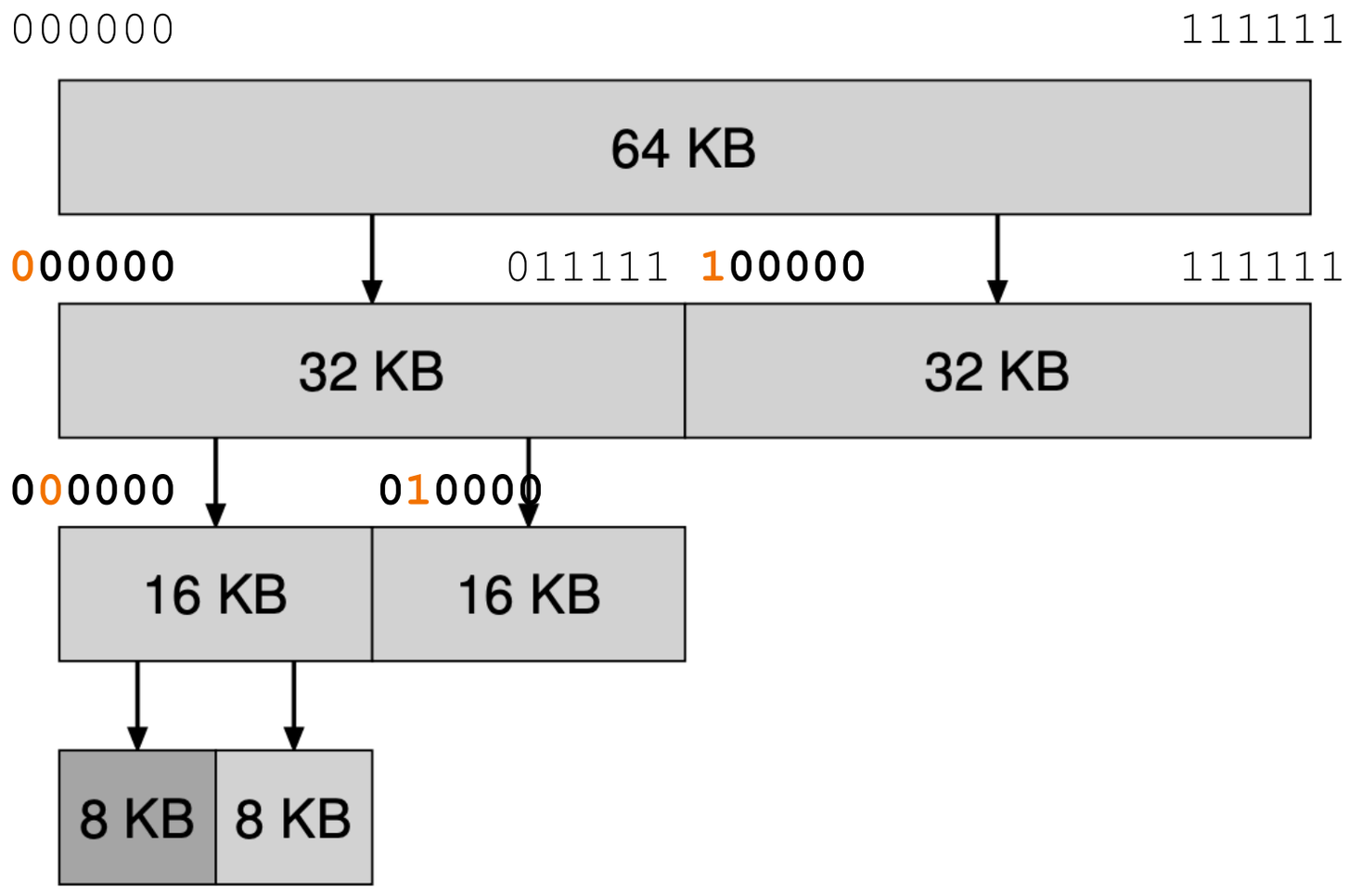
按照 2 的幂来进行内存分配 (power-of-2 allocator)

- 对一个大小为 2^N 的空闲内存空间进行管理
- 总是按 2 的幂大小来进行空间分配
- 当一个大小为 *M* 的请求到达时，递归搜索空闲空间
  - 每次都将当前可用空间一分为二，直到找到满足 *M* 向上舍入的下一个 2 的幂的块来进行分配
  - 另外一个块 (a *buddy*) 放置在合适的空闲列表中用于后续分配

