Virtual Memory
Segmentation & Paging

Liu yufeng
Fx_yfliu@163.com
Hunan University

Segmentation

Virtualizing Memory

- Memory virtualizing takes a similar strategy known as **limited direct execution(LDE)** for efficiency and control.
- In memory virtualizing, efficiency and control are attained by hardware support.
  - e.g., registers, TLB(Translation Look-aside Buffer)s, page-table
- Hardware transforms a virtual address to a physical address.
  - The desired information is actually stored in a physical address.
- The OS must get involved at key points to set up the hardware.
  - The OS must manage memory to judiciously intervene（明智的干预）

关键问题：如何高效、灵活地虚拟化内存

如何实现高效的内存虚拟化？如何提供应用程序所需的灵活性？如何保持控制应用程序可访问的内存位置，从而确保应用程序的内存访问受到合理的限制？如何高效地实现这一切？

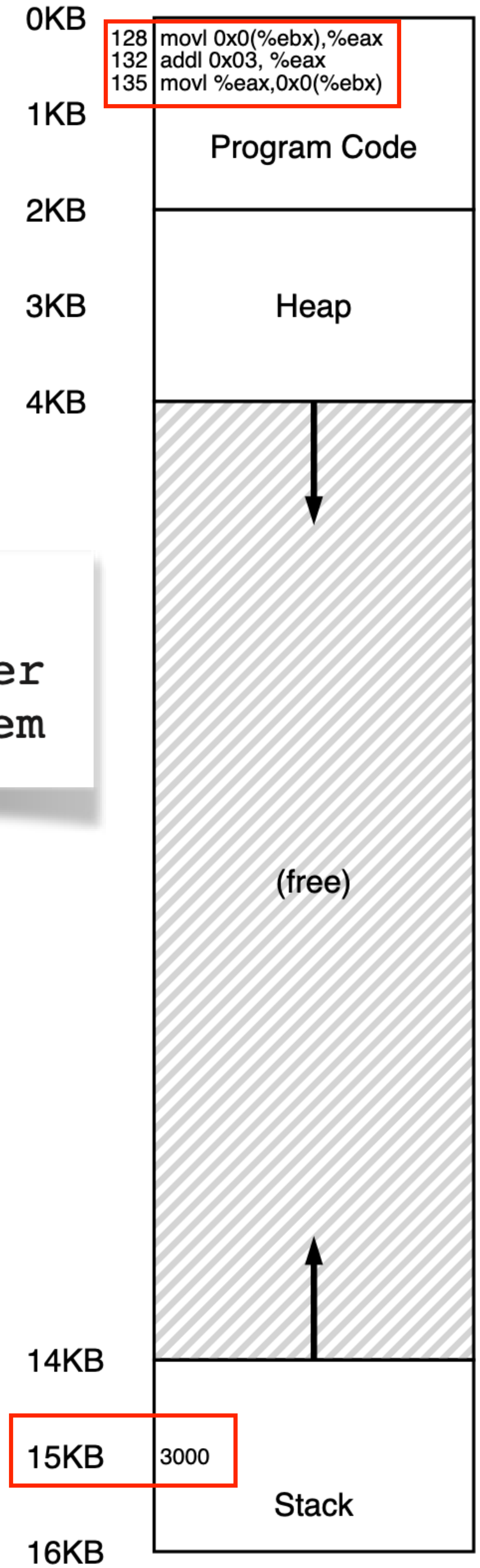
Assumptions

- Our first attempts at virtualizing memory will be very simple.
- Specifically, we will assume for now that the user's address space must be placed **contiguously** in physical memory.
- We also assume that the size of address space is not too big; specifically, that it is **less than the size of physical memory**.
- We also assume each address space has the **same size**.
- We will relax these assumptions as we go, thus achieving a realistic virtualization of memory.

