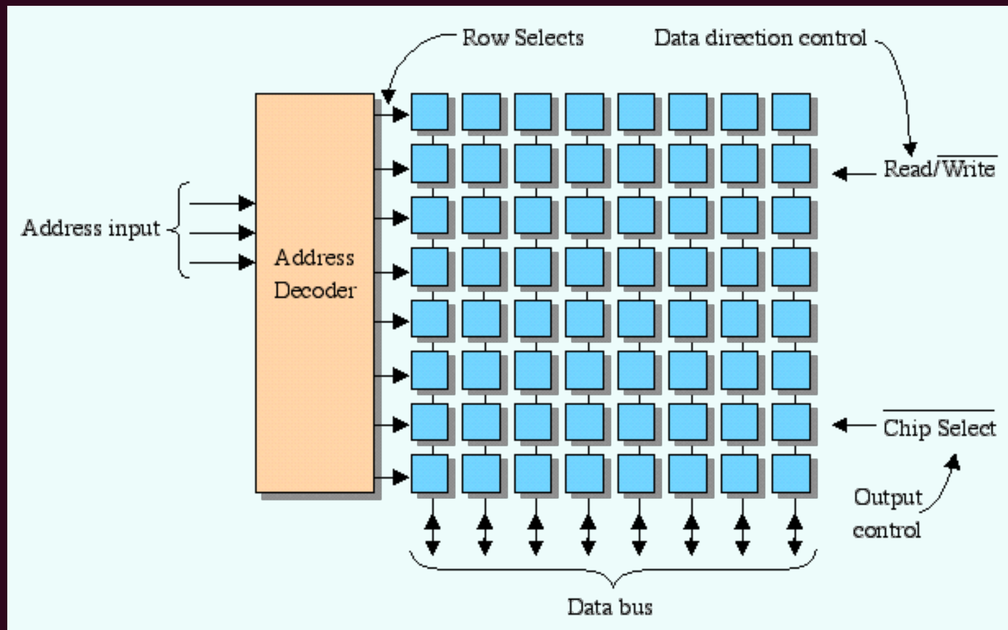
内存管理
Memory Management

钮鑫涛
南京大学
2026春

本门课由蒋炎岩和钮鑫涛共同构建

物理内存

- 物理内存没有其他意义，只是存储（易失）运行信息的一个物理介质
  - 连续字节/字（Byte）的数组，每行都有自己的地址（n位地址总共有2^n个字节）
  - 通过地址可以写/读相应的字节
  - 地址和数据分别通过地址总线和数据总线进行传播



保护(Protection)

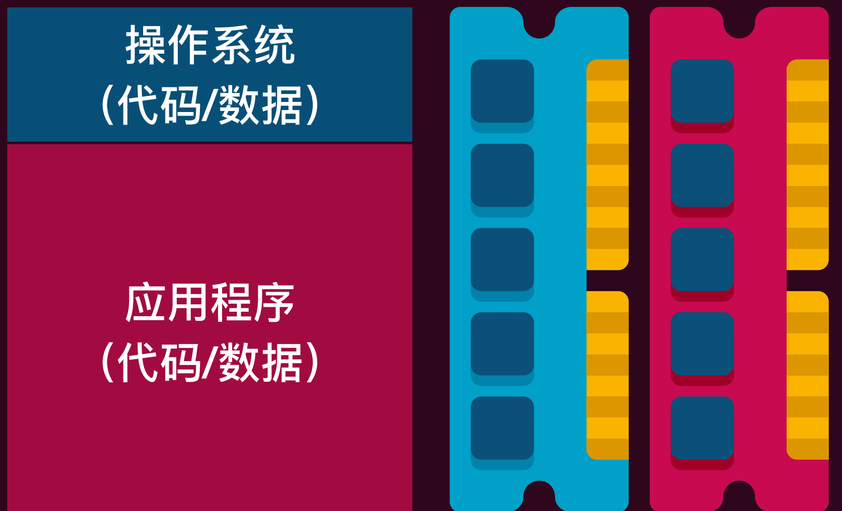
- 有了进程抽象，内存就有了“语义”
  - 其代表了当前进程的“状态”!
  - 而这个“状态”应该遵循状态机（即进程）的规约而改变
    - 比如规约：给定分配内存$S = [l_0, r_0) \cup [l_1, r_1) \cup \ldots$下， `kalloc(s)` 返回的$[l, r)$必须满足$[l, r) \cap S = \varnothing$
    - 违背规约就会出现“undefined state”， 从而导致“undefined behavior”, 因此作为状态机的管理者—操作系统，必须提供出现违背规约之后的保护

保护(Protection)

- 哪些“状态”的改变操作是非法的呢?
  - 修改别的状态机的状态!
    - 需要界定状态机的边界（哪些内存是自己的，哪些是别的状态机的）
  - 增加状态机的大小时，占用的额外空间还处于“有用”状态
    - 我们只能申请“空闲”的内存来增加状态机的大小，而不能使用尚被占用的空间（无论是被别的进程还是自己占用），因此得维护什么是“空闲”内存
  - 不遵守状态机的内部状态的读、写权限
    - 需要标记内存块的权限，并且能assert相应的读写是否符合权限

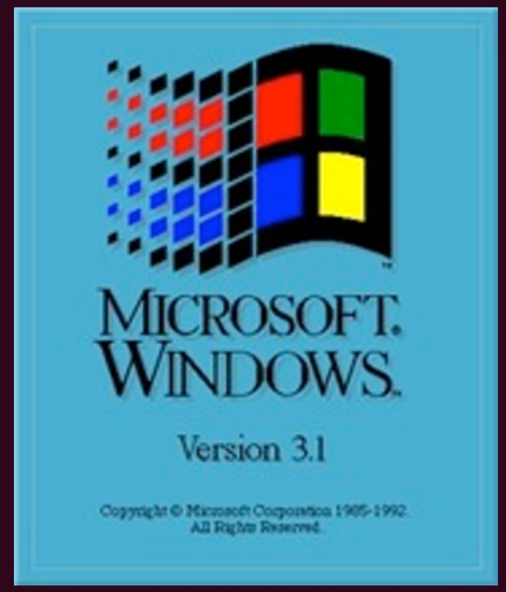
最早期的计算机系统

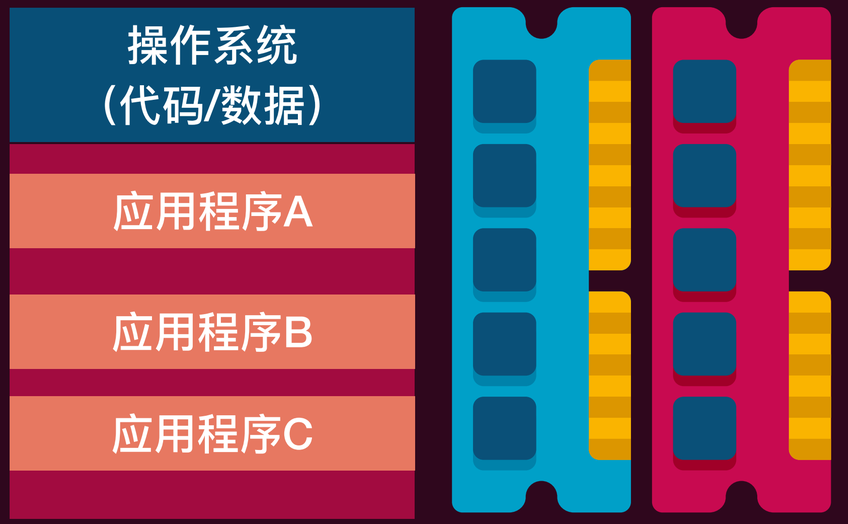
- 保护不是一开始就有的，在单道程序设计(Uniprogramming)年代：
  - 硬件
    - 物理内存容量小
  - 软件
    - 单个应用程序 + (简单)操作系统 (各自使用内存的一部分)
    - 直接面对物理内存编程
    - 由于一次只能运行一个应用程序，应用程序总是运行在物理内存中的同一位置



原始多道程序设计

- 多道程序设计(Multiprogramming)允许内存中有多个进程
- 上下文切换可以达成分时操作系统
- 编译后的程序的地址是可重位的(relocatable), 比如一个符号的地址可以变为类似“在这个模块后的14个字节”等
  - 使用装载器/链接器：在程序加载到内存时转换为绝对地址（absolute addresses）(这就是静态重定位)

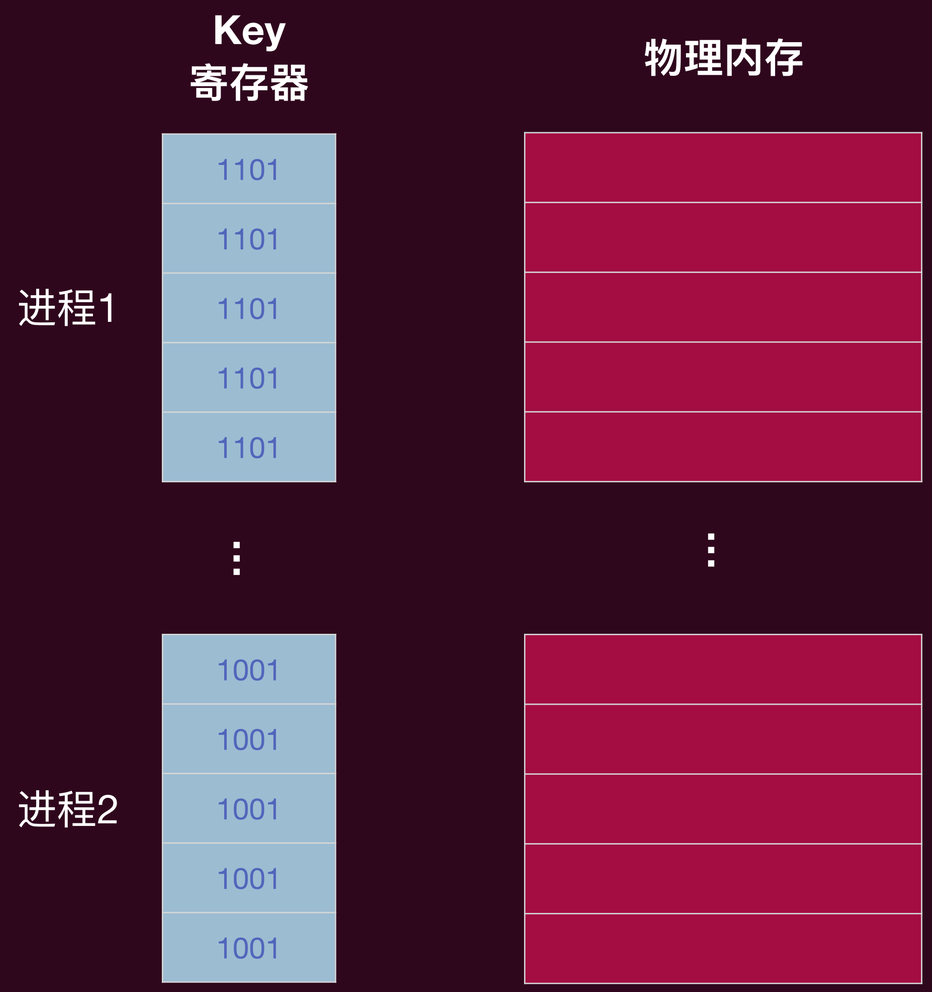




但早期的多道程序设计没有保护机制，一个程序中的bug会导致其他程序也crash（包括OS crash），程序也没有被禁止访问其他进程的地址

增加保护位?

- IBM/360采用了这个方法：Protection Key
  - 内存被划分为一个个大小为2KB的内存块(Block)
  - 每个内存块有一个4-bit的key，保存在寄存器中
  - 每个进程对应一个key
    - CPU用另一个专门的寄存器，保存当前运行进程的key
    - 不同进程的key不不同
  - 一个进程访问一块内存时：
    - CPU检查进程Key与内存Key是否匹配

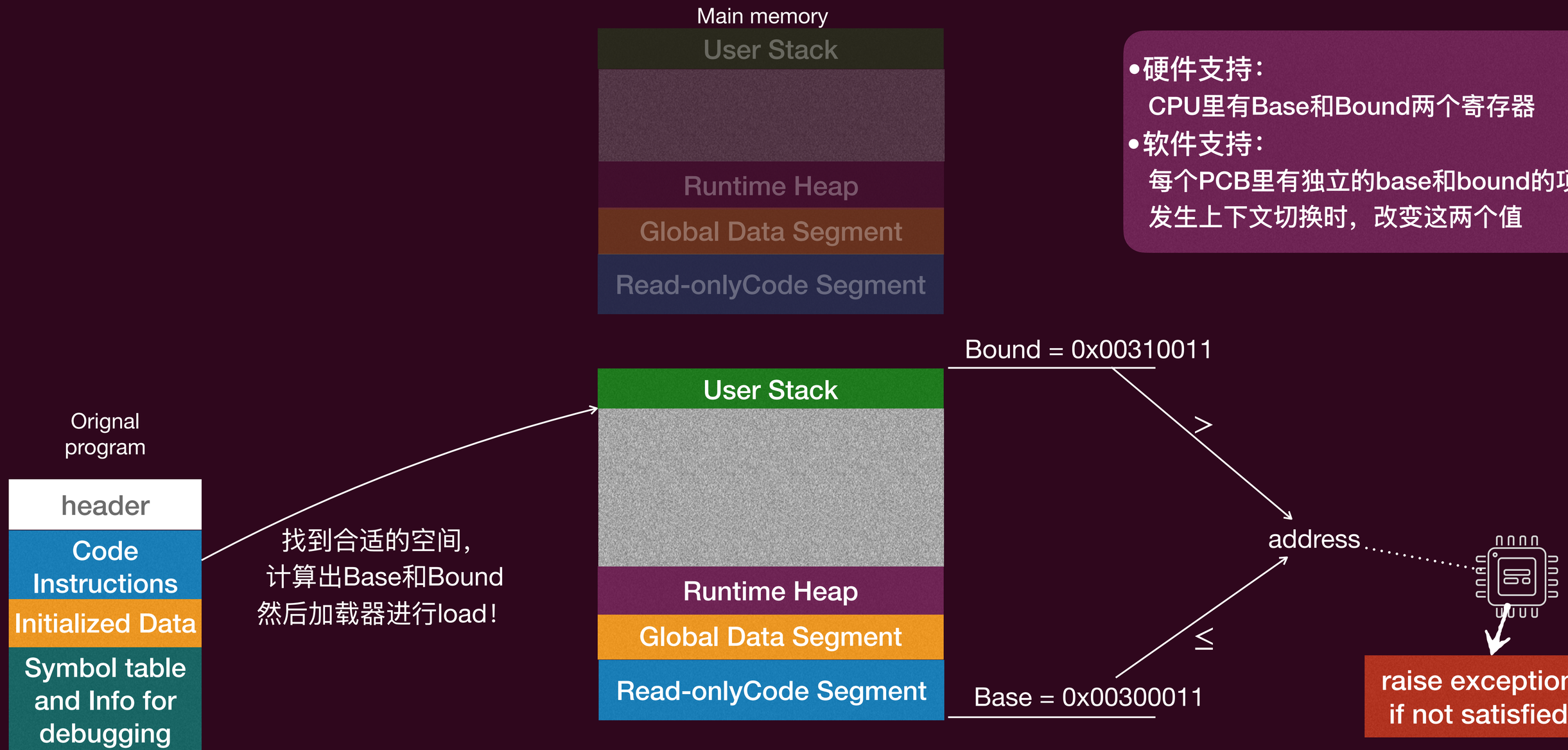


问题： 需要太多寄存器了！不现实！

有保护的多道程序设计

- 一种通用的保护方法：
  - 利用基址（Base）和界限（Bound/Limit）来保护各自的内存
  - 比如大型机Cray-1就采用了这个机制

有保护的多道程序设计



- 硬件支持：
  CPU里有Base和Bound两个寄存器
- 软件支持：
  每个PCB里有独立的base和bound的项
  发生上下文切换时，改变这两个值

使用物理地址的缺点

- 物理地址对应用是可知的，导致：
  - 一个应用会因其他应用的加载而受到影响(loader加载器压力骤增)
  - 一个应用可通过自身的内存地址，猜测出其他应用的加载位置
- 是否可以让应用看不见物理地址?
  - “看不见”，指应用对物理地址不可知
  - 一个进程不用关心其他进程占了什么地址，不受其他进程的影响
  - 看不见其他进程的信息，带来更强的隔离和保护能力

“All problems in computer science can be solved by another level of indirection”

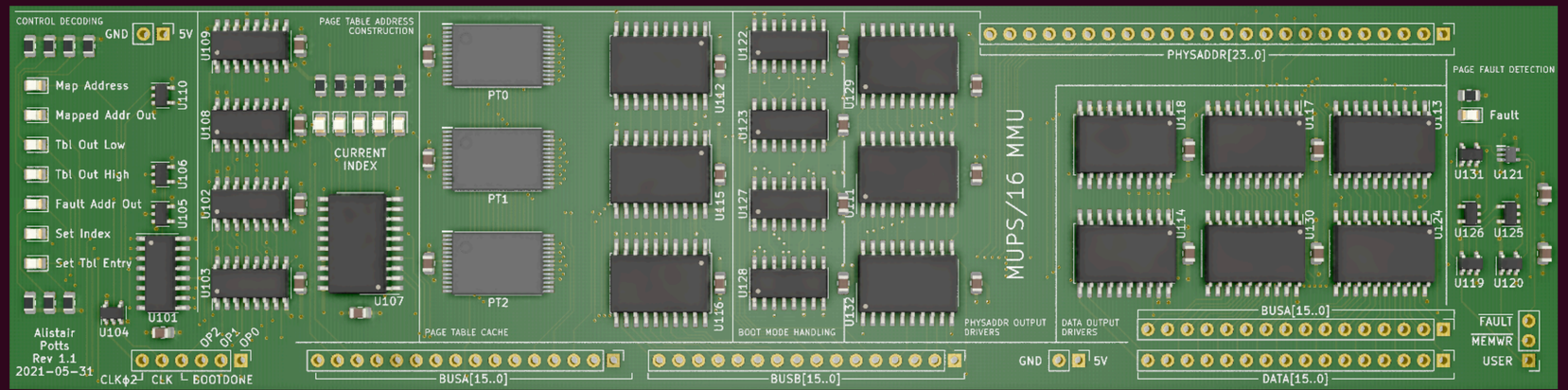
David Wheeler

Know as the “The fundamental theorem of software engineering”, which is actually a general principle for managing complexity through abstraction.

But be careful with your abstractions though, the full quote is: "All problems in computer science can be solved by another level of indirection, except for the problem of too many levels of indirection,”.