分配策略 *

Buddy System Allocation

- 例如，在 2^6 = 64 KB 中分配 5 KB 大小空间
- 在释放空间时，可以快速将未使用的块合并为更大的块 (不断递归)
- 容易检查当前块的 buddy 是否空闲
- 每一对 buddy 的地址仅相差一个 bit



Address of a block of size 2^K is a multiple of 2^K
(at least *k* zeroes on the right)

分配策略

例如，`malloc()`会根据用户请求空间的大小调用不同的系统调用

- 对于小空间请求使用 `brk()` 在 `heap` 上进行分配
  - 释放时不立即归还
  - 针对高频小分配，提高系统性能
- 对于大空间请求使用 `mmap()` 在 `heap` 和 `stack` 间进行分配
  - 释放时立即归还
  - 针对低频大分配，减少 `heap` 中碎片

连续内存空间管理

使用连续内存空间管理来管理 `heap` 是相对合理的，但用来给进程分配物理内存就过于低效

- 进程 `heap` 和 `stack` 间存在的大段未使用的空间也被分配
  - 严重的内部碎片 (internal fragmentation) 问题
- 不同进程之间不能共享代码或数据

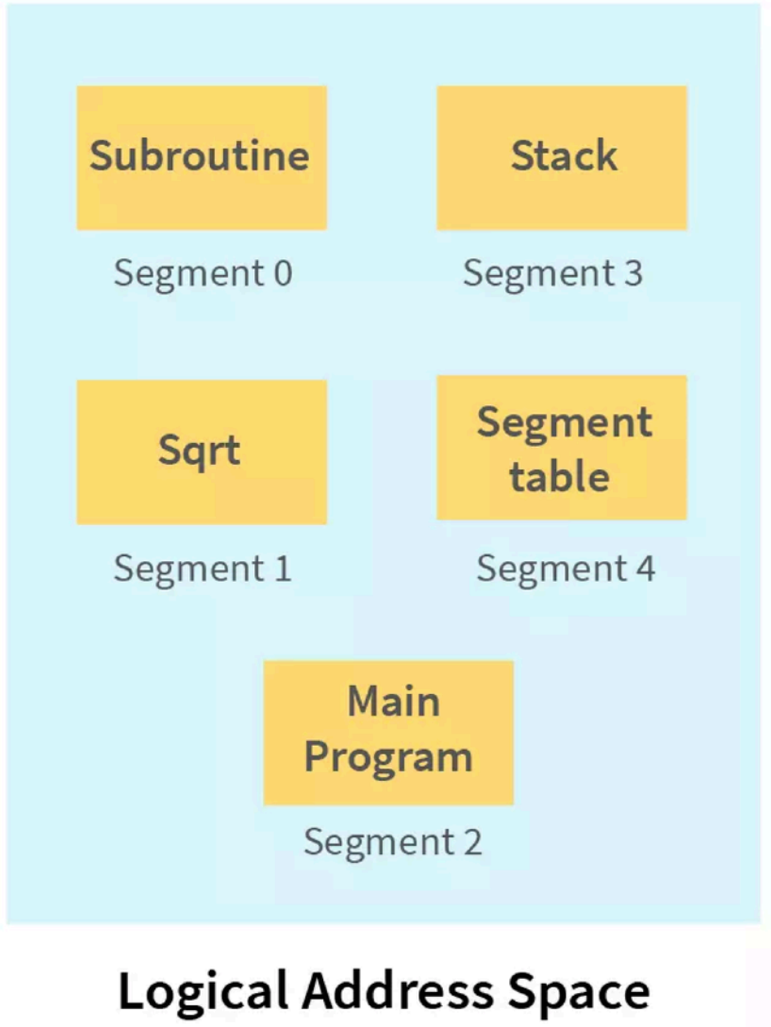
分段 & 分页

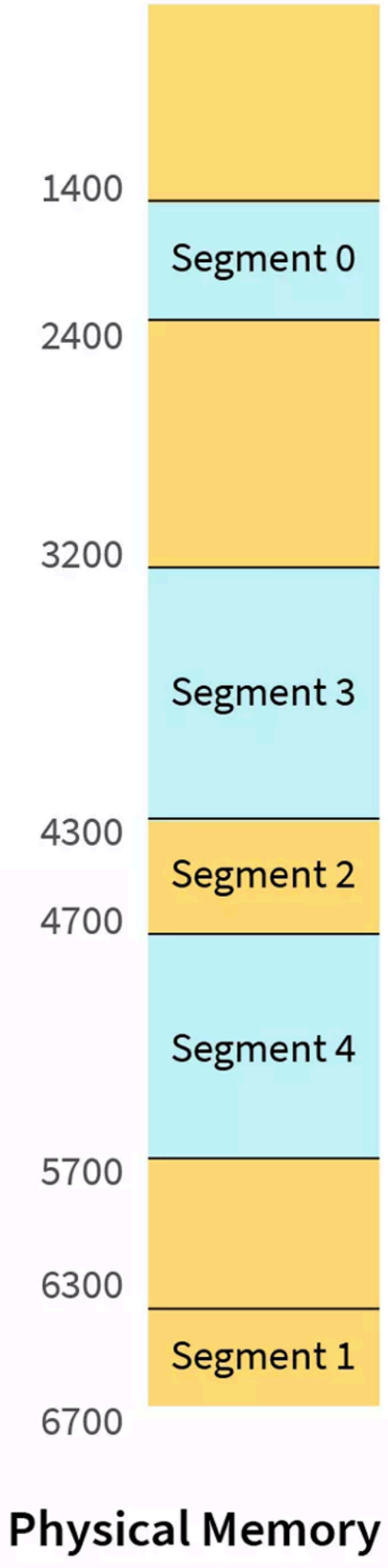
What if the physical address can be non-contiguous?

分段 Segmentation

从进程 (用户) 使用内存的视角进行内存管理

- 将进程虚拟地址空间视为一组逻辑独立的段 (segments)
- 按照段来进行内存分配：每个段独立映射到一段连续的物理内存地址
- 不同段间没有特定顺序
- 没有使用的虚拟地址空间不需要映射
- 不同的段可独立增长/缩减