An Example

```
void func()
    int x;
    ...
    x = x + 3;
```

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



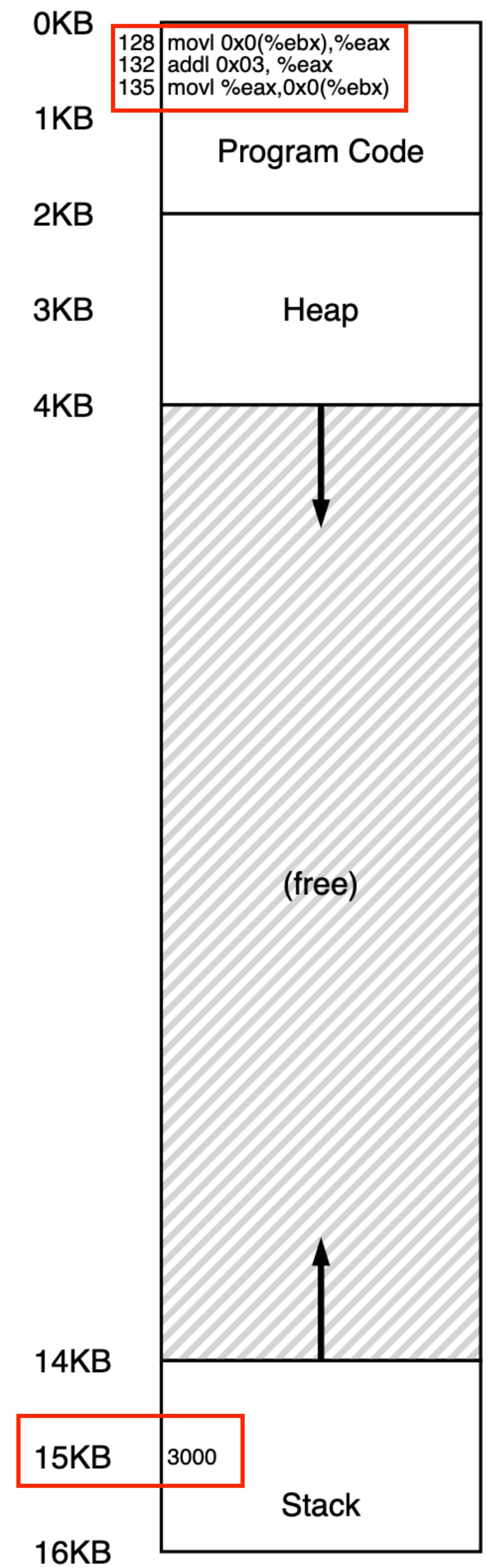
- 假定 x 的地址已经存入寄存器 ebx，之后通过 movl 指令将这个 地址的值加载到通用寄存器 eax。下一条指令对 eax 的内容加 3。最后一条指令 将 eax 中的值写回到内存的同一位置.