虚拟内存抽象

- 以虚拟内存抽象为核心的内存管理
  - **CPU**:支持虚拟内存功能，新增了虚拟地址空间（通过Memory Management Unit, MMU单元）
  - **操作系统**: 配置并使能虚拟内存机制
  - **所有软件**: 均使用**虚拟地址**，无法直接访问**物理地址**



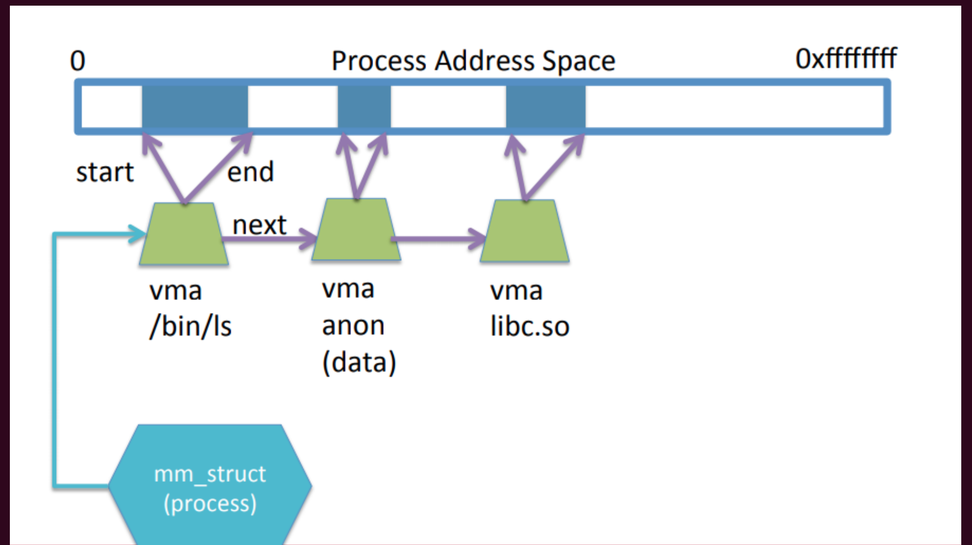
一般集成到CPU核上

虚拟/逻辑地址（ Virtual/Logical Address）

- 虚拟内存抽象下，程序使用虚拟地址访问主存
  - 虚拟地址会被硬件"自动地"翻译成物理地址
- 每个应用程序拥有独立的虚拟地址空间
  - 应用程序认为自己独占整个内存(透明性)
  - 应用程序不再看到物理地址
  - 应用加载时不用再为地址增加一个偏移量

Linux虚拟空间

Process	Address space	Virtual memory segment
task_struct	mm_struct	vm_area_struct
... mm ...	start_code start_stack start_brk	vm_start vm_end vm_flags *vm_file vm_pgoff