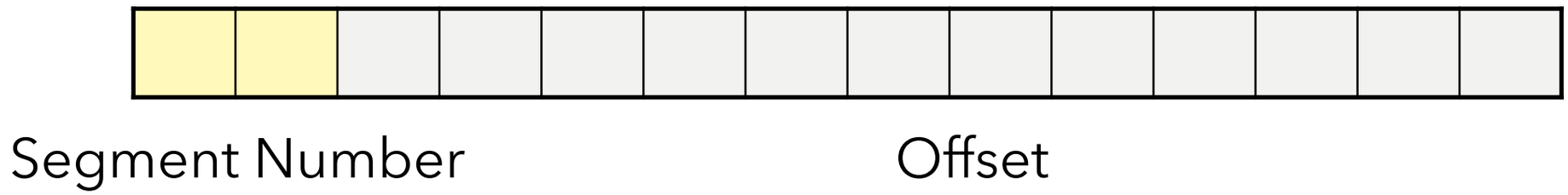




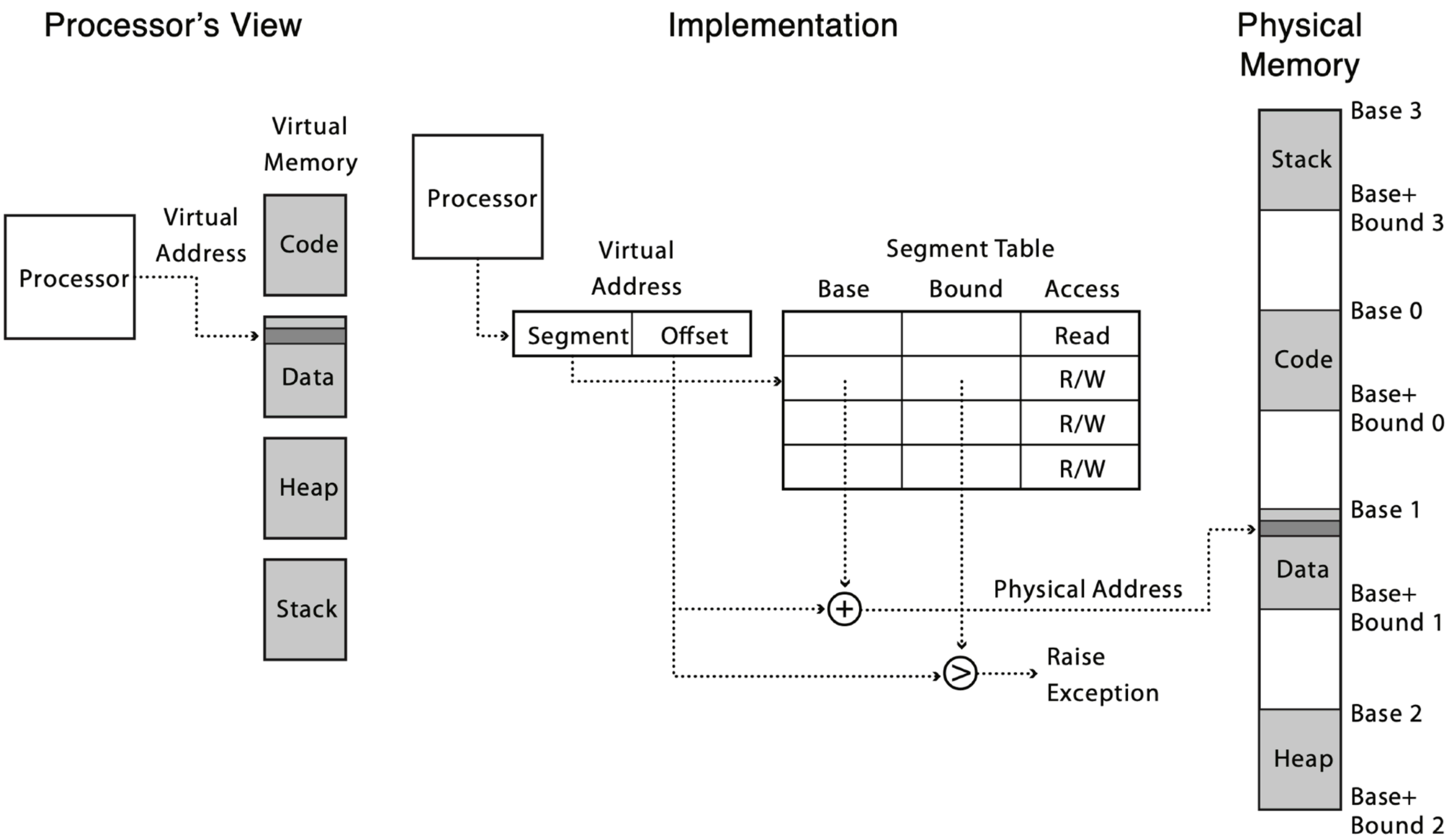
分段 Segmentation

Address Translation

在分段方式下，虚拟地址由〈**segment number, offset**〉构成



- 一种 Base and Bound 地址转换的泛化
  - 每个段的地址转换可以通过一组 Base and Bound 来实现
  - 每个进程维护一个段表 (segment table)，记录每个段的 Base and Bound 信息 (相应地，利用段号指明段表中的哪一项)