An Example

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

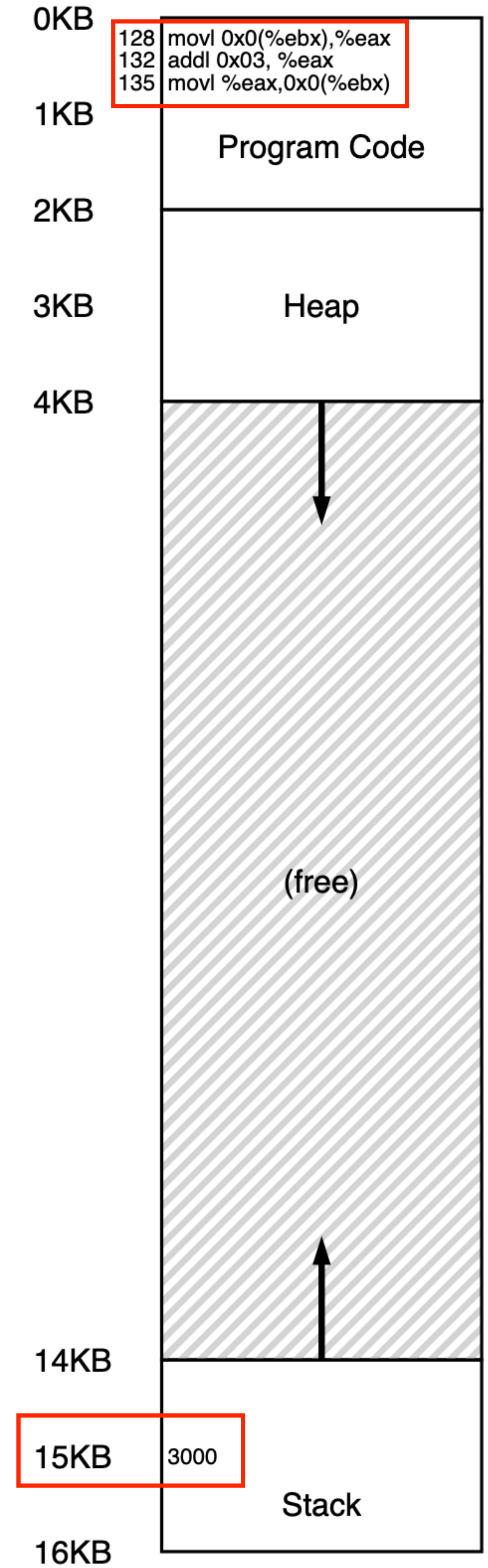
- 可以看到代码和数据都位于进程的地址空间，3 条指令序列位于地址 128（靠近头部的代码段），变量 x 的值位于地址 15KB（在靠近底部的栈中）。如图所示，x 的初始值是 3000。
- 从进程的角度来看，发生了以下几次内存访问