Linux虚拟空间

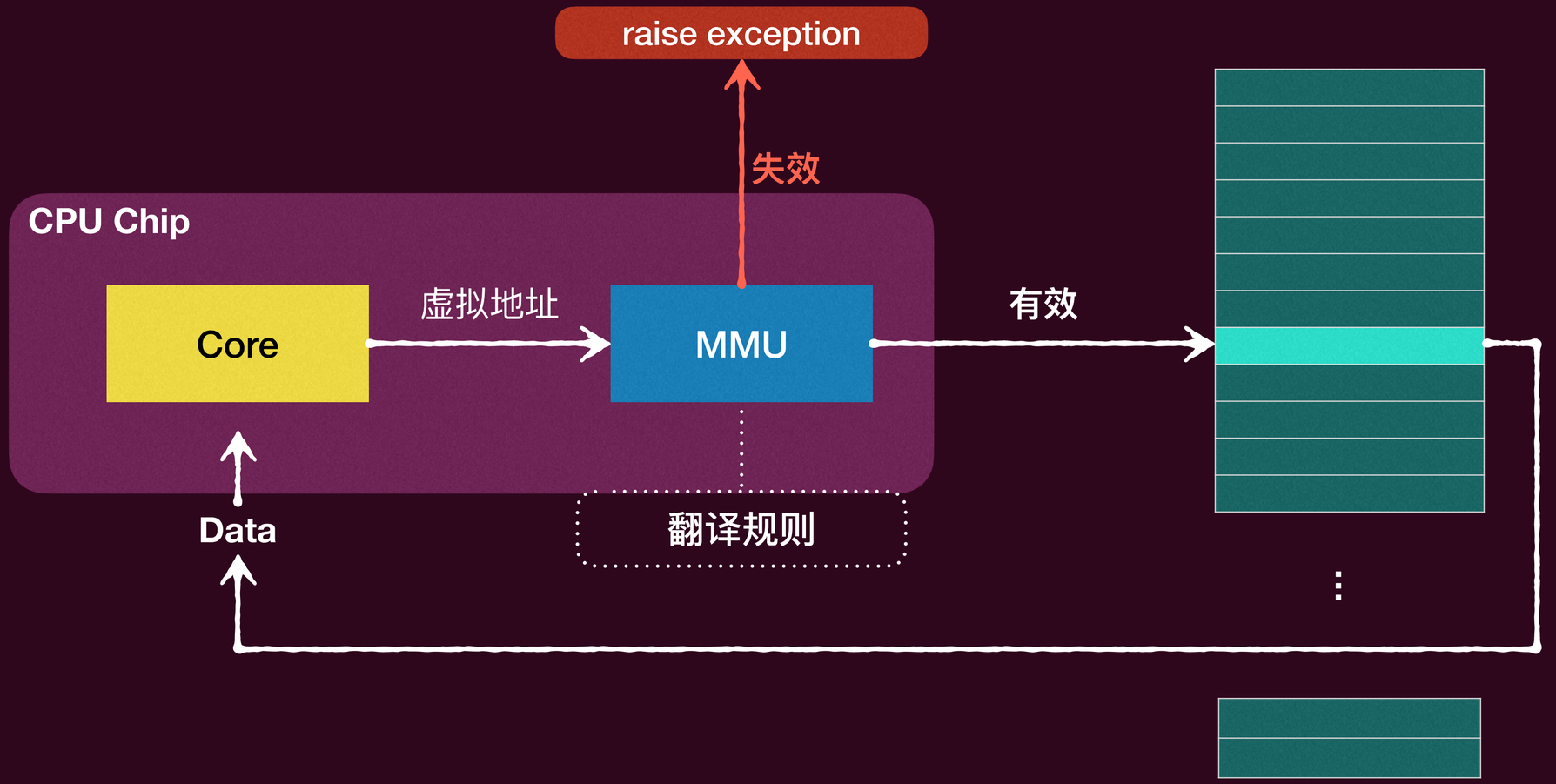
- 利用 /proc/[PID]/maps 可以看到虚拟空间的信息

```
xintao@llvmubuntu:~$ cat /proc/self/maps
55c34cfb2000-55c34cfb4000 r--p 00000000 103:02 4850025                   /usr/bin/cat
55c34cfb4000-55c34cfb9000 r-xp 00002000 103:02 4850025                   /usr/bin/cat
55c34cfb9000-55c34cfbb000 r--p 00007000 103:02 4850025                   /usr/bin/cat
55c34cfbb000-55c34cfbc000 r--p 00008000 103:02 4850025                   /usr/bin/cat
55c34cfbc000-55c34cfbd000 rw-p 00009000 103:02 4850025                   /usr/bin/cat
55c36ab96000-55c36abb7000 rw-p 00000000 00:00 0                          [heap]
7ff8f5800000-7ff8f5d98000 r--p 00000000 103:02 4850629                   /usr/lib/locale/locale-archive
7ff8f5dde000-7ff8f5e00000 rw-p 00000000 00:00 0
7ff8f5e00000-7ff8f5e28000 r--p 00000000 103:02 4852504                   /usr/lib/x86_64-linux-gnu/libc.so.6
7ff8f5e28000-7ff8f5fbe000 r-xp 00028000 103:02 4852504                   /usr/lib/x86_64-linux-gnu/libc.so.6
7ff8f5fbe000-7ff8f600d000 r--p 001be000 103:02 4852504                   /usr/lib/x86_64-linux-gnu/libc.so.6
7ff8f600d000-7ff8f6011000 r--p 0020c000 103:02 4852504                   /usr/lib/x86_64-linux-gnu/libc.so.6
7ff8f6011000-7ff8f6013000 rw-p 00210000 103:02 4852504                   /usr/lib/x86_64-linux-gnu/libc.so.6
7ff8f6013000-7ff8f6020000 rw-p 00000000 00:00 0
7ff8f603d000-7ff8f6040000 rw-p 00000000 00:00 0
7ff8f6052000-7ff8f6054000 rw-p 00000000 00:00 0
7ff8f6054000-7ff8f6058000 r--p 00000000 00:00 0                          [vvar]
7ff8f6058000-7ff8f605a000 r-xp 00000000 00:00 0                          [vdso]
7ff8f605a000-7ff8f605b000 r--p 00000000 103:02 4852495                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff8f605b000-7ff8f6088000 r-xp 00001000 103:02 4852495                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff8f6088000-7ff8f6092000 r--p 0002e000 103:02 4852495                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff8f6092000-7ff8f6094000 r--p 00038000 103:02 4852495                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff8f6094000-7ff8f6096000 rw-p 0003a000 103:02 4852495                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffd5b8aa000-7ffd5b8cb000 rw-p 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

- 可以通过系统调用brk、mmap、munmap 来改变虚拟空间的映射和分配

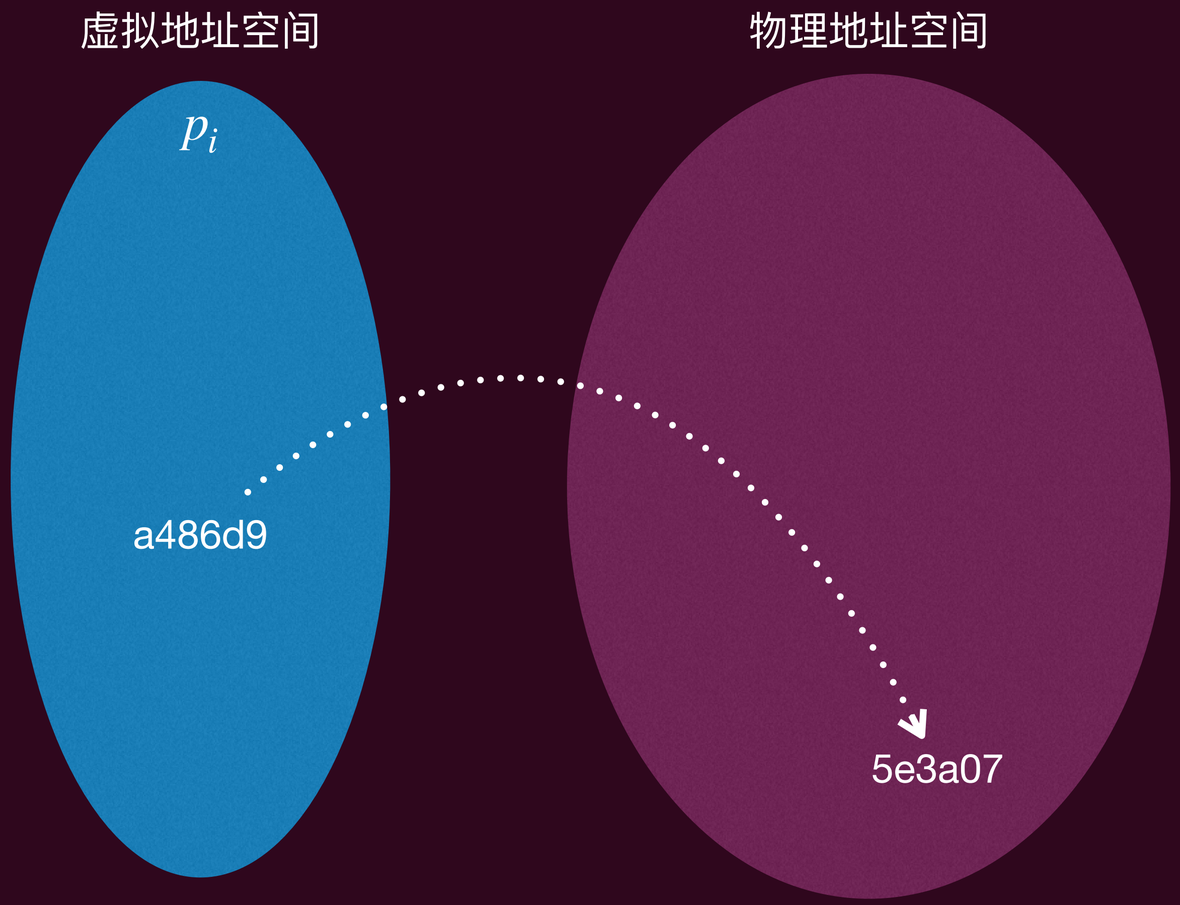
地址翻译(Address Translation)

- 翻译就是一个函数f：其将〈pid, virtual address〉映射到physical address



地址翻译

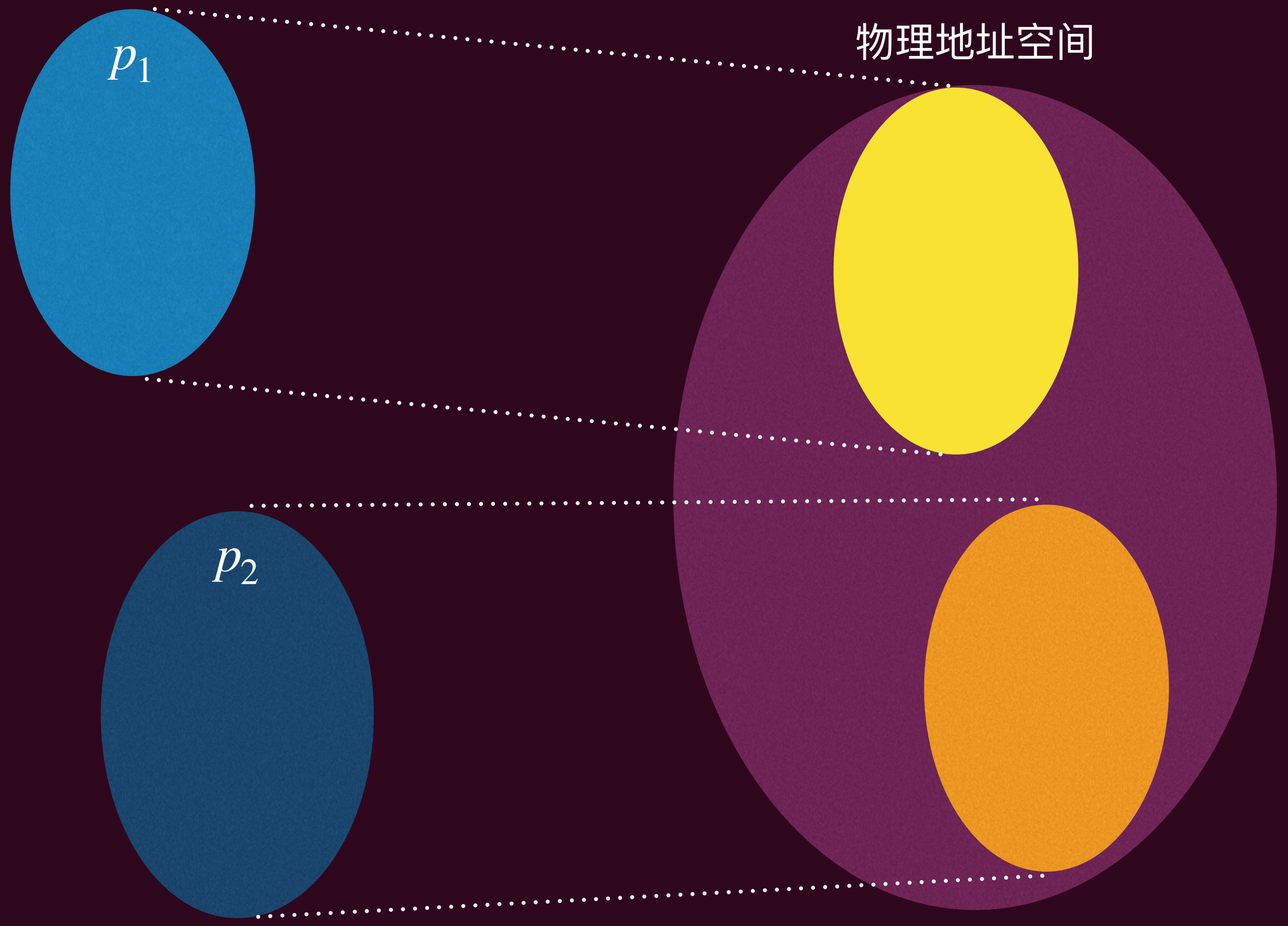
- 翻译就是一个函数f：其将〈pid, virtual address〉映射到physical address



- 优点，可以很自然地提供：
  - 保护
  - 重定位
  - 数据共享
  - 连续空间假象