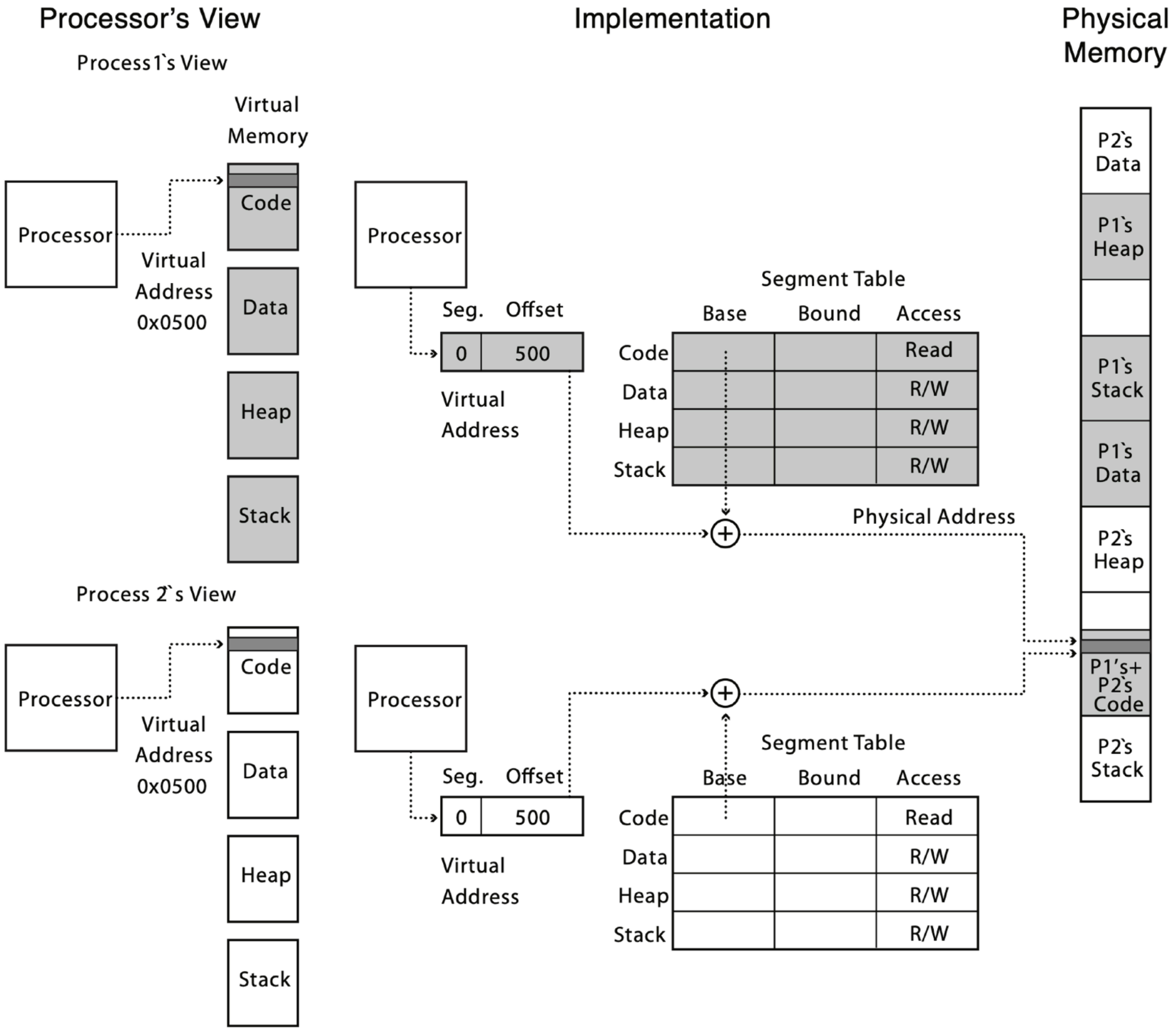

Processor's View
Virtual Memory
Processor
Virtual Address
Code
Data
Heap
Stack
Implementation
Processor
Virtual Address
Segment
Offset
Segment Table
Base
Bound
Access
Read
R/W
R/W
R/W
Physical Address
Raise Exception
Physical Memory
Stack
Base 3
Base+ Bound 3
Code
Base 0
Base+ Bound 0
Base 1
Data
Base+ Bound 1
Base 2
Heap
Base+ Bound 2

分段 Segmentation

Support for Sharing

物理内存中的同一个段可以被映射到多个虚拟地址空间中

- 让多个进程的相关段表项具有相同的 Base 和 Bound
- 例如，多个进程共享同一个代码段 (i.e., text editors, compilers, GUI)
  - 每个进程认为访问的是自己虚拟地址空间中的代码段
  - 通过地址转换，不同进程最终访问的是同一块物理内存区域
- 需要在段表中为每个段增加一些保护位 (protection bits)，来指明进程对其的权限 (read / write / execute)


Processor's View
Implementation
Physical Memory
Process1`s View
Virtual Memory
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Process 2`s View
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Processor
Seg.
Offset
0
500
Virtual Address
Segment Table
Base
Bound
Access
Code
Data
Heap
Stack
Read
R/W
R/W
R/W
Physical Address
Processor
Seg.
Offset
0
500
Virtual Address
Segment Table
Base
Bound
Access
Code
Data
Heap
Stack
Read
R/W
R/W
R/W
P2`s Data
P1`s Heap
P1`s Stack
P1`s Data
P2`s Heap
P1's+ P2`s Code
P2`s Stack