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

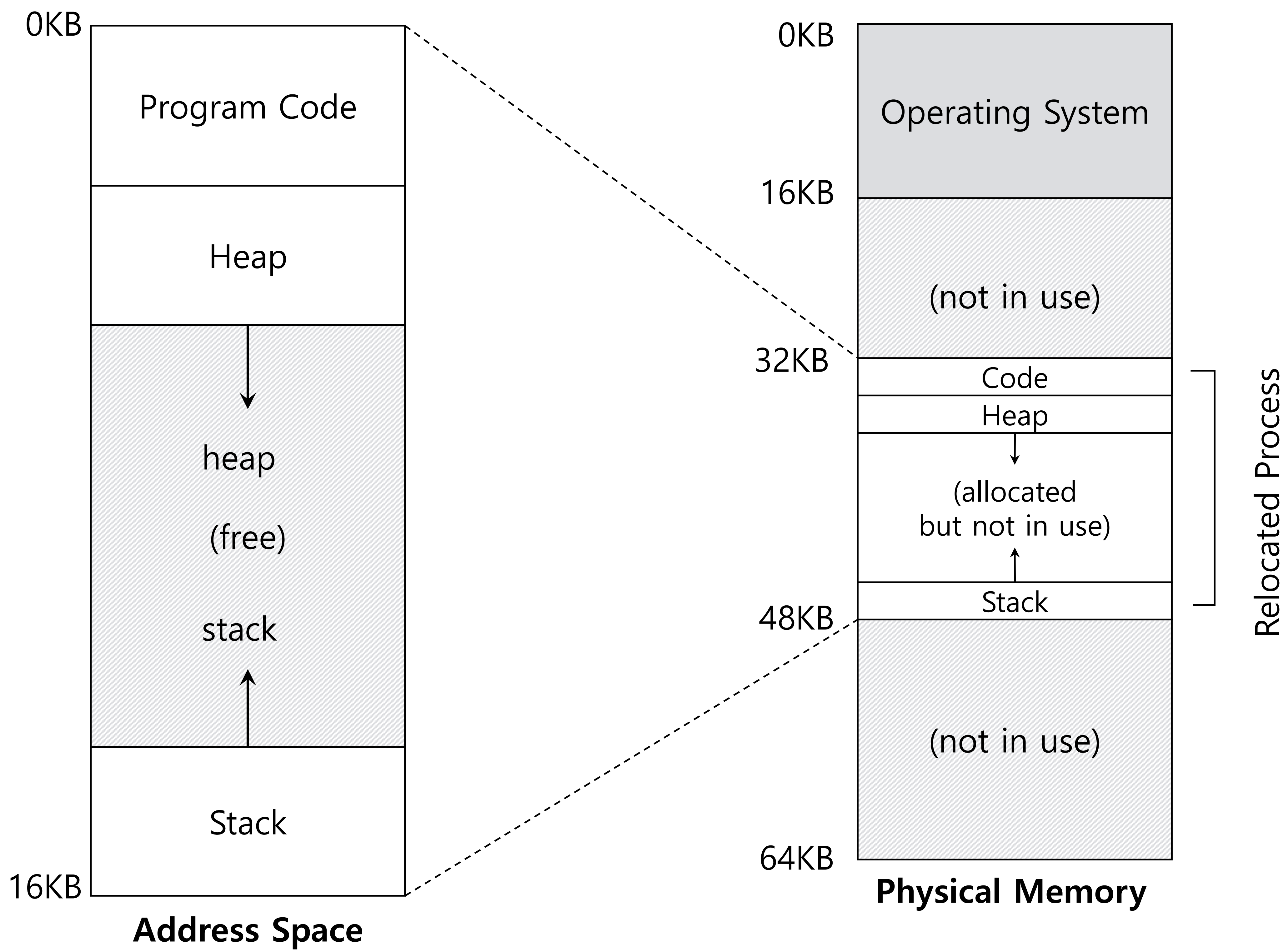


An Example

- 它的地址空间（address space）从 0 开始到 16KB 结束。它包含的所有内存引用都应该在这个范围内。
- 操作系统希望将这个进程地 址空间放在物理内存的其他位置，并不一定从地址 0 开始。
- 因此我们遇到了如下问题：怎 样在内存中重定位这个进程，同时对该进程透明（transparent）？ 怎么样提供一种虚拟地址 空间从 0 开始的假象，而实际上地址空间位于另外某个物理地址?