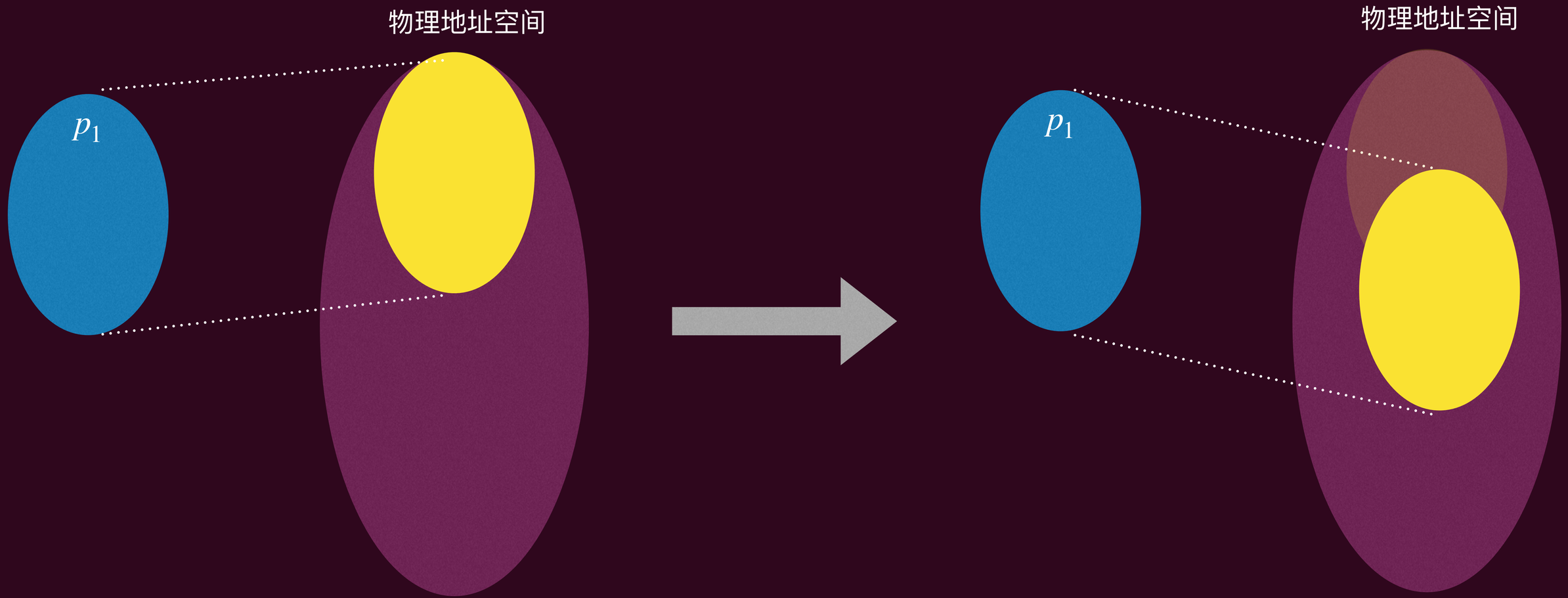
地址翻译

- 保护：让不同的进程映射到不同的区域即可（即两个进程的映射函数的值域不相交）



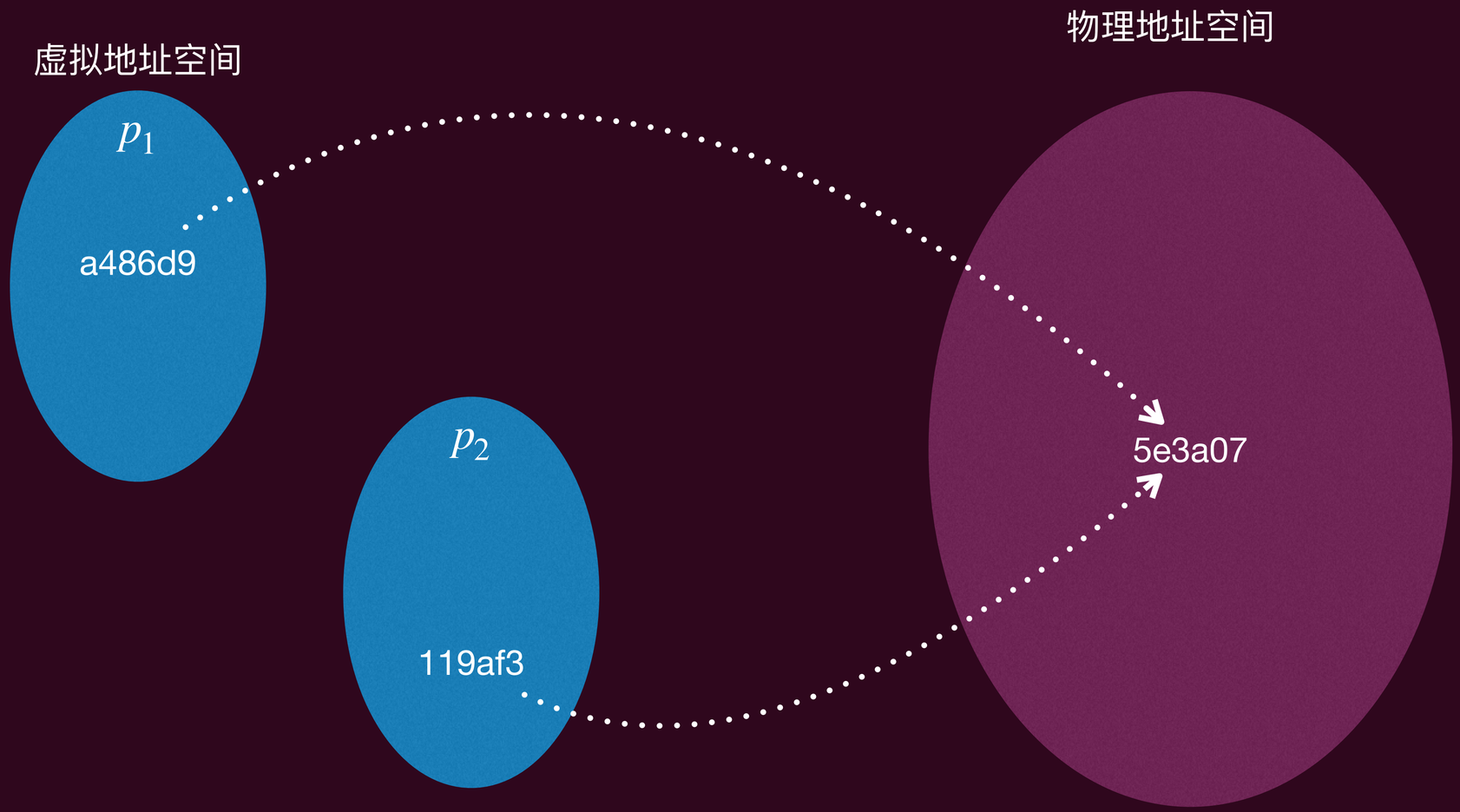
地址翻译

- 可重地位：进程被映射到的物理地址可以在运行时不断变化（运行时，不是编译时，因此是动态重定位）



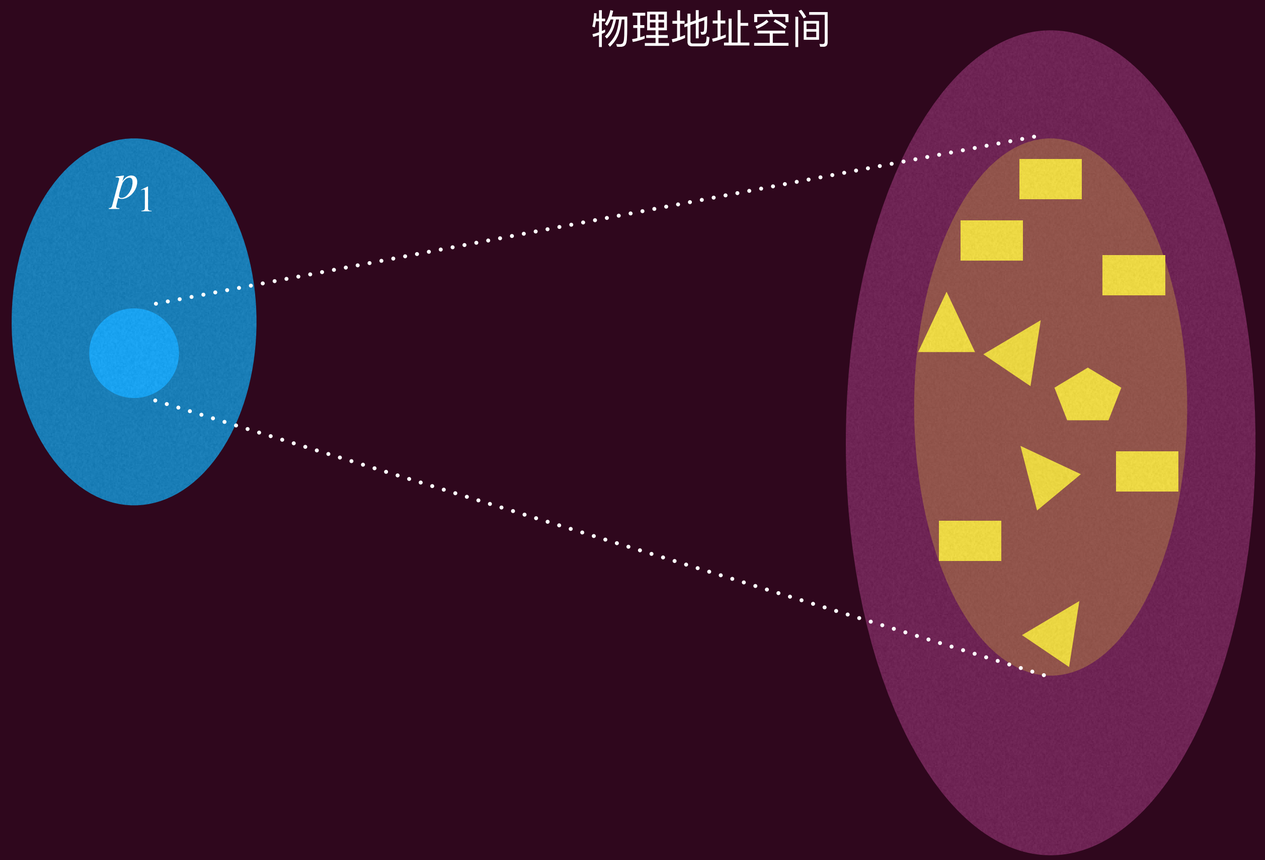
地址翻译

- 数据共享：将不同进程的不同虚拟地址映射到同一个物理地址



地址翻译

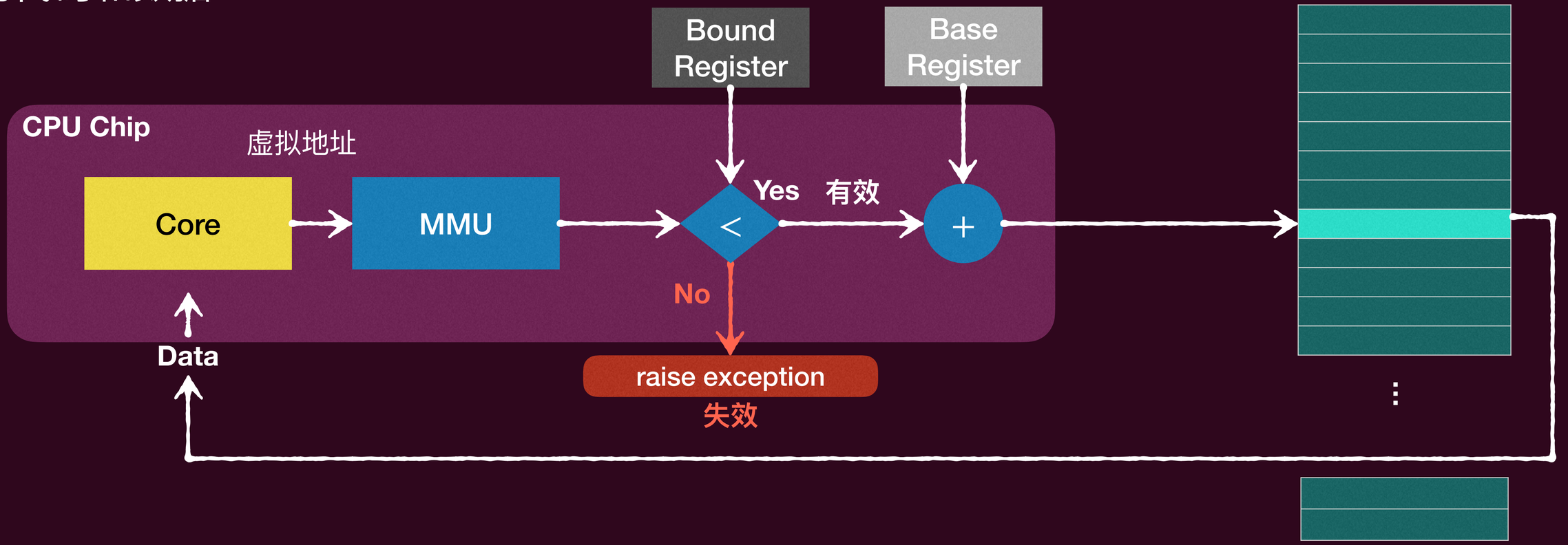
- 构建连续空间假象：虚拟地址中的连续地址空间（编程友好），映射到物理内存可以不必连续



连续内存分配

连续内存分配

- 最简单的内存分配方式，进程（包括操作系统内核）被分配一个连续的物理内存地址
- 利用基址和界限机制隔离用户进程之间的地址空间，以及防止用户进程修改操作系统的代码和数据

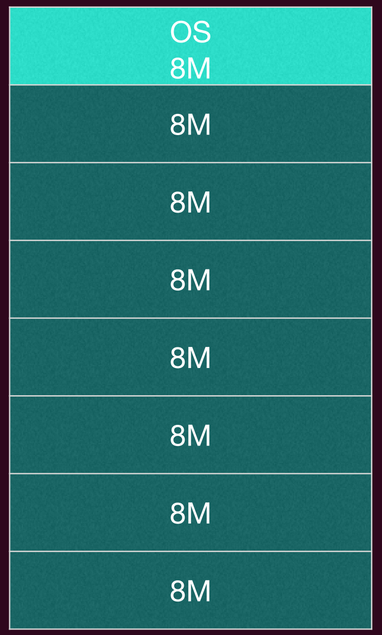


连续内存分配

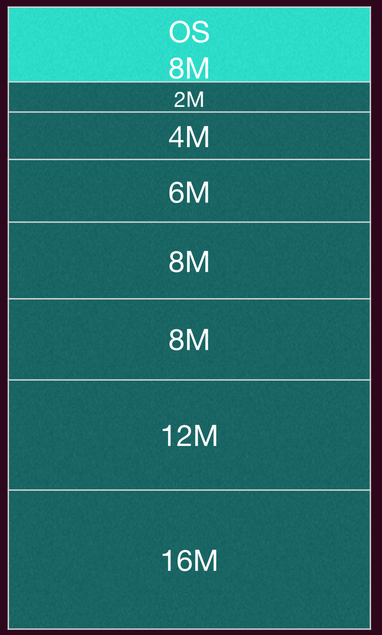
- 多个进程时需要将物理内存进行分割，每个进程占据一个连续的物理内存分区，有两种分割方式
  - 固定分区（Fixed-Partition）
    - 物理内存一开始就被分为“固定”大小的区域，各个区域的大小可以相同，也可以不同，一个进程选择一个空闲区域进行分配
  - 可变分区（Variable-Partition）
    - 物理内存区域大小和数量是可变的，根据进程的具体需要分配相应的空间
      - 需要维护一个空闲的内存集合，开始是整个巨大的空间，但随着进程的分配和回收，内存会存在很多大小不一的空闲的“孔”

固定分区

- 固定分区的方式是简单的
  - 不管是分配和回收效率都非常高



Fixed-partition：equal-size partitions



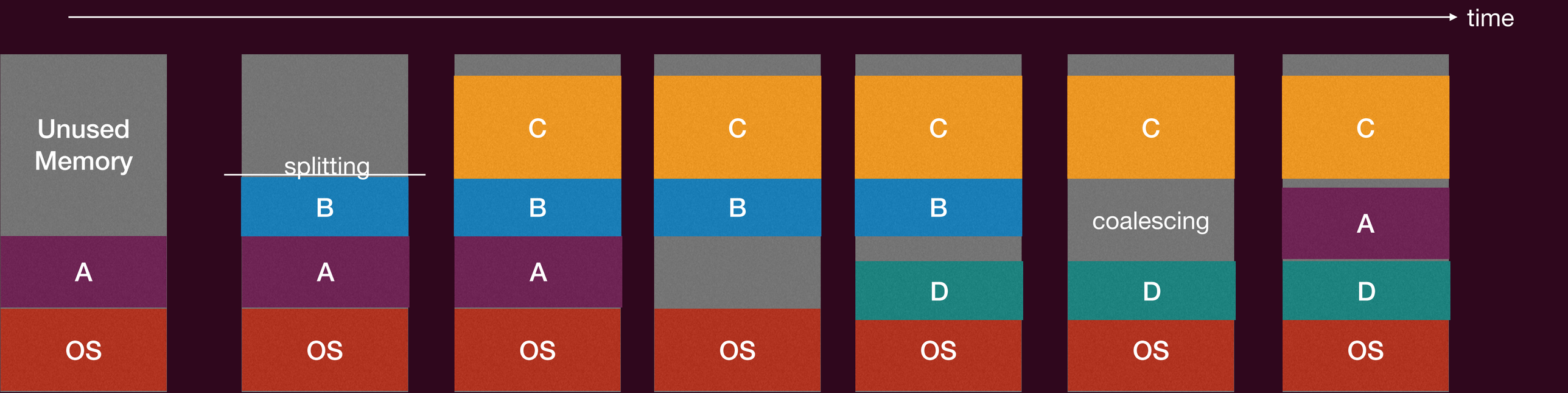
Fixed-partition：unequal-size partitions

固定分区

- 问题是：如果一个进程所需要的空间非常大，这种固定分区就不适用了
- 此外，进程的size各种各样，但只能选择一个分区存放，这个分区大于进程的部分就是一个内部碎片（internal fragmentation），其无法被其他进程占用，自己也不用
- 此外，固定的分区数也限制了可以同时放入内存的进程数

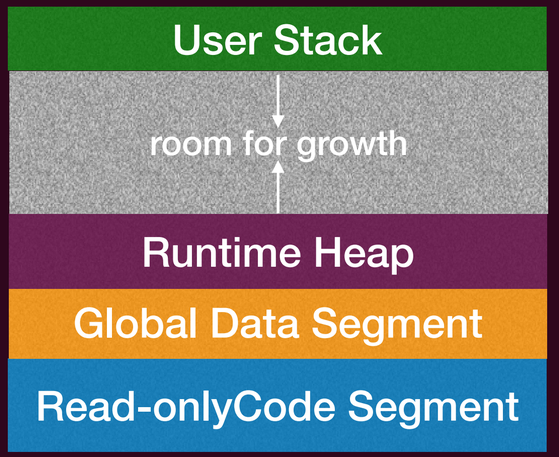
可变分区

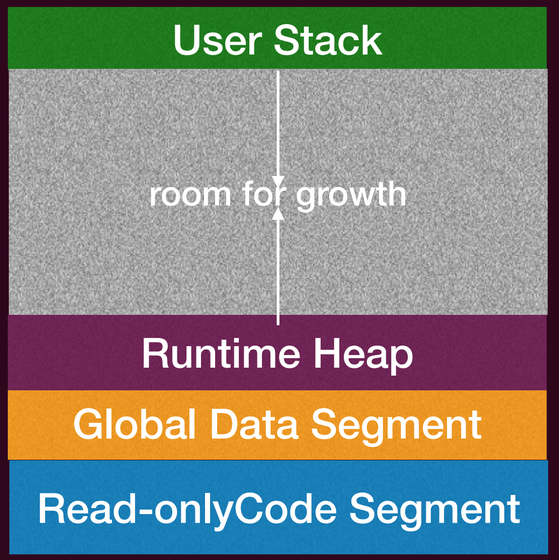
- 可以支持动态的按需分配空间
  - 当一个进程需要加载到内存时，操作系统从一个足够大的空闲块分配内存（多了的部分进行分割（splitting）为剩余的空闲块）
  - 当进程终止时释放其分区，并且与相邻的空闲分区合并（合并， coalescing）



可变分区

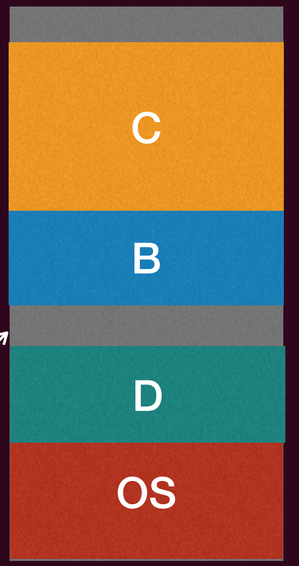
- 鉴于大多数进程在运行时会增长，在加载进程时会分配一些额外的内存。
- 此外，如果给进程所分配的区域用完，那么该进程将可能：
  - 被移动到一个有足够空间的空闲区域中
  - 被交换（Swap）出内存（至磁盘），直到能够创建一个足够大的空洞
  - 或者直接被终止（OOM Killer）



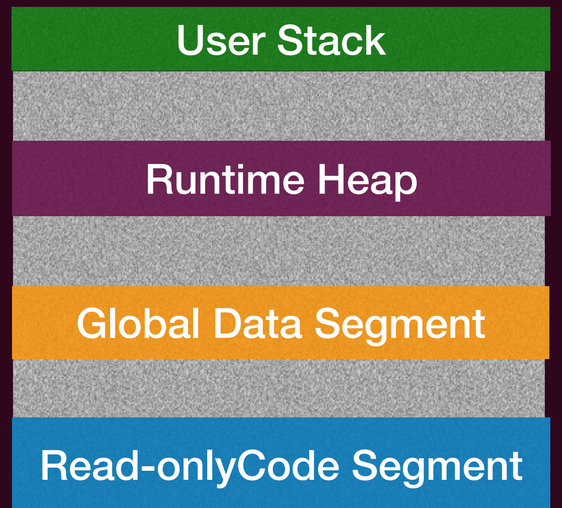


碎片

- 碎片（Fragmentation）是指无法被分配的未使用内存
  - 外部碎片（空洞）：由于分散的小的不连续的空闲空间导致的内存浪费，发生在分区之间，通常是由于进程的不断加载和释放造成
  - 内部碎片：由于分区大小和加载的进程大小之间的差异（即进程小于分配的分区）导致的内存浪费，发生在分区内



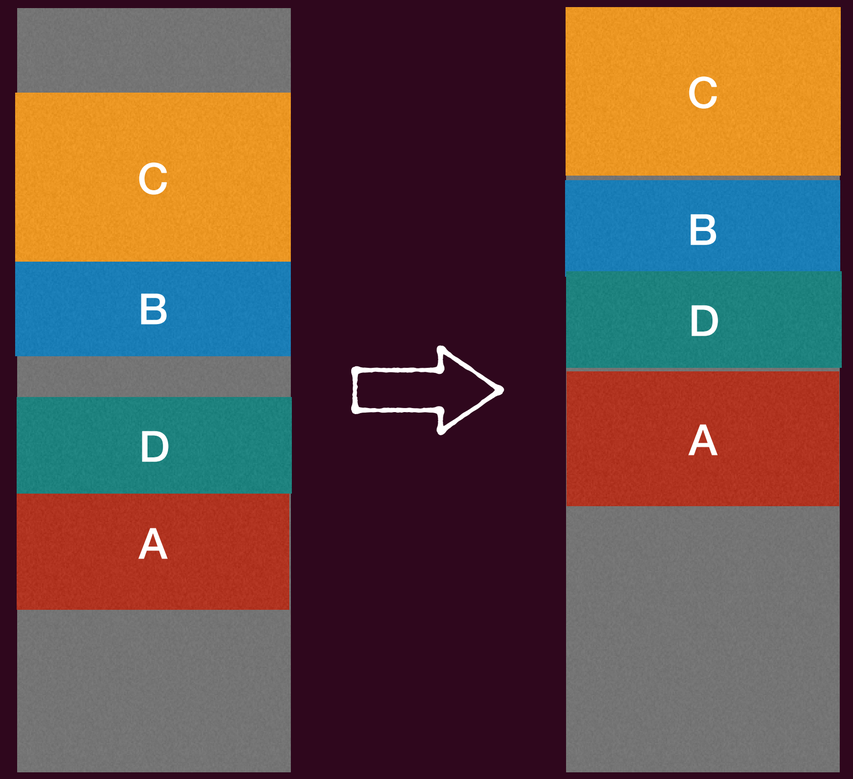
过于小，以至于无法分配给新的进程



一个分区内部的碎片，进程不用，也无法分配给其他进程了

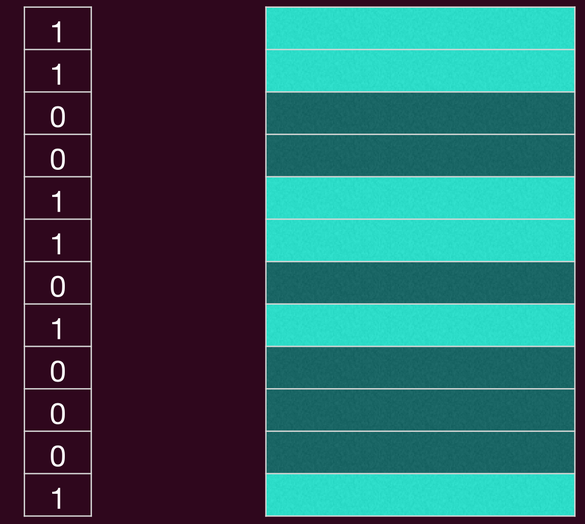
外部碎片的处理方法

- 进程终止或暂时交换出磁盘，可以释放内存，可能会合并一些碎片形成大的可分配区域
- 通过紧缩（compaction）减少外部碎片化
  - 重排内存内容以将所有空闲内存放在一起
  - 时间复杂度较高（一般在系统实在没有内存的情况下才会做，内存分配slow path的一部分）



空闲内存管理

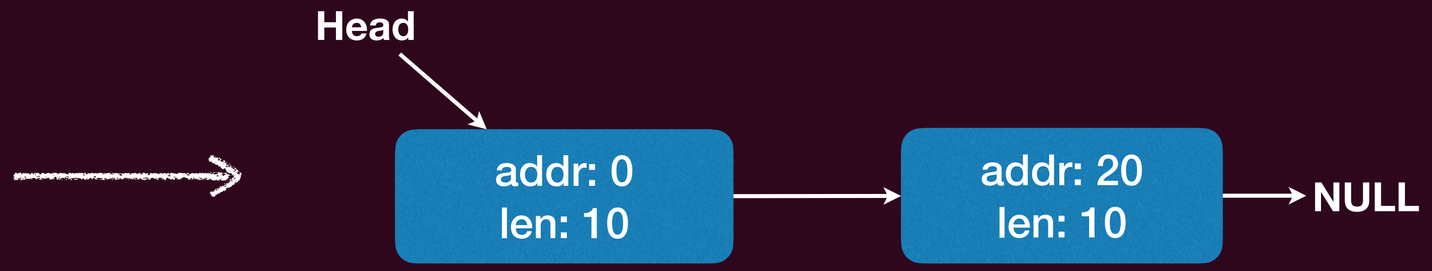
- 为了实现动态可变分区内存分配，操作系统应维护以下信息：
  - 已分配的分区
  - 空闲的分区（空洞）
- 一种简单的办法：Bitmap
  - 内存被划分为分配单元（几字节到几千字节）
  - 每个分配单元对应位图中的一位，如果该单元空闲则该位为0，如果被占用则该位为1（或反过来）



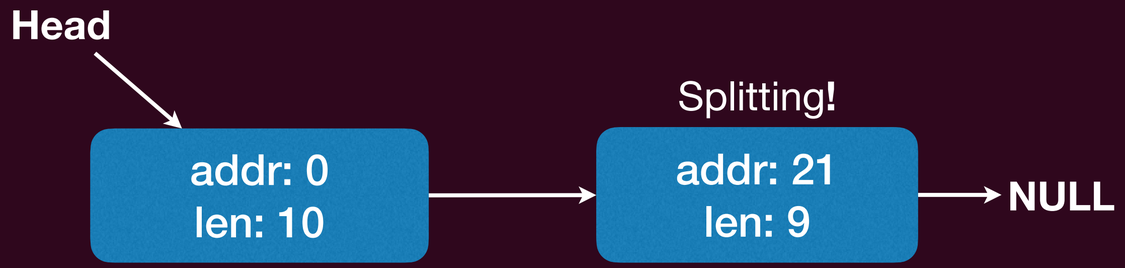
必须搜索位图以找到k个连续的0位来加载一个k单元的进程。

空闲内存管理

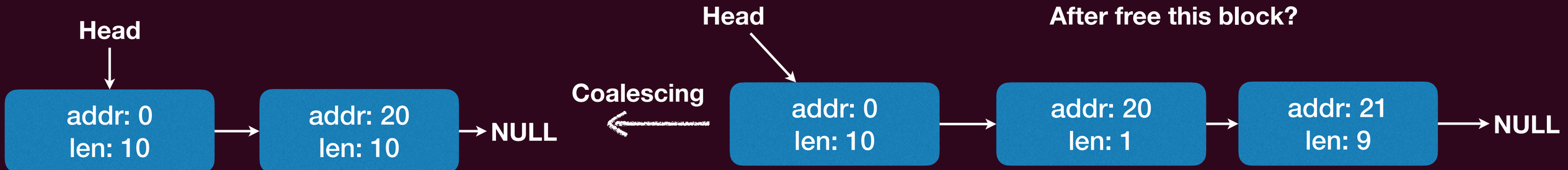
- 一个更加常用的做法：空闲内存链表



After allocating one byte from the second free block:



After free this block?



空闲内存管理

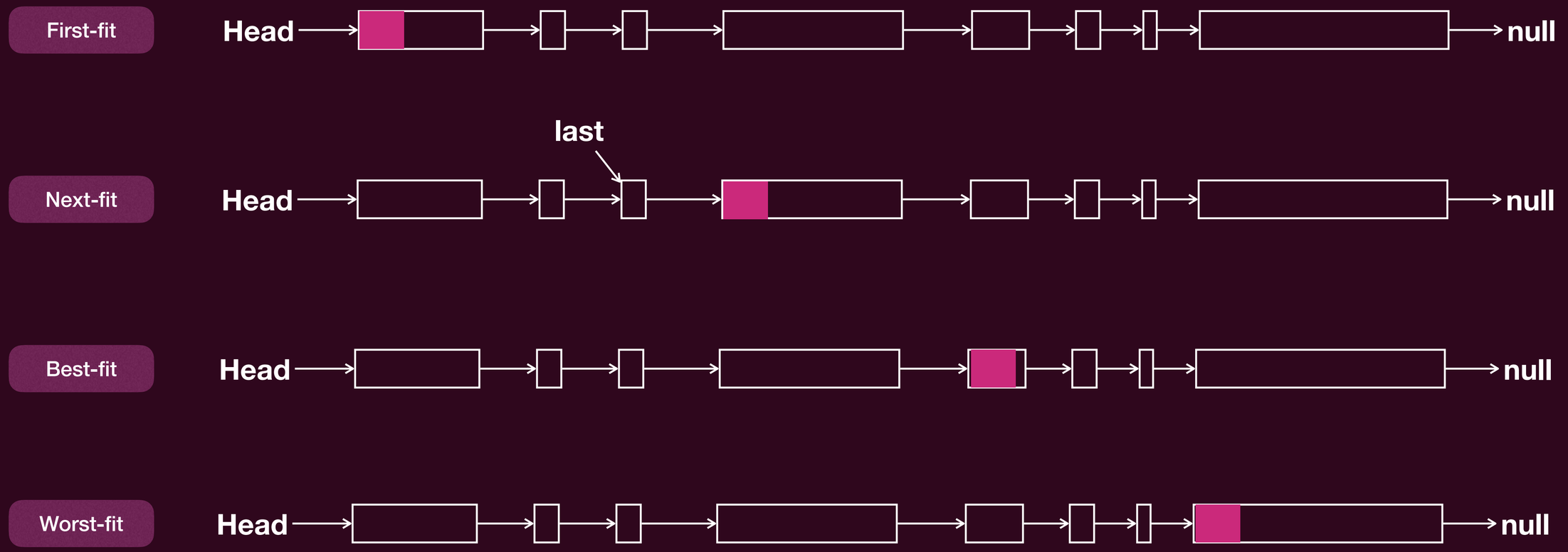
- 我门还需要跟踪已分配区域的大小
  - 这样free(*p)时才能正确的返回一个正确的空闲结点
- 方法：每次多malloc一点空间记录元信息

基本分配策略

- 空闲内存列表有很多都可以分配给一个内存的申请需求，只要大于等于申请的空间即可，那么选择哪个呢?
- 不同的策略会影响分配和回收的性能和有效性（比如碎片数量）