分段 Segmentation

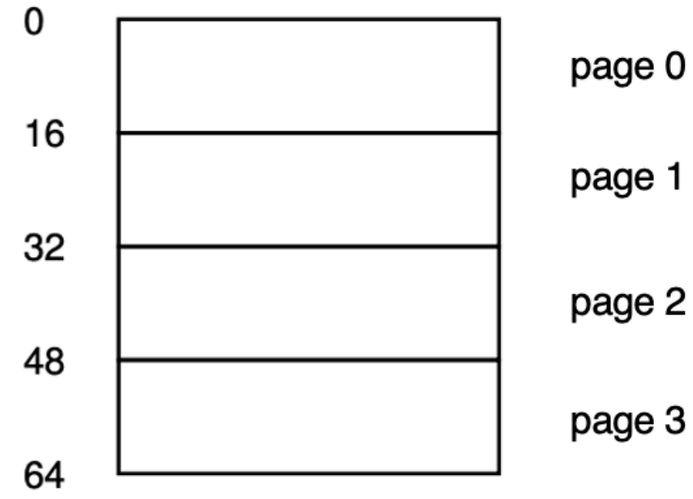
分段机制也会存在一些问题

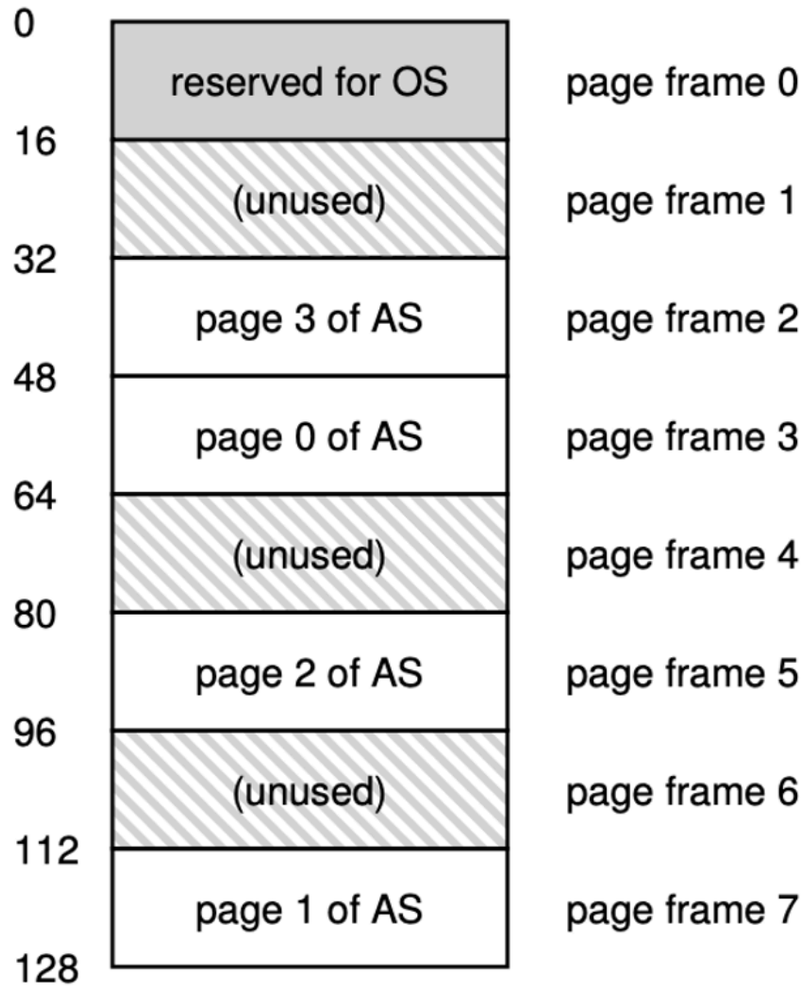
- 如果进程有一个很大但稀疏使用的 Heap？
- 如果进程对地址空间的使用模式与段的设计不匹配？
- 更重要的，每个段的长度不一样，因此是动态内存空间分配问题 (dynamic memory allocation problem)
- 可能会随时间推移产生很多外部碎片 (external fragmentation)

分页 Paging

基于固定大小单元来管理内存 (Fixed-Sized Partition)

- 将物理内存划分为若干固定大小的页 (Page) (也叫页框/页帧 Page Frame)
- 虚拟地址空间也按同样大小的页划分
- 非常灵活且易于管理
  - 任意虚拟页可以映射到任意物理页
  - OS 只需要维护一个空闲页框的列表，进程需要几个页就分配几个页
- 没有外部碎片 (external fragmentation)