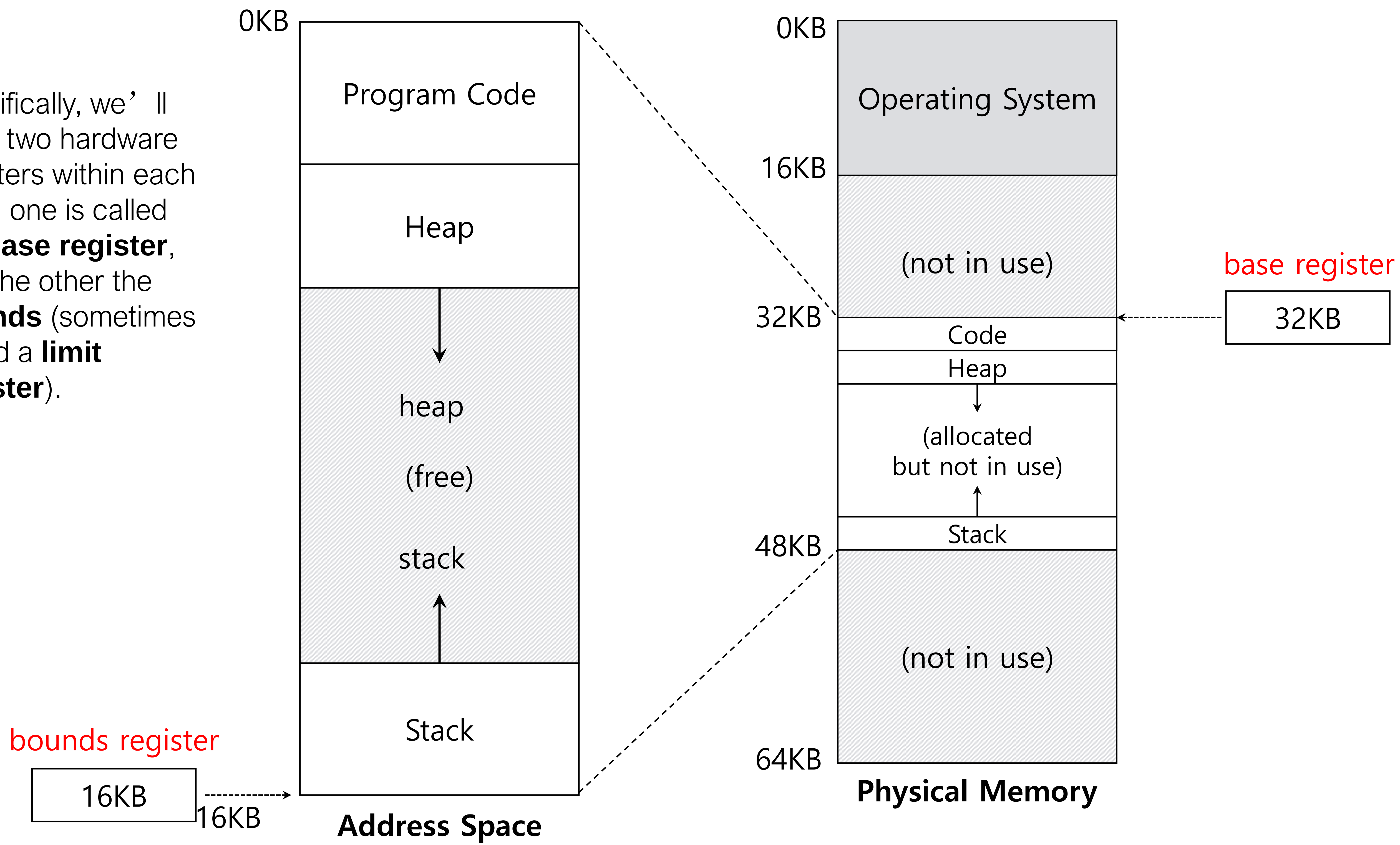


A Single Relocated Process



Dynamic (Hardware-based) Relocation

Specifically, we' ll need two hardware registers within each CPU: one is called the **base register**, and the other the **bounds** (sometimes called a **limit register**).



- When **any memory reference** is generated by the process, it is **translated** by the processor in the following manner:

$$\boxed{phycal\ address = virtual\ address + base}$$

$$\boxed{0 \leq\ virtual\ address < bounds}$$

Relocation and Address Translation

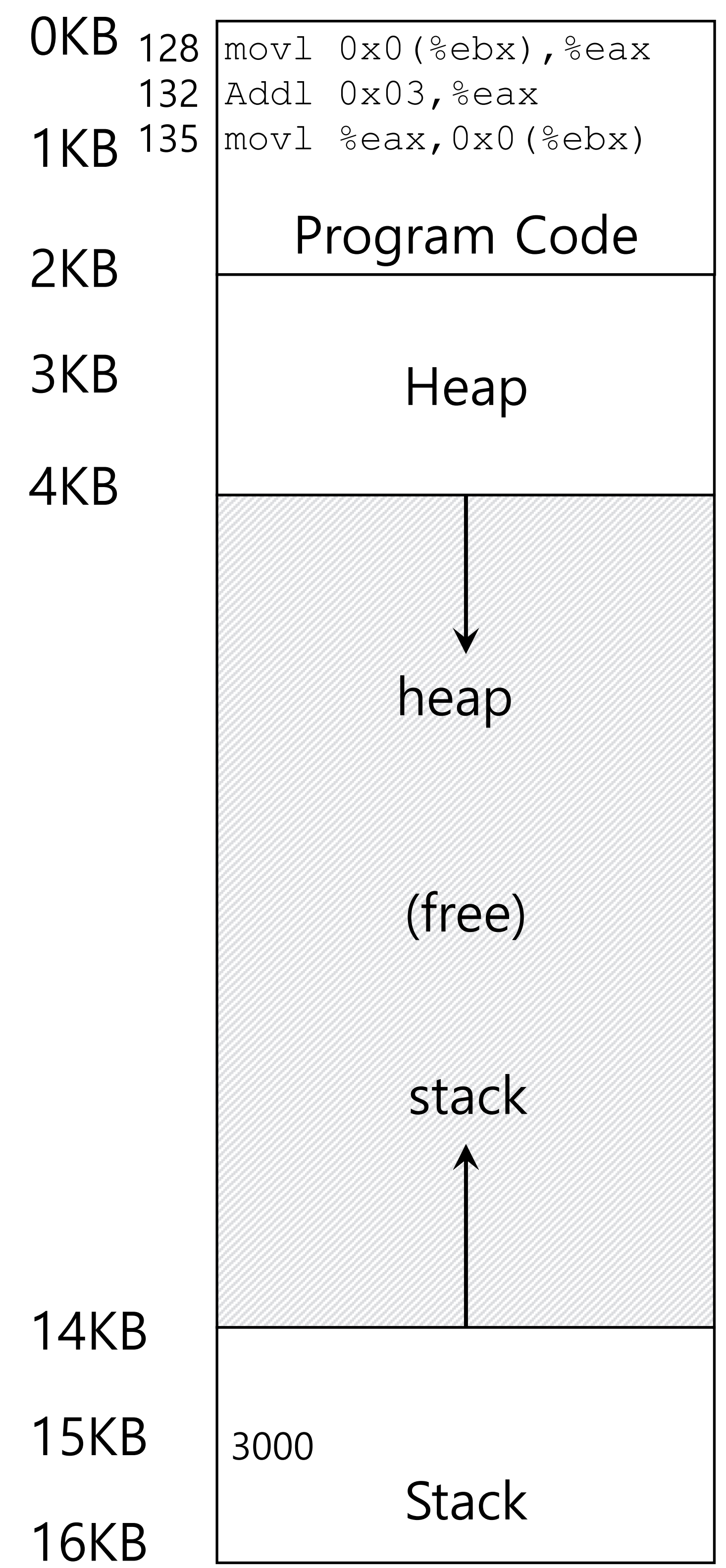
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