  - Best-fit：分配最小的足够大的空闲区域（尽可能物尽其用），需要遍历整个列表
    - 但可能导致产生微小且无用的外部碎片
  - Worst-fit：分配最大的空闲区域
    - 剩余的部分最大，可以为其他进程所占用，而不是形成碎片，但同样遍历整个列表
  - First-fit：分配第一个足够大的空闲区域（尽可能少搜索）
  - Next-fit：跟踪上次适配的位置，并从上次搜索结束的地方开始搜索 （尽可能均匀的搜索整个空间）

基本分配策略

- 不同的workload下性能各异
  - 现实中workload? Mimalloc: free list sharding in action (APLAS'19)



性能还是问题

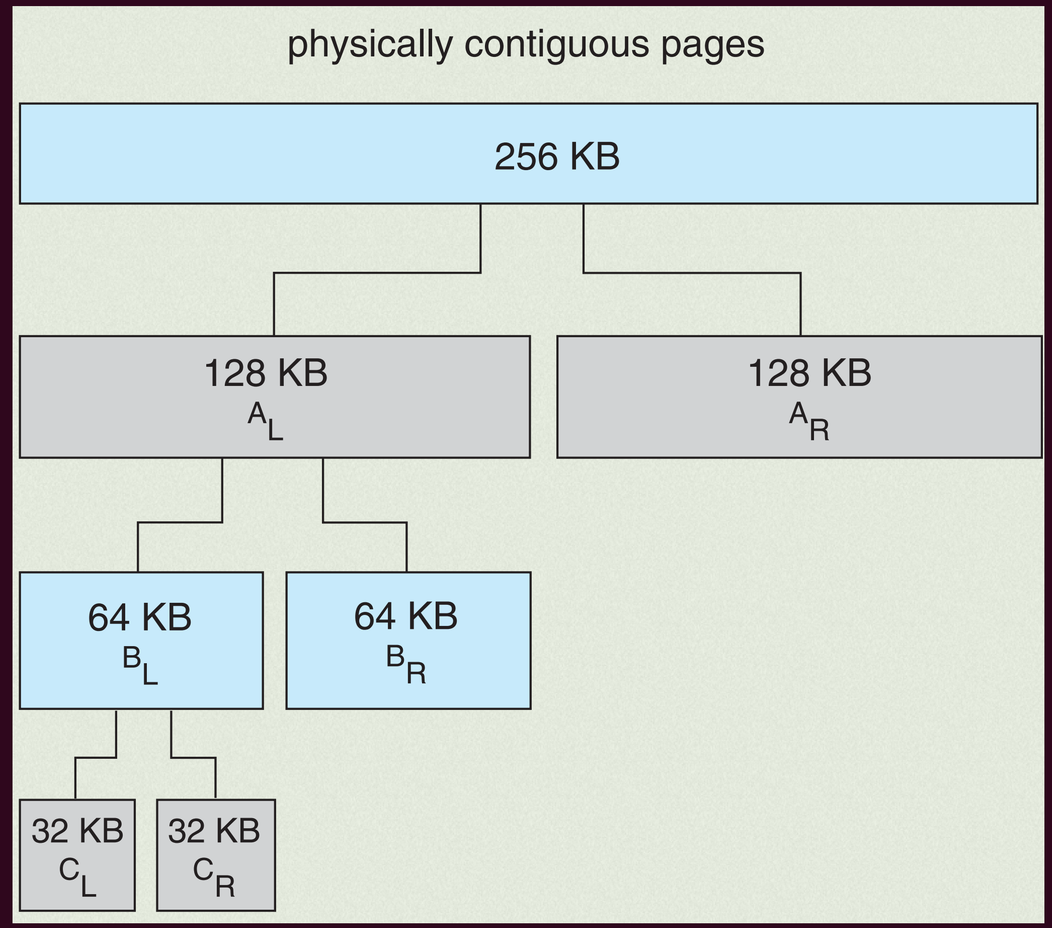
- 链表找还是太费力了
  - 其搜索的算法复杂度是$O(n)$
- 更好的数据结构
  - 比如：红黑树，结点存放的空闲内存的大小，这样就可以以$\lg n$的复杂度搜索相应的内存
- 但还有一个问题，当free之后，如何合并?
  - 在地址上，相邻的空闲块才能合并，因此空闲的结点链表应该按照地址进行链接，然后free的时候扫描整个链表
  - 有没有更加方便的合并?

伙伴系统（Buddy System）

- 从包含物理上连续页面的固定大小段中分配内存
  - 使用2的幂分配器power-of-2 allocator 分配内存
    - 按2的幂大小单位满足请求：请求向上舍入到下一个最高的2的幂
    - 当需要比可用空间更小的分配时，当前块被分割为两个下一个较低2的幂的伙伴
    - 持续进行直到有适当大小的块可用

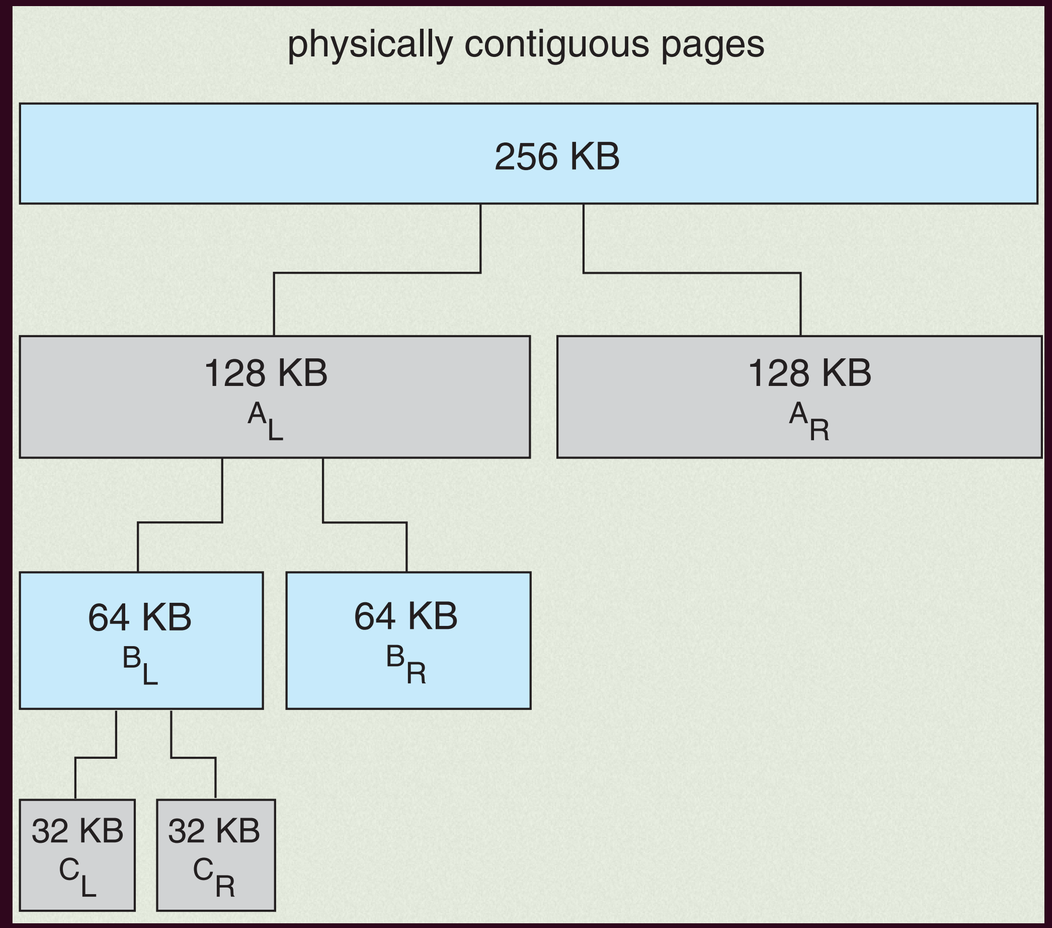
伙伴系统（Buddy System）

- 例如，假设有一个256KB的可用块，请求21KB
  - 分割为两个大小为128KB的A_L和A_R
  - 其中一个进一步分割为两个大小为64KB的B_L和B_R。
  - 再进一步分割为两个大小为32KB的C_L和C_R： 其中一个用于满足请求



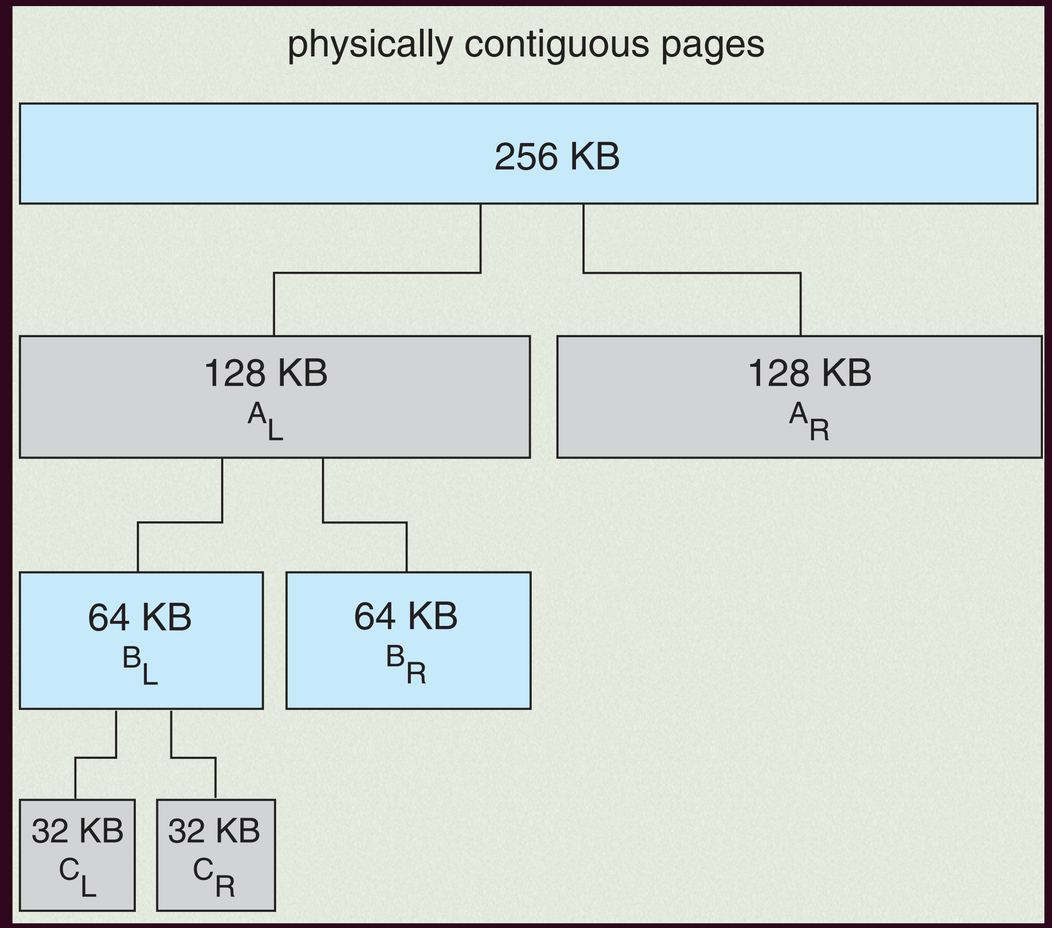
伙伴系统（Buddy System）

- 伙伴系统的好处在于合并效率高：
  - 比如如果将这个21KB的块归还给空闲列表
    - 分配程序会检查“伙伴”32KB是否空闲。
    - 如果是，就合二为一，变成64KB的块。然后会检查这个64KB块的伙伴是否空闲，如果是，就合并这两块。
    - 这个递归合并过程继续上溯，直到合并整个内存区域，或者某一个块的伙伴还没有被释放



伙伴系统（Buddy System）

- 伙伴的位置非常容易得到（其地址和伙伴只有一位不同，而正是这一位决定了它们在整个伙伴树中的层次）
  - 大小为2^k的块的地址是2^k的倍数（右边有k个零）
  - 比如一个2^5的地址: xxx...xx00000
  - 分割之后, 两个伙伴块大小为2^4，地址分别为：
    - xxx...xx0000
    - xxx...xx1000



伙伴系统（Buddy System）

- Buddy System已经是linux内部分配“大”粒度的连续的物理页面的方法了
- 但还是存在一些问题：
  - 内部碎片
    - 只允许分配2的整数次幂大小的空闲块（实际上，伙伴系统分配的最小单位是4K，即一个物理页），因此如果不是2的整数次幂的大小，会有内部碎片
    - 比如刚刚的例子中有 32K-21K无法使用（属于该进程，但该进程没有使用他们，也不能被其他进程使用）

Segregated List (Slab)分配器

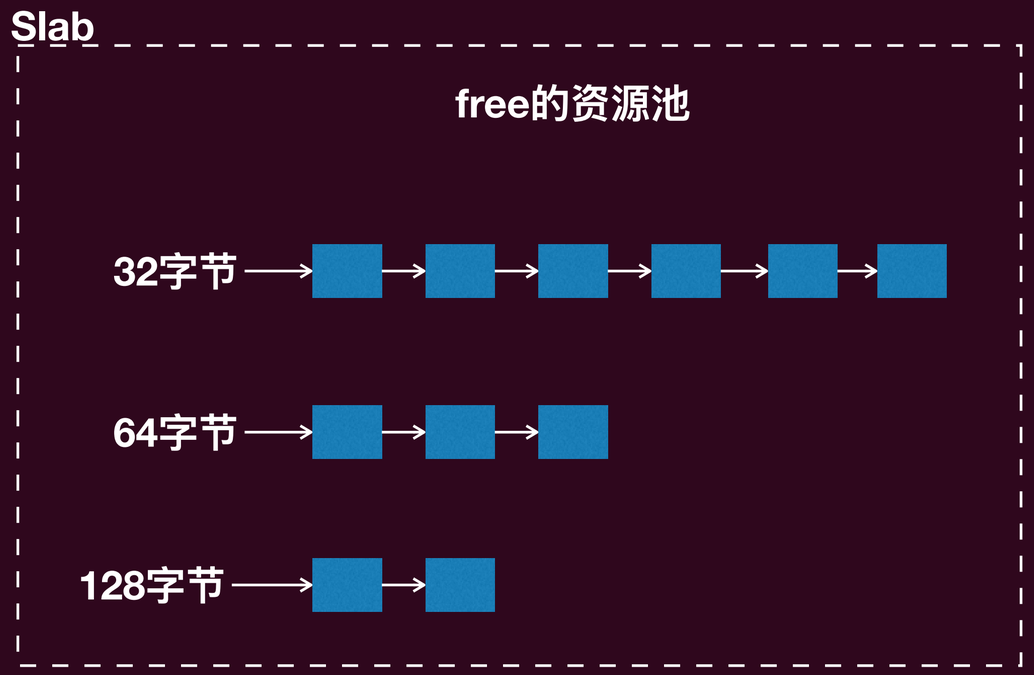
- 操作系统里面的结构体大小常为几十、几百字节
  - Buddy System来分配的话会产生大量的碎片
- 此外，一个经验观察：
  - 系统频繁分配的对象大小相对比较较小且固定
    - 小对象分配/回收的 scalability 是System内存分配的主要瓶颈

Slab分配器

- 目标:快速分配小内存对象
- 解决方案：SLAB （上世纪 90 年代， Jeff Bonwick在Solaris 2.4中首创SLAB，Christoph Lameter在07年左右在Linux中给出了优化版本SLUB，并在Linux-2.6.23及之后的版本中，成为默认分配器）
  - 从伙伴系统获得大块内存
  - 进一步细分成**固定**大小的小块内存进行管理
  - 块大小通常是2^n个字节(一般来说，$3 \leq n \leq 12$)
    - 可以额外增加特殊大小(一些频繁使用的特定数据结构大小)如198字节从而减小内部碎片

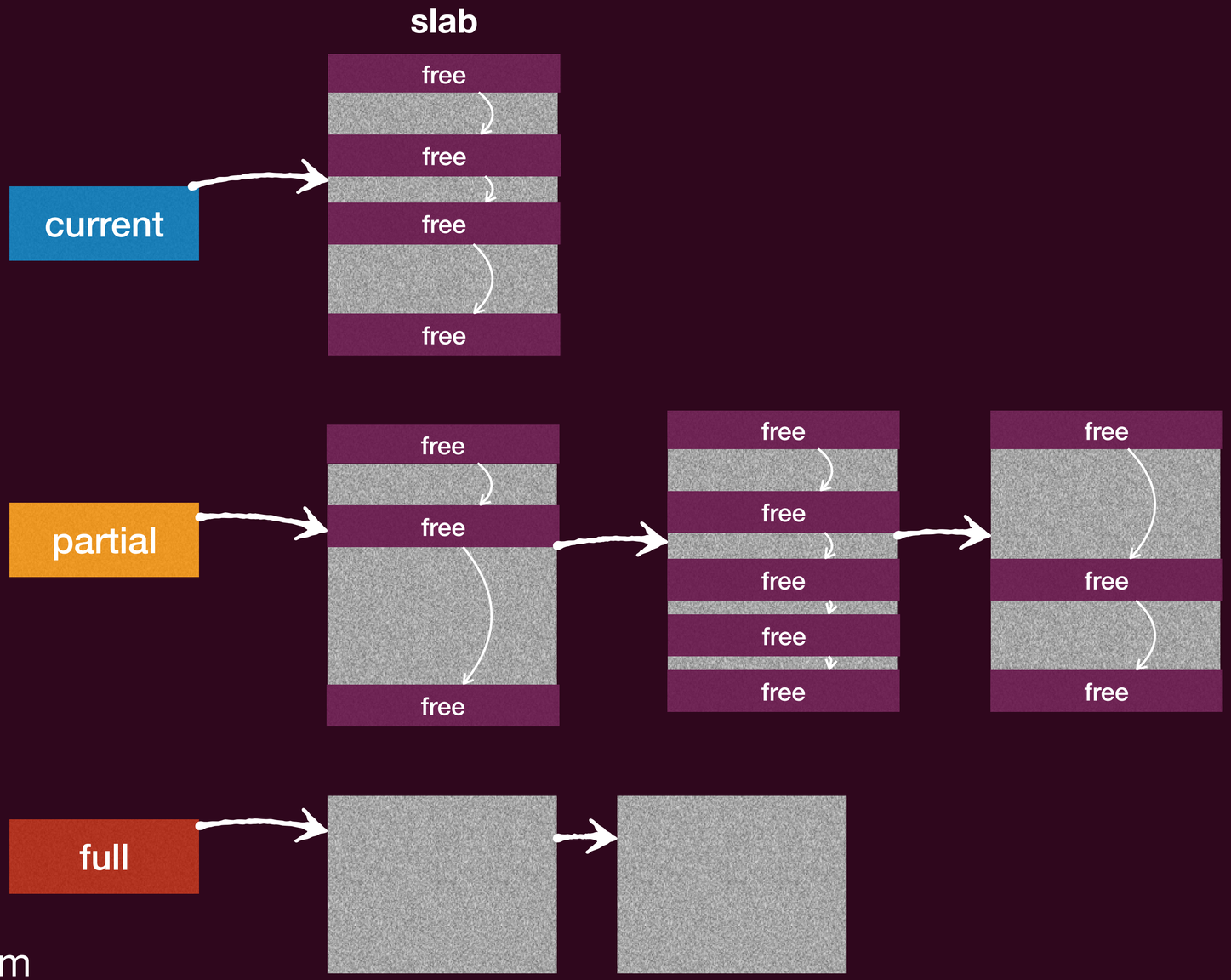
Slab分配器

- 只分配固定大小块
  - 对于每个固定块大小，Slab分配器都会使用独立的内存资源池进行分配
  - 采用best fit定位资源池



Slab分配器

- 三个指针：
  - current仅指向一个slab
  - partial指向未满slab链表
  - full指向全满slab链表
- 分配使用current slab (per-cpu)
  - 若满：移动到full slab
  - 从partial里申请一个
- 释放时放到对应的slab
  - 若是full，移到partial
  - 如果partial全是free，则还给buddy system



多线程下

- Fast path
  - Per CPU从当前slab中取出一个适合大小的块（快速，没有和其他CPU的竞争）
- low path
  - 需要从一个全局的partial里去找一个作为当前的slab，不巧的话（paritial里也没空闲内存），甚至需要从buddy system里重新分配连续空间，再次分割为可用的slab

Linux下的Slab

- Linux中所有的SLAB信息记录在 /proc/slabinfo 文件中
  - 试试sudo cat /proc/slabinfo