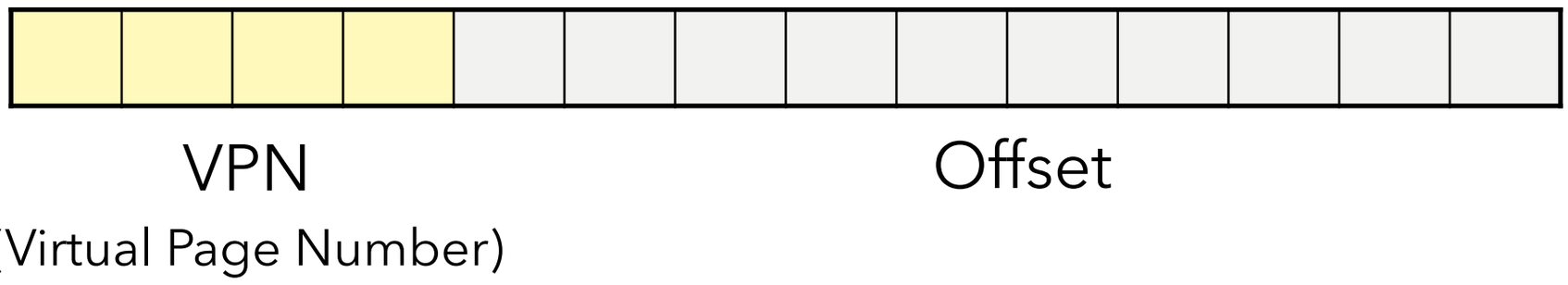




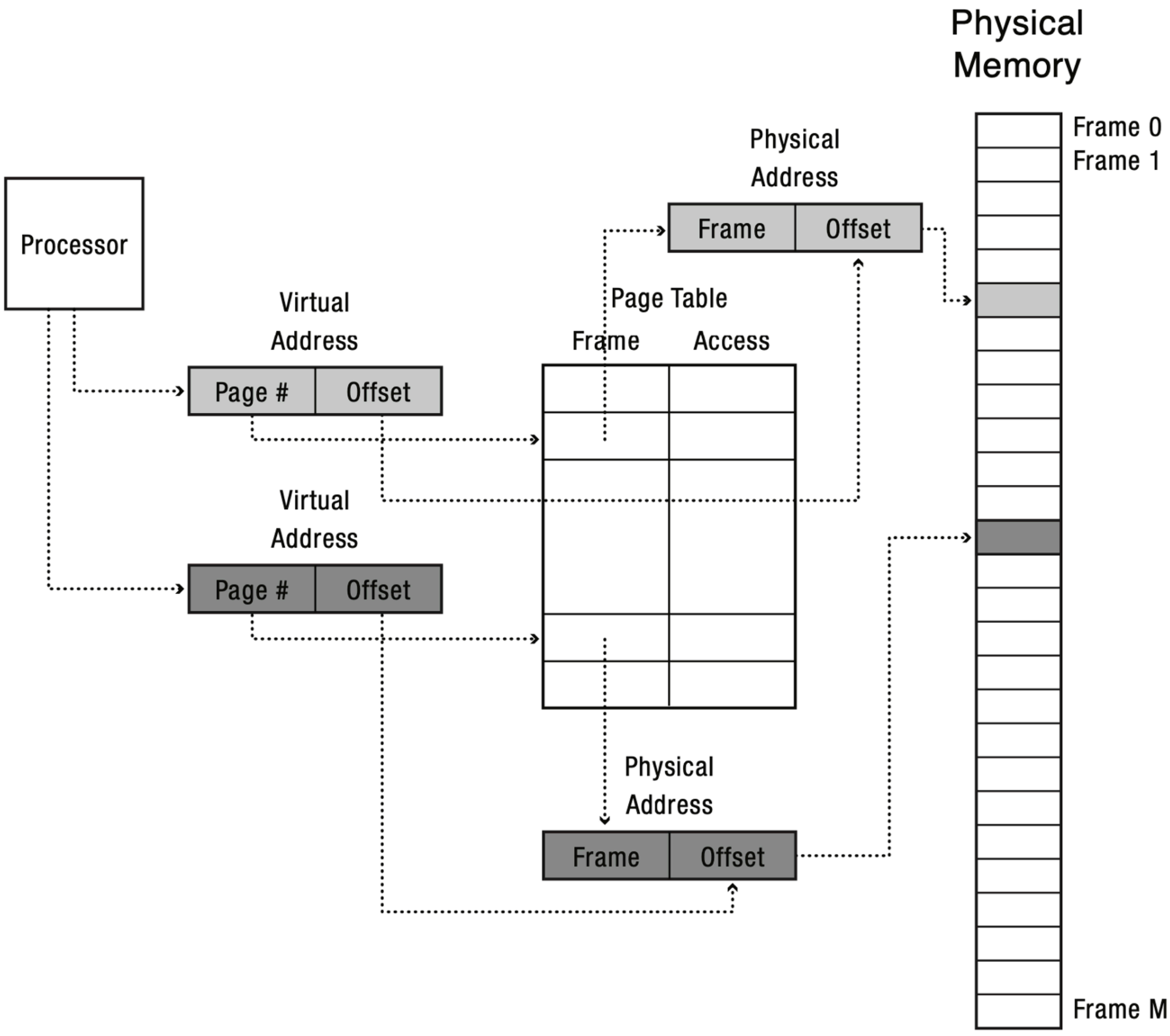
分页 Paging

Address Translation

在分页方式下，虚拟地址由〈**page number, offset**〉构成



- 也是一种 Base and Bound 地址转换的泛化
  - 通过一个 Base 实现每个页的地址转换 (页大小已经限定了 Bound)
  - 每个进程维护一个页表 (page table)，其中每一个页表项 (page table entry, PTE) 记录该页的 Base 信息
    - 即该页对应的物理页框号 (Physical Page Frame Number, PFN)


Physical Memory
Processor
Physical Address
Frame
Offset
Frame 0
Frame 1
Virtual Address
Page #
Offset
Page Table
Frame
Access
Virtual Address
Page #
Offset
Physical Address
Frame
Offset
Frame M

分页 Paging

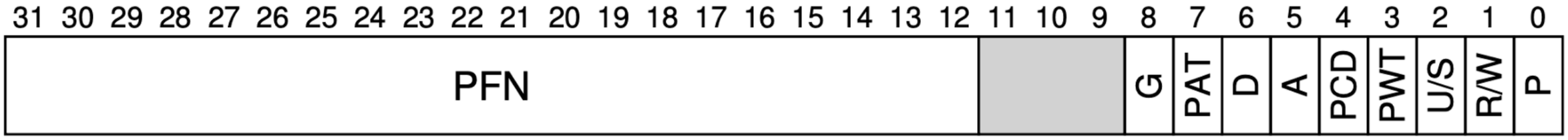
Page Table

在分页机制中，页表 (page table) 是存储虚拟地址到物理地址映射方式的重要数据结构

- 线性页表 (an array of PTE) 是最简单的一种数据结构
- 由硬件完成地址转换，因此 PTE 的具体设计和机器硬件相关
  - 需要有足够的 bits 来存储页框号 (physical page frame number)
  - 预留足够的 bits 来存储相关控制信息
  - 通常按 byte 对齐

分页 Paging

Page Table (x86)



- Page frame number (`PFN`)
- Present bit (`P`): whether this page is in physical memory
- Read and write bit (`R/W`): whether can be written to
- Reference and dirty bits (`A`, `D`): recent access
- User/supervisor bit (`U/S`): if user-mode process can access the page
- Caching related bits (`PWT`, `PCD`, `PAT`, `G`): how hardware caching works for the page

分页 Paging

Page Table

Linux 中的 `/proc/[PID]/pagemap` 提供了访问进程页表相关信息的接口 (需要 root 权限)

- 每一个 PTE 项的长度为 8 bytes

  - Bits 0-54 page frame number (PFN) if present
  - Bits 0-4 swap type if swapped
  - Bits 5-54 swap offset if swapped
  - Bit 55 pte is soft-dirty (see Documentation/admin-guide/mm/soft-dirty.rst)
  - Bit 56 page exclusively mapped (since 4.2)
  - Bits 57-60 zero
  - Bit 61 page is file-page or shared-anon (since 3.5)
  - Bit 62 page swapped
  - Bit 63 page present

https://www.kernel.org/doc/html/v4.18/admin-guide/mm/pagemap.html

分页 Paging

Page Table

计算进程虚拟地址 VA 对应的物理地址 FA

```
int fd = open("/proc/[PID]/pagemap", O_RDONLY);

// seek to the correct position in the pagemap file
unsigned long offset = (unsigned long) VA / PAGE_SIZE * PAGEMAP_LENGTH;
lseek(fd, offset, SEEK_SET);

// read the PTE entry
uint64_t pagemap_entry;
read(fd, &pagemap_entry, sizeof(uint64_t));

// extract the page frame number, which is in bits 0-54 of PTE
unsigned long PFN = pagemap_entry & ((1ULL << 55) - 1);

// calculate the physical address (a generalised base & bound)
unsigned long FA = (PFN * PAGE_SIZE) + ((unsigned long) VA % PAGE_SIZE);
```

分页 Paging

Support for Sharing

为了让多个进程共享物理内存，只需要让这些进程各自 PTE 映射到同一个物理页框即可 (需要特定的保护位指明访问权限)