- `kmalloc` 是内核运行时申请内存的通用slab, 内核可能申请任意大小的内存，为了满足各种应用场景，内核预备了各种大小的 kmalloc slab, 从最小的8Byte一直到最大的8KB

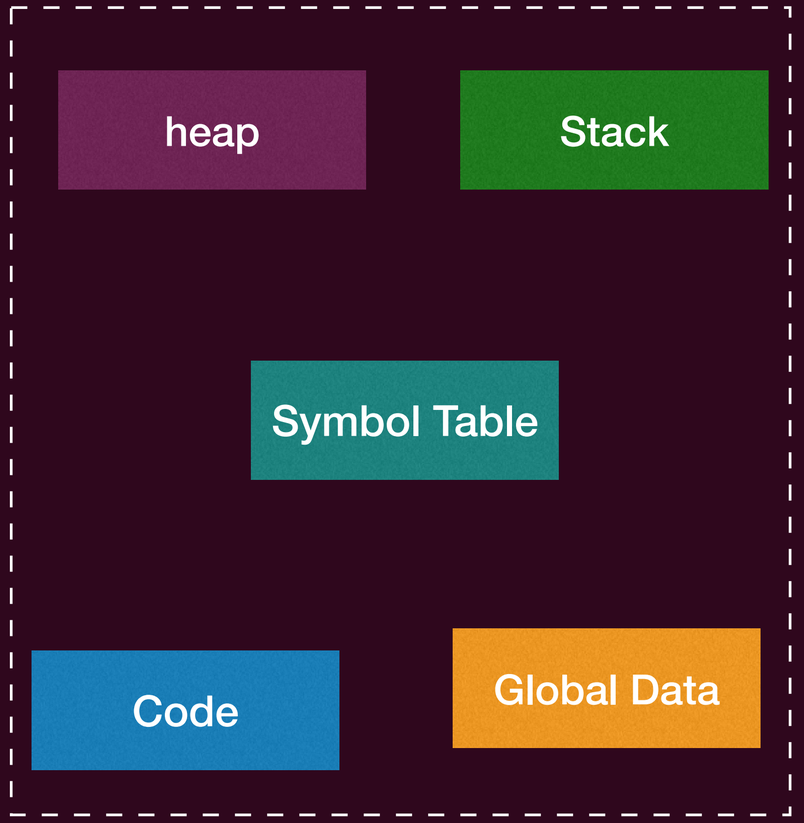
连续内存分配的问题

- 如果直接分配给**进程**内存的话，stack和heap中间的部分都被浪费了
  - 内部碎片无法避免
- 无法和其他进程共享内存（比如代码和glibc）
  - 主要是保护机制粗糙，整体空间的保护，没有精细到具体的部分内存

分段（Segment）

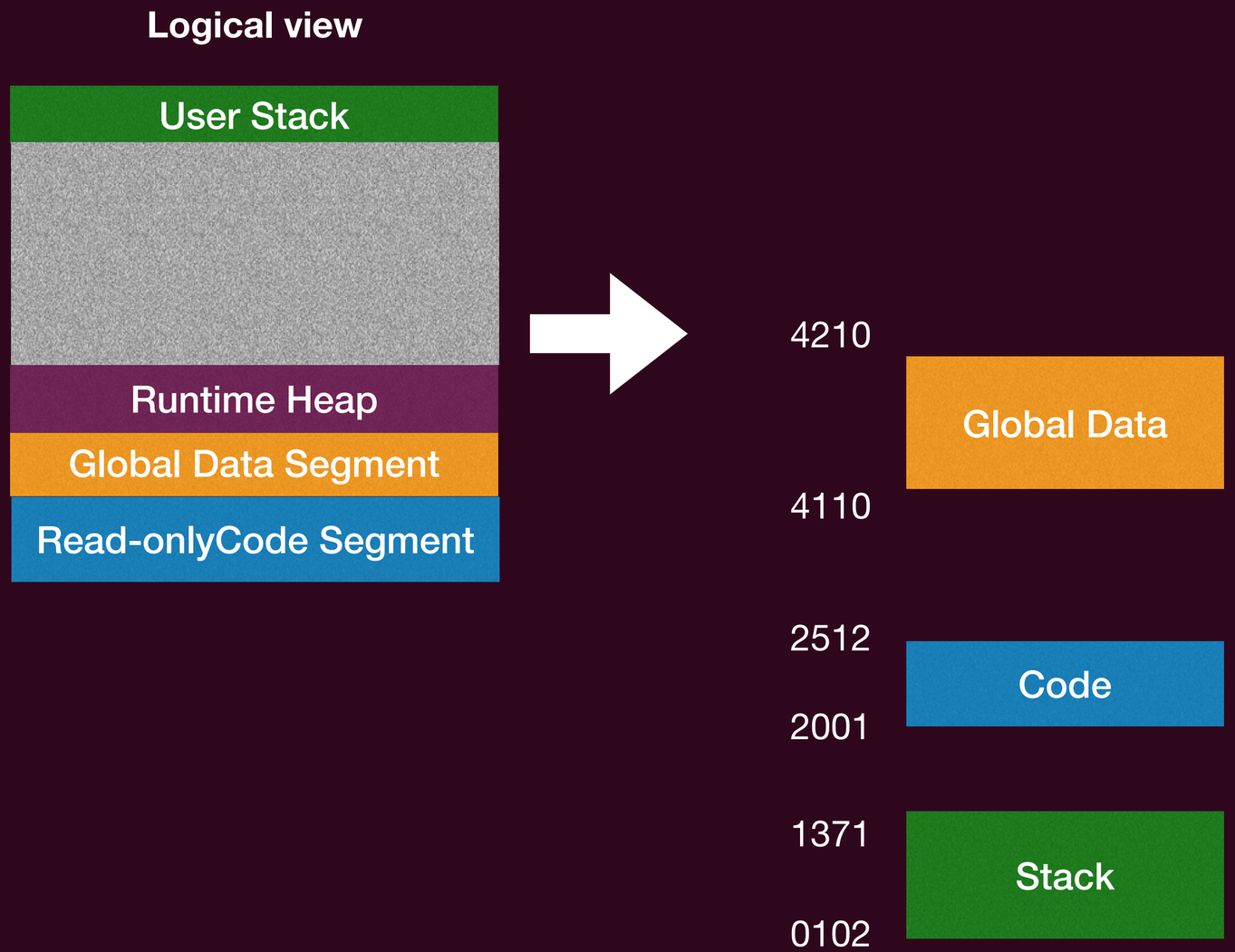
分段

- 用户视角的内存管理
  - 将程序视为一组段（segments）
  - 段是虚拟内存空间的连续区域是一个逻辑单元
    - 例如代码段、栈、堆等
    - 按照这样的段分配内存



分段

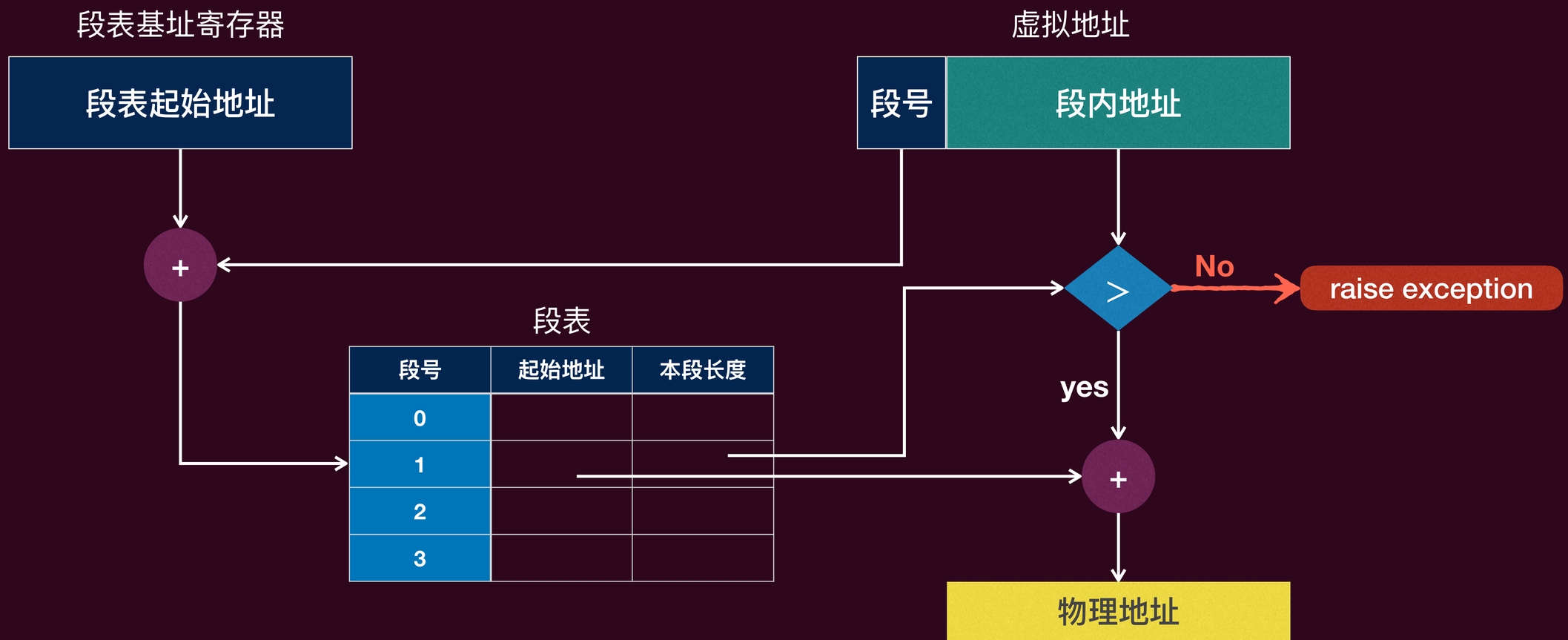
- 每个段独立映射到物理内存中的一组连续地址
  - 没有特定的顺序
  - 不需要映射未使用的虚拟地址
    - 可以消除内部碎片
  - 不同的段可以独立增长或缩减



分段底层机制

- 虚拟地址空间分成若干个不同大小的段
  - 段表（每个进程对应一个）存储着每个分段的信息（由段号索引），可供MMU查询
    - 段基址（Segment Base）：段在内存中所在的起始物理地址
    - 段界限（Segment Bound）：段的大小
  - 虚拟地址分为: 段号(segment number) + 段内地址(偏移, Offset)
- 物理内存也是以段为单位进行分配
  - 虚拟地址空间中相邻的段，对应的物理内存可以不相邻

分段机制

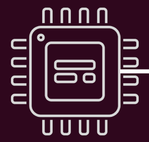

段表基址寄存器
段表起始地址
虚拟地址
段号
段内地址
+
>
No
raise exception
yes
段表
段号
起始地址
本段长度
0
1
2
3
+
物理地址

支持共享

- 多个虚拟地址空间可以映射到内存中的同一物理段
  - 比如：只读代码的一个副本在进程间共享，一个可执行文件被加载多次
  - 也可以用作进程间通信
- 段表需要增加保护位
  - 指示程序是否具有相应段的读/写/执行权限
- 每个进程仍然认为它在访问自己的私有内存（透明性）

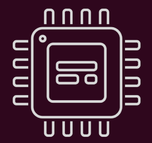
支持共享

Process 1's view



Virtual address
0x0500

Heap
Global Data
Code
Stack



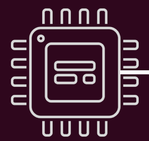
段号	段内地址
0	500

段表

段号	起始地址	本段长度
0		
1		
2		
3		

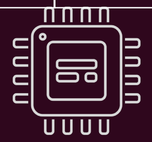
+

Process 2's view



Virtual address
0x0500

Heap
Global Data
Code
Stack



段号	段内地址
0	500

段表

段号	起始地址	本段长度
0		
1		
2		
3		

+

Physical view

P2's Heap
P2's Global Data
P1's Stack
P1's Global Data
P1's and P2's Code
P2's Stack
P1's Heap

OS对段表机制的支持

- 操作系统应在上下文切换时保存和恢复段表（指向段表的寄存器）
- 当段增长或缩小时，操作系统应进行交互（更新段表）
- 创建新的地址空间时，操作系统应在物理内存中为其段找到空间
  - 操作系统维护着空闲内存块
- 由于段的长度各不相同，内存分配是一个动态内存分配问题
  - 需要分割和合并

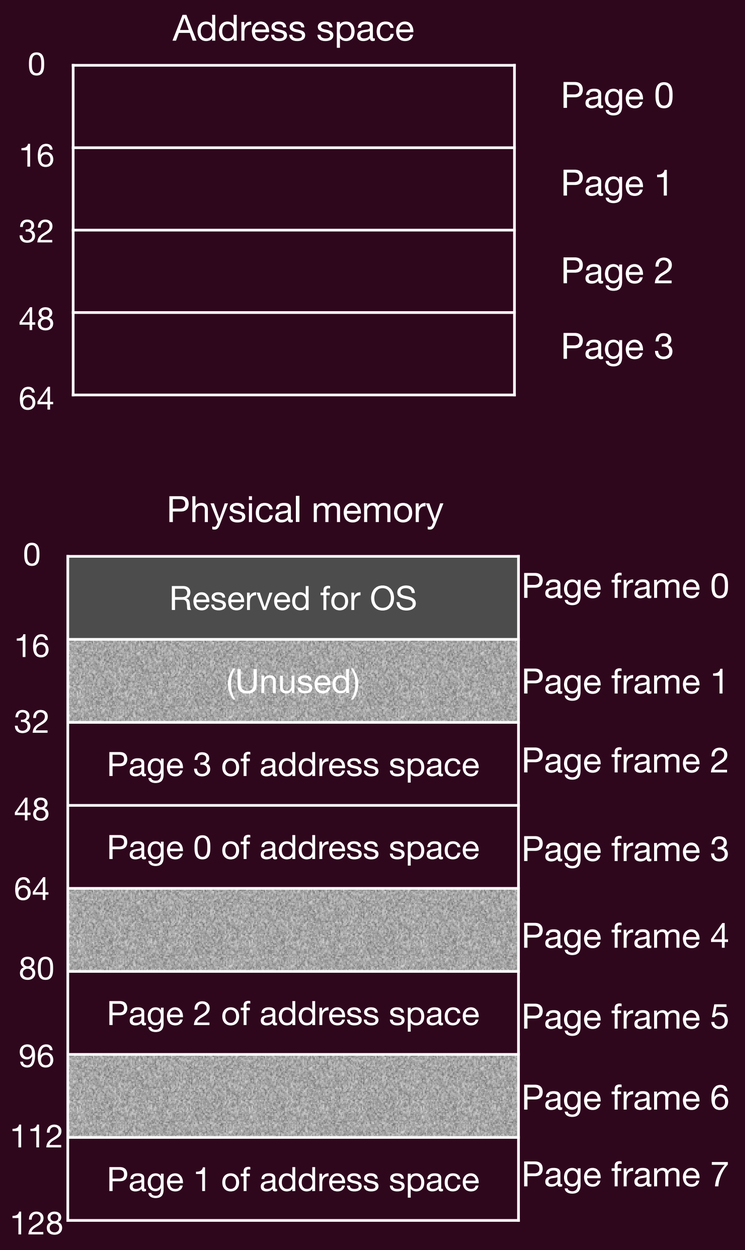
分段机制

- 存在问题
  - 分配的粒度太粗，随着时间的推移会产生外部碎片
  - 如果是一个大但使用稀疏的堆?
  - 如果地址空间的使用模型与段设计不匹配怎么办?

分页（Paging）

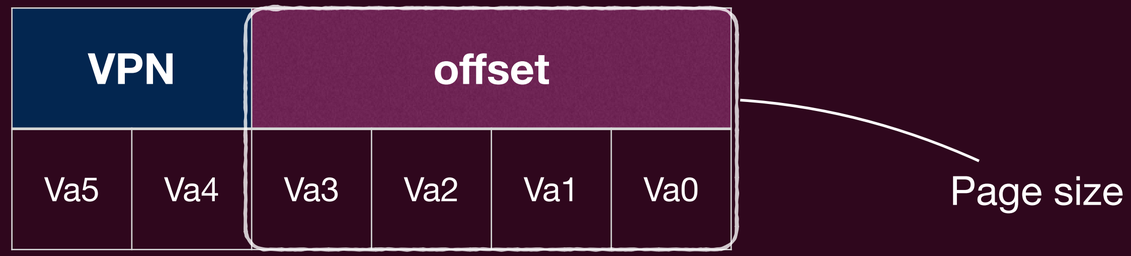
分页机制

- 更细粒度的内存管理
  - 物理内存被划分成连续的、等长的物理页（也叫帧frame）
    - 大小一般为2的幂，比如默认是4KB （2^{12}B的size）
  - 虚拟内存也被划分为相同大小虚拟块—虚拟页（Page）
  - 任意虚拟页可以映射到任意物理页
    - 非常灵活，不需要是连续的空间
  - 由于都是按照页为单位分配内存
    - 没有外部碎片



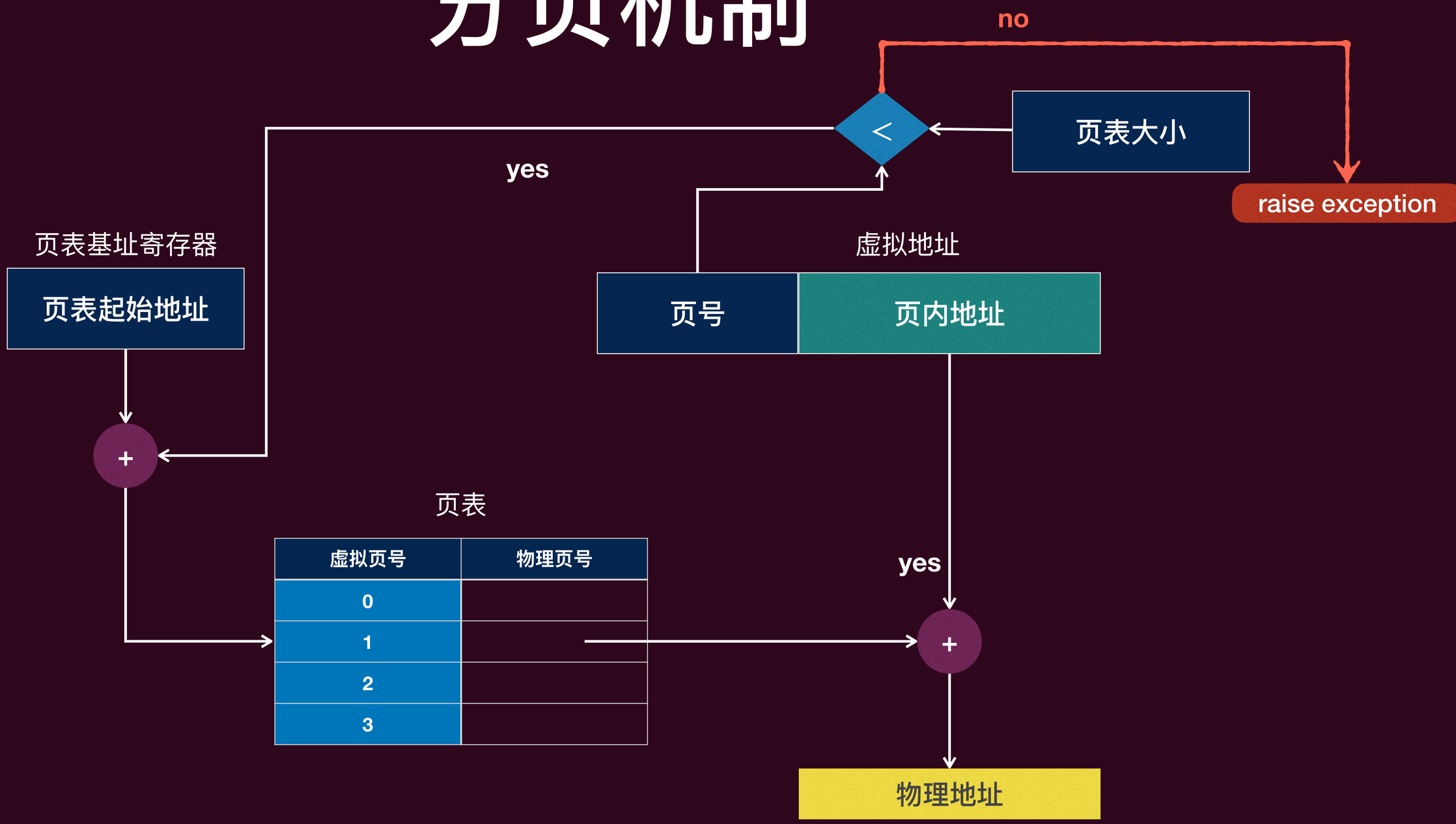
分页机制

- 虚拟地址分为：
  - 虚拟页号（Virtual Page Number, VPN） + 页内偏移



- **每个进程**都有一个页表（Page Table），每个页表项（Page Table Entry, PTE）包含一个物理页号(Physical Page Number, PPN)，也叫物理帧号（Physical Frame Number, PFN）指示每个页在物理内存中的基地址。

分页机制


no
<
页表大小
raise exception
yes
页表基址寄存器
页表起始地址
虚拟地址
页号
页内地址
+
页表
虚拟页号
物理页号
0
1
2
3
yes
+
物理地址

页表使能(Enabling)

- CPU启动流程
  - 上电后默认进入物理寻址模式
  - 系统软件配置控制寄存器，使能页表，进入虚拟寻址模式
- x86_64
  - CR0，第31位(PG位)置1，使能页表

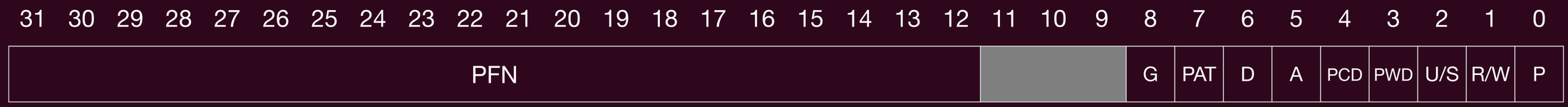
页表

- 页表最简单的形式称为线性页表（一个数组），存储在物理内存中
- 页表项（PTE）的具体布局高度依赖于机器
  - 需要足够的位来标识物理页
  - 应包括一些控制位

页表

- 页表项中的常见控制位
  - 有效位（Valid bit）：转换是否有效（支持稀疏空间）
  - 存在位（Present bit）：页面是否实际存储在内存中
  - 保护位（Protection bits）：页面是否可以被读取、写入或执行
  - 引用位（Reference bit）：页面是否已被访问
  - 脏（修改）位（Dirty/modified bit）：页面自被载入内存以来是否已被修改

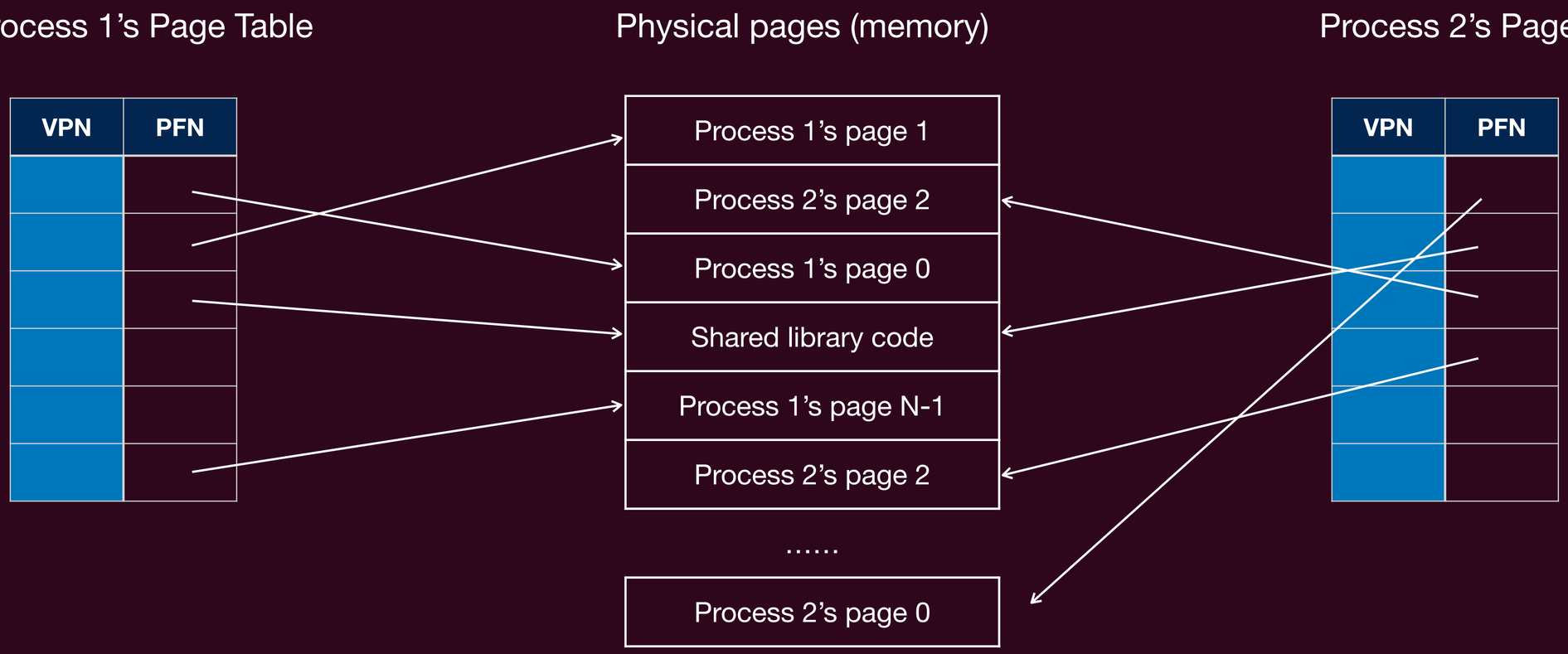
一个x86的PTE



- 物理页帧号 (PFN)
- 存在位 (P)
- 读写位 (R/W)
- 用户/主管位 (U/S)：用户模式进程是否可以访问该页面
- 引用和脏位 (A 和 D)
- 缓存相关位 (PWT, PCD, PAT 和 G)：这些页面的硬件缓存工作方式

支持共享

- 共享相同的物理页面：在两个页表中的条目指向相同的页帧（带有某些保护位）。

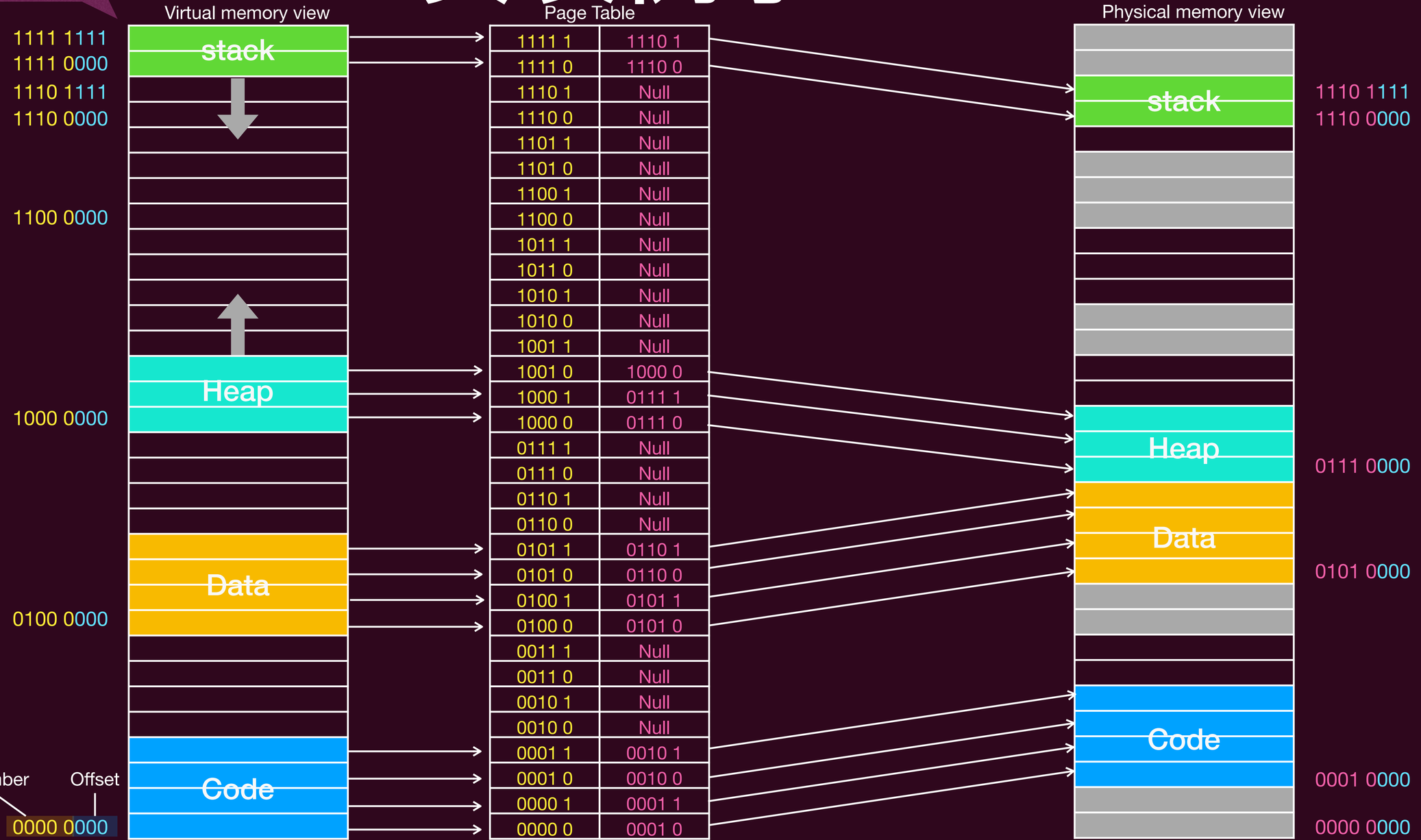


回顾Copy-on-Write

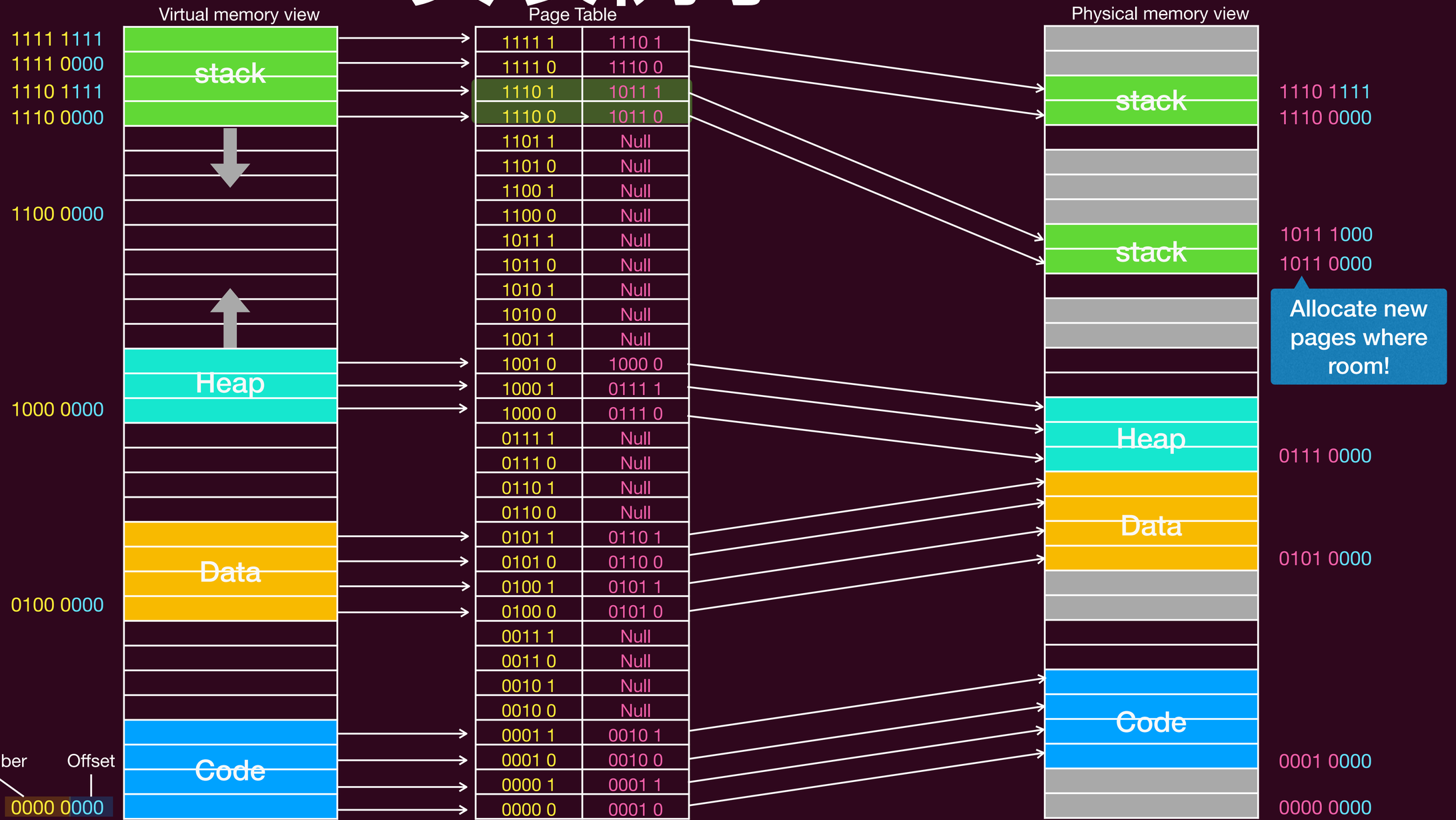
- Copy-On-Write: 尽可能共享，并仅在需要时创建自己的副本。
  - 例如，fork()通过完全复制父进程来创建一个新进程：
  - 父进程和子进程是相同的，除了fork的返回值（共享大量内存）
  - 复制父进程的页表
  - 所有共享的页面都标记为“只读”
  - 如果任何一方修改了页面中的数据，就会引发异常，并创建该页面的完整内存副本。

页表例子

What happens if stack grows to 1110 0000?



页表例子


Virtual memory view
Page Table
Physical memory view
1111 1111
1111 0000
1110 1111
1110 0000
1100 0000
1000 0000
0100 0000
stack
Heap
Data
Code
1111 1 1110 1
1111 0 1110 0
1110 1 1011 1
1110 0 1011 0
1101 1 Null
1101 0 Null
1100 1 Null
1100 0 Null
1011 1 Null
1011 0 Null
1010 1 Null
1010 0 Null
1001 1 Null
1001 0 1000 0
1000 1 0111 1
1000 0 0111 0
0111 1 Null
0111 0 Null
0110 1 Null
0110 0 Null
0101 1 0110 1
0101 0 0110 0
0100 1 0101 1
0100 0 0101 0
0011 1 Null
0011 0 Null
0010 1 Null
0010 0 Null
0001 1 0010 1
0001 0 0010 0
0000 1 0001 1
0000 0 0001 0
stack
stack
Heap
Data
Code
1110 1111
1110 0000
1011 1000
1011 0000
0111 0000
0101 0000
0001 0000
0000 0000
Allocate new pages where room!
Page number
Offset
0000 0000

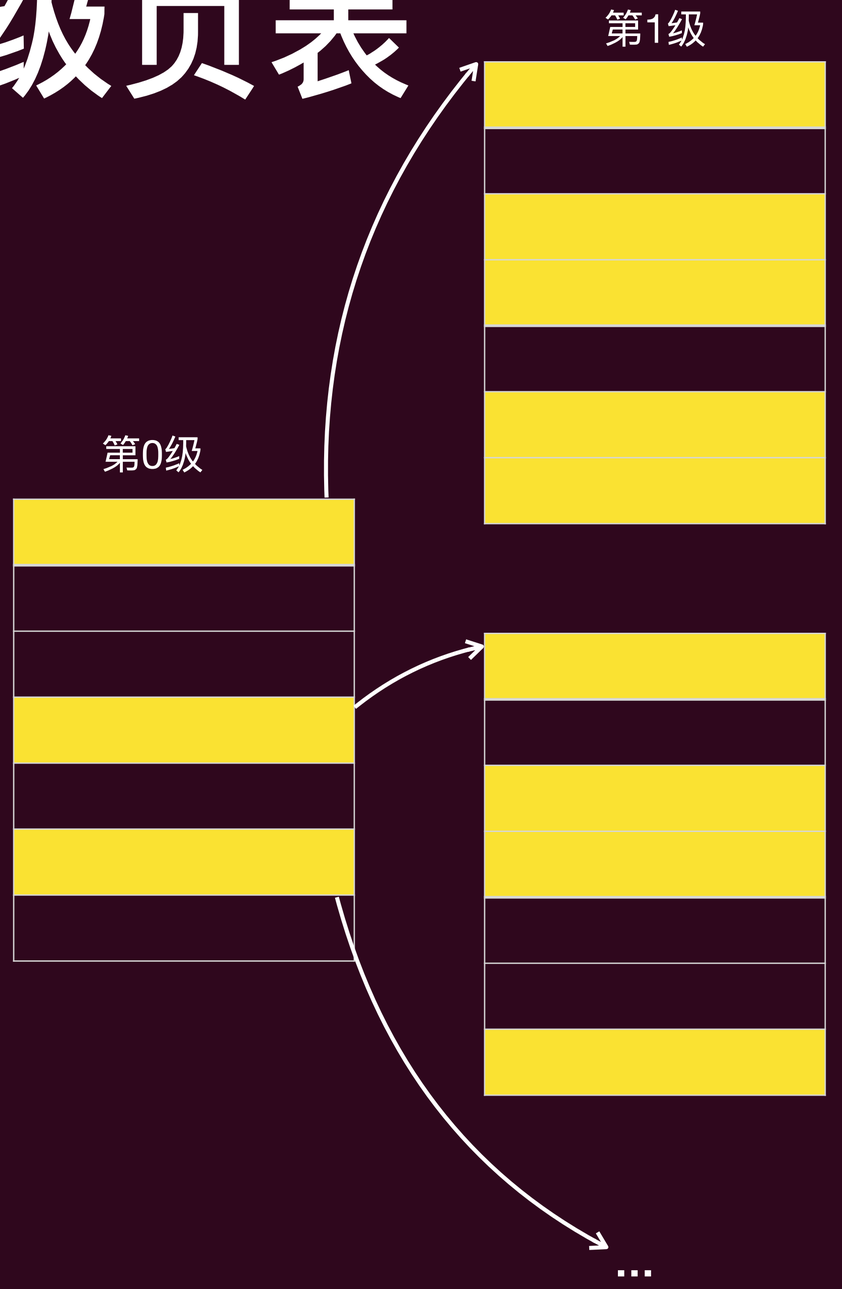
单级页表的问题

- 若使用单级页表结构，一个页表有多大?
  - 32位地址空间，页4K，页表项4B，
    - 页表大小: $\frac{2^{32}}{4K}*4$ = 4MB
  - 64位地址空间，页4K，页表项8B，
    - 页表大小: $\frac{2^{64}}{4K}*8$ = 33,554,432 GB

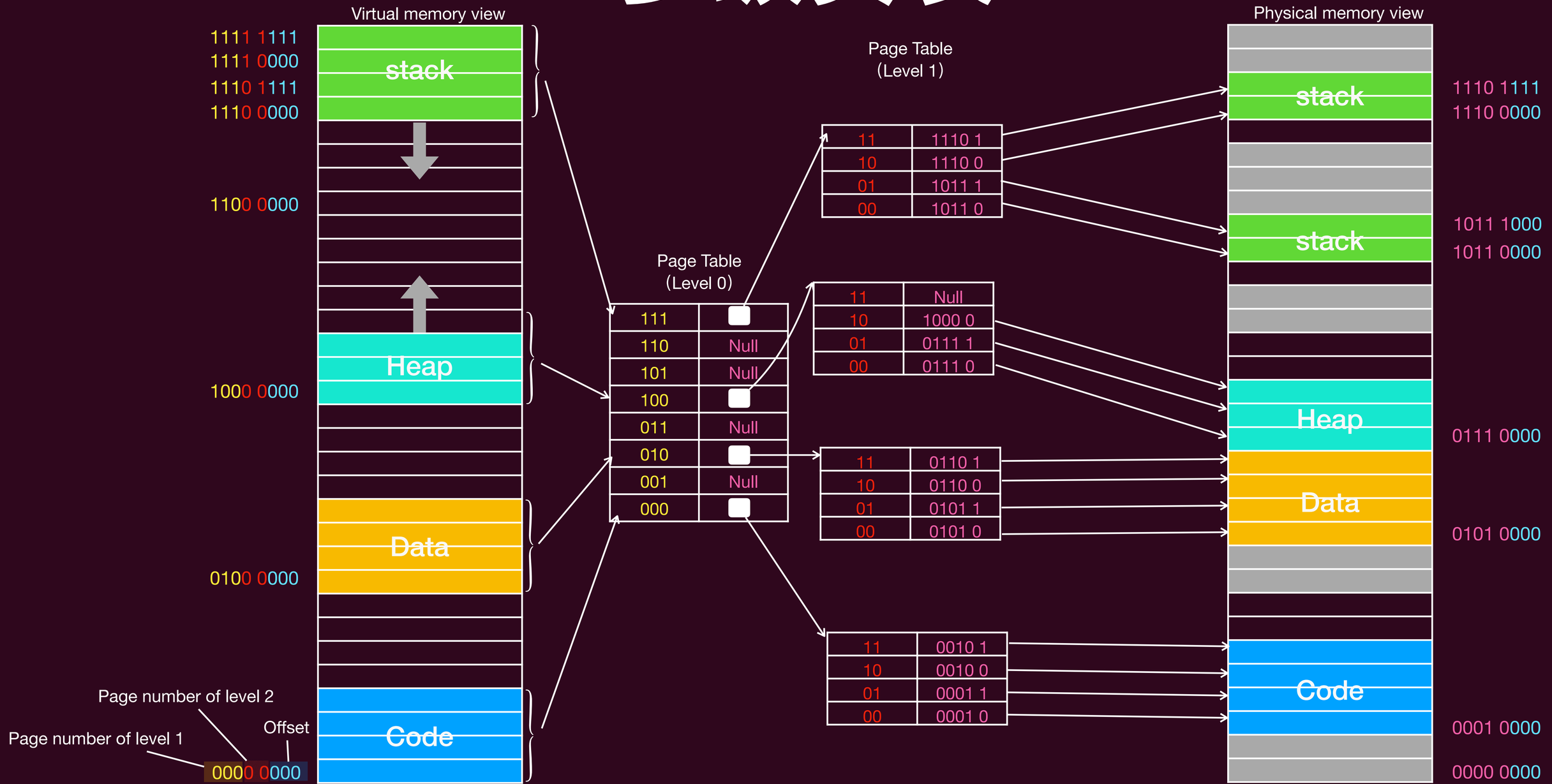
这还只是一个进程的页表!

解决方案：多级页表

- 使用多级页表减少空间占用
  - 若某级页表中的某条目为空，那么对应的下一级页表无需存在
  - 实际应用的虚拟地址空间大部分都未被使用，因此无需分配页表



多级页表

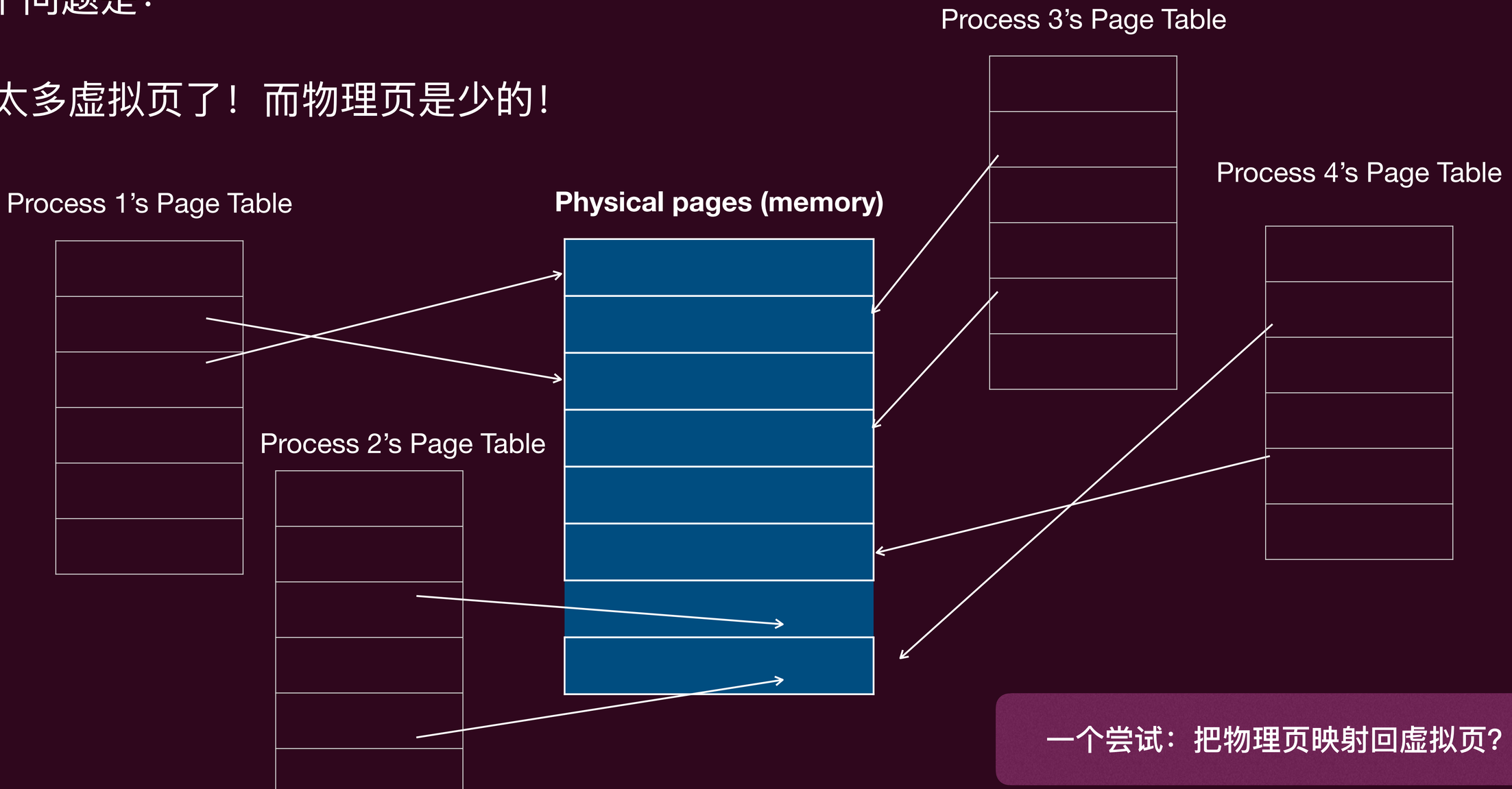

Virtual memory view
1111 1111
1111 0000
1110 1111
1110 0000
1100 0000
1000 0000
0100 0000
stack
Heap
Data
Code
Page number of level 2
Page number of level 1
Offset
0000 0000
Page Table
(Level 0)
111
110 Null
101 Null
100
011 Null
010
001 Null
000
Page Table
(Level 1)
11 1110 1
10 1110 0
01 1011 1
00 1011 0
11 Null
10 1000 0
01 0111 1
00 0111 0
11 0110 1
10 0110 0
01 0101 1
00 0101 0
11 0010 1
10 0010 0
01 0001 1
00 0001 0
Physical memory view
stack
stack
Heap
Data
Code
1110 1111
1110 0000
1011 1000
1011 0000
0111 0000
0101 0000
0001 0000
0000 0000

页表页

- 每级页表有若干离散的页表页
  - 每个页表页占用一个物理页
- 第 0 级(顶层)页表有且仅有一个页表页
  - 页表基地址寄存器(TTBR)存储的就是该页的物理地址
- 每项为8个字节
  - 即总共4096/8 = 512 项，用于存储物理地址和权限
- 可以不只是两级的页表

倒排页表(Inverted Page Tables)

- 一个问题是：
  - 太多虚拟页了！而物理页是少的！

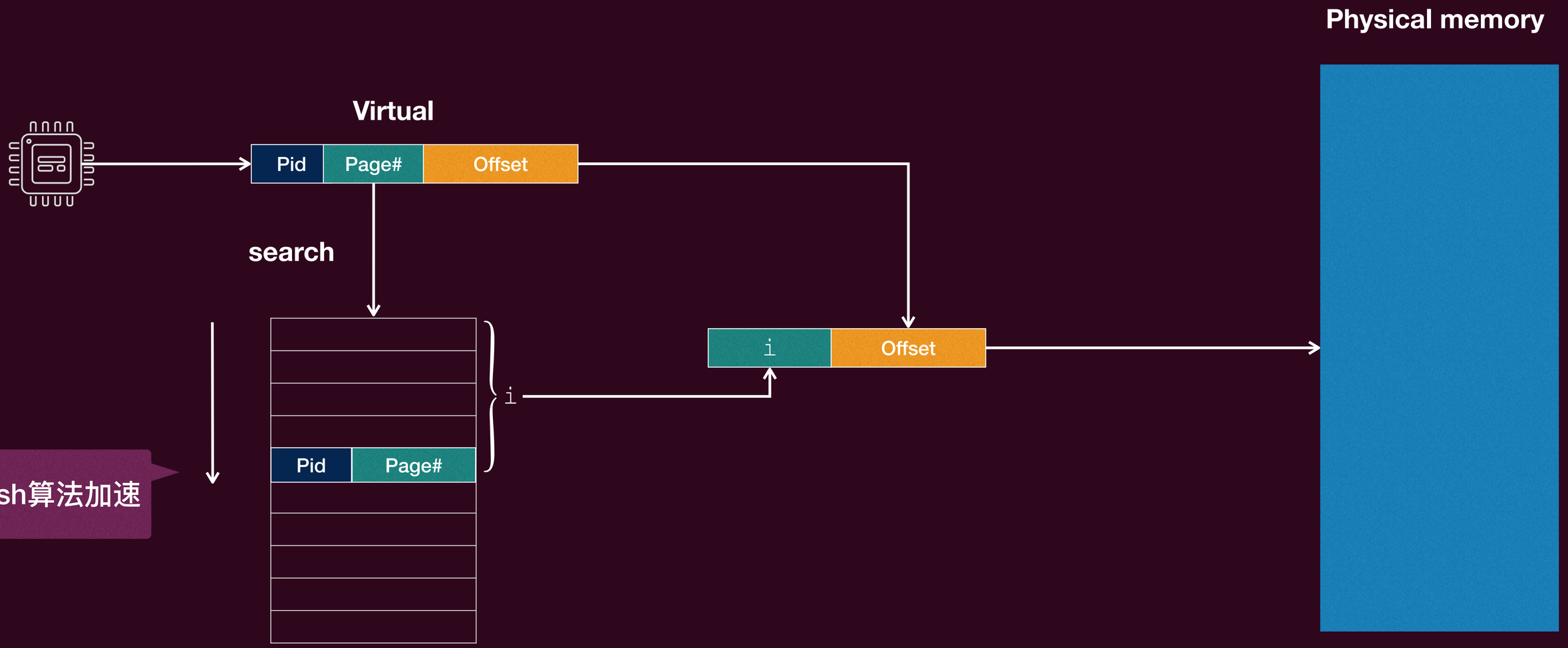


一个尝试：把物理页映射回虚拟页？

倒排页表(Inverted Page Tables)

- 解决方案：倒排页表
  - 不再为每个进程分别维护多个页表，而是保留一个单一的页表，每个物理页对应一个条目
- 每个页表条目包括：
  - 使用该物理页的进程(Pid)
  - 该进程的哪个虚拟页映射到该物理页
- 减少了存储页表所需的内存，但增加了在发生页引用时查找表所需的时间
  - 使用哈希表来加速查找
- 此外一个问题就是无法共享

倒排页表(Inverted Page Tables)

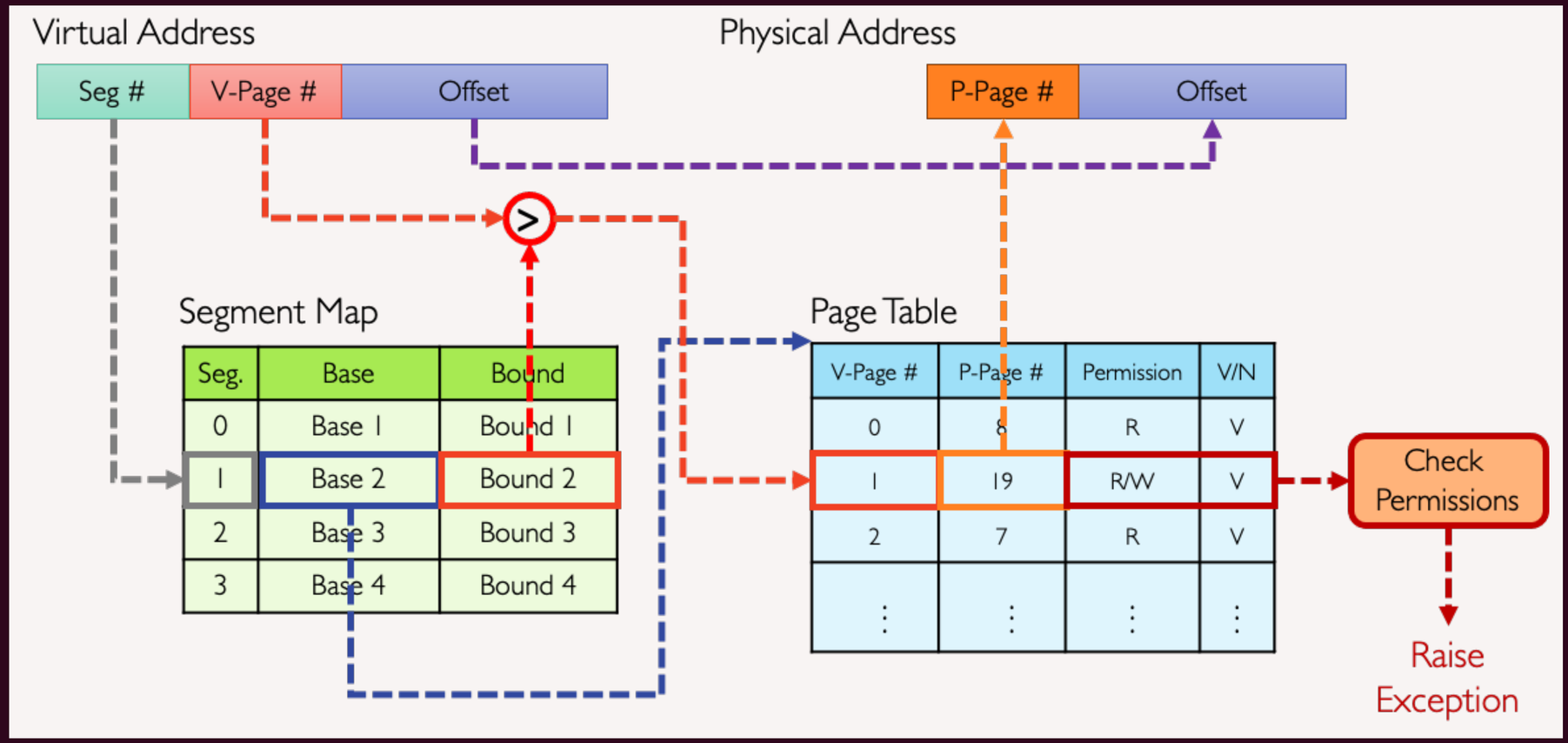


地址翻译比较

方法	优点	缺点
分段	快速上下文切换（段映射由CPU维护）	外部碎片
单级页表	无外部碎片 快速且易于分配	表的规模巨大 内部碎片
多级页表	表大小约为虚拟内存中的需要用的页面数量 快速且易于分配	每次页面访问涉及多次内存引用
倒排页表	表大小约为物理内存中的页面数量	需要复杂的哈希函数 页表没有缓存局部性

段页（Segments and Pages）

- 段表和页表可以混合使用：



(多级)页表不是完美的

- 多级页表的设计是典型的用时间换空间的设计
  - 能够减小页表所占空间
  - 但是增加了访存次数(逐级查询，级数越多越慢)
    - 即使是单级页表，也需要访存两次才能真正得到物理内存上的数据
- Tradeoff是计算机中经典而永恒的话题

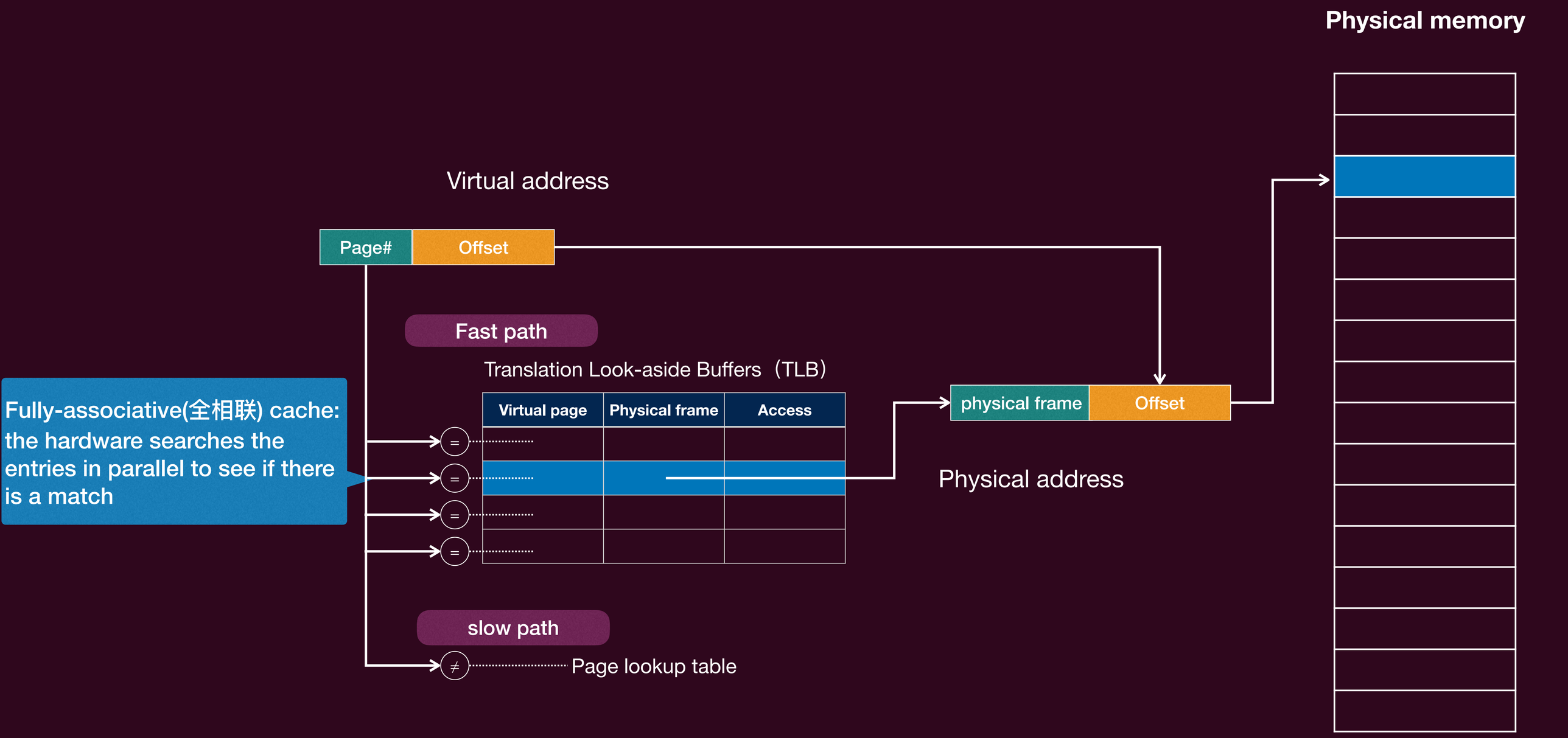
如何降低地址翻译的开销?

Translation Look-aside Buffers（TLB）
地址转换旁路缓冲

TLB

- 如何加速分页的地址转换?
  - 观察：大多数程序倾向于对少量页面进行大量引用
  - 使用一种特殊的快速查找硬件缓存，称为翻译后备缓冲器（TLB）或联想存储器（在内存管理单元中）
- 缓存最近地址转换
  - 如果TLB命中，直接应用转换（fast path）
  - 否则，如果TLB未命中，则在页表中查找映射（页表遍历），并更新TLB (slow path)

TLB


Physical memory
Virtual address
Page#
Offset
Fast path
Translation Look-aside Buffers（TLB）
Virtual page
Physical frame
Access
=
=
=
=
physical frame
Offset
Physical address
Fully-associative(全相联) cache: the hardware searches the entries in parallel to see if there is a match
slow path
≠
Page lookup table

TLB

- 一个典型的TLB缓存的项包括
  - 页号及其对应的帧号
  - 有效位：条目是否有有效的转换(注：和Page Entry的有效位意义不同！)
  - 保护位：页面的访问方式
  - 脏（修改）位：页面是否已被修改

Valid	Virtual page	Modified	Protection	Physical frame
1	140	1	RW	31
1	20	0	R X	32
1	130	1	R	38
1	860	1	RW	101

局部性原理（Principle of Locality）

- TLB背后的理念是利用指令和数据引用的局部性
  - 时间局部性：如果某个数据被访问过，那么在不久的将来它可能会再次被访问
  - 空间局部性：如果某个数据被访问过，那么它附近的数据在不久的将来也可能会被访问。
- TLB通常很小，包含64到1024个条目
  - 存储最可能被多的选中的地址翻译才能高效发挥TLB的作用
  - 支持某些条目可以固定下来以便永久快速访问

有效访问时间

- 有效访问时间（Effective Access Time, EAT）= $(\varepsilon + t)\alpha + (\varepsilon + 2t)(1 - \alpha)$
  - 命中率（α）：在TLB中找到页号的百分比
  - 内存访问时间（t）
  - TLB查找时间（ε）
- 假设命中率为80%，TLB查找时间为20ns，内存访问时间为100ns
  - 平均访问时间 (EAT) = 0.8 × 120 + 0.2 × 220 = 140ns（比率下降了40%）
- 如果命中率为99%
  - EAT = 0.99 × 120 + 0.01 × 220 = 121ns

处理TLB Miss

- 在TLB缺失时，值被加载到TLB中，以便下次更快地访问。
- 但如果TLB已满，应该替换谁呢?
  - 先来先出?
  - 最近最被少引用的(Least-Recently-Used，LRU)?
  - 随机?
- 谁来处理TLB miss?

处理TLB Miss

- 硬件处理的TLB Miss（如x86这样的CISC架构）
  - 当TLB缺失时，硬件进行页面遍历，获取页表项，并将其插入TLB
  - 硬件必须确切地知道页表在内存中的位置（通过PTBR寄存器），以及它们的确切格式
  - 更加迅速，对系统软件透明地完成
- 软件处理的TLB Miss（如MIPS这样的RISC架构）
  - 当TLB缺失时，硬件会引发异常（TLB故障）
  - 操作系统内部的代码处理TLB缺失（返回指令与系统调用返回不同）
  - 更加灵活（如可以定制替换策略等）

TLB一致性

- 上下文切换时，旧进程的虚拟到物理地址的转换应该不再有效，否则就会出现同样的虚拟地址映射到不同物理地址的问题，解决方案有两种：
  - 清空TLB：简单地将所有有效位设置为0
  - 带标记的TLB：在每个TLB条目中添加一个地址空间标识符（ASID）字段，该字段唯一地标识每个进程，为该进程提供地址空间保护

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

阅读材料

- [OSTEP]13、14、15、16、17、18、19、20章
- [现代操作系统]第4章