$$32896 = 128 + 32KB(base)$$

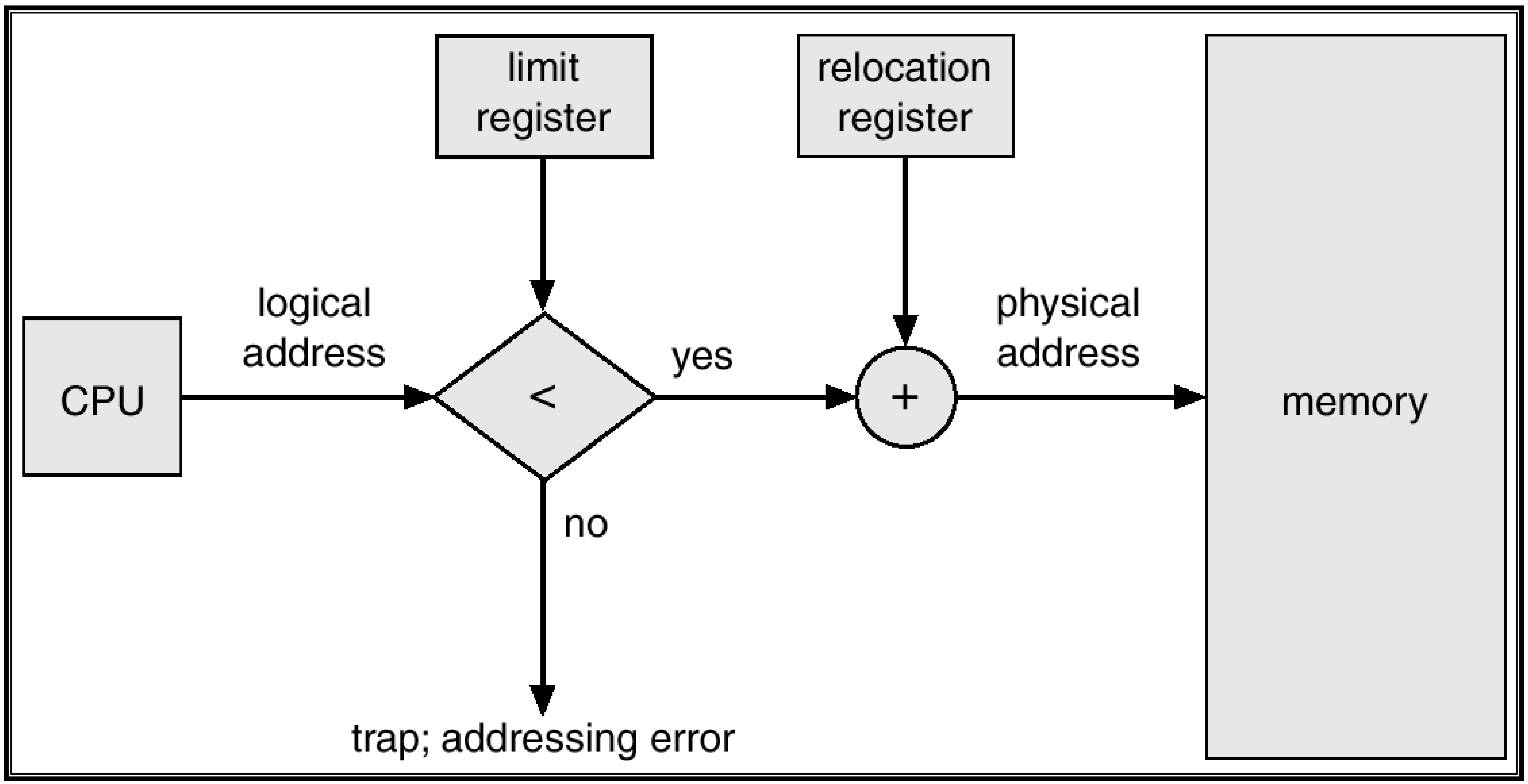
- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$



- Transforming a virtual address into a physical address is exactly the technique we refer to as **address translation**.
- Because this relocation of the address happens at runtime, the technique is often referred to as **dynamic relocation**.

Hardware Support for Relocation and Limit Registers



提示：基于硬件的动态重定位

在动态重定位的过程中，只有很少的硬件参与，但获得了很好的效果。一个基址寄存器将虚拟地址转换为物理地址，一个界限寄存器确保这个地址在进程地址空间的范围内。它们一起提供了既简单又高效的虚拟内存机制。

OS Issues

- 进程创建时，操作系统为进程的虚拟地址空间找到物理内存空间。由于我们假设每个进程的地址空间小于物理内存的大小，并且大小相同，这对操作系统来说 很容易。它可以把整个物理内存看作一组槽块，标记了空闲或已用。当新进程创建时，操 作系统检索这个数据结构（常被称为空闲列表，free list），为新地址空间找到位置，并将其 标记为已用。
- 在进程终止时（正常退出，或因行为不端被强制终止），操作系统也必须做一些 工作，回收它的所有内存，给其他进程或者操作系统使用。
- 在上下文切换时，操作系统也必须执行一些额外的操作。每个 CPU 毕竟只有一 个基址寄存器和一个界限寄存器，但对于每个运行的程序，它们的值都不同，因为每个程 序被加载到内存中不同的物理地址。因此，在切换进程时，操作系统必须保存和恢复基础 和界限寄存器。

Summary

- Base-and-bounds virtualization is quite **efficient**, as **only a little more hardware logic is required**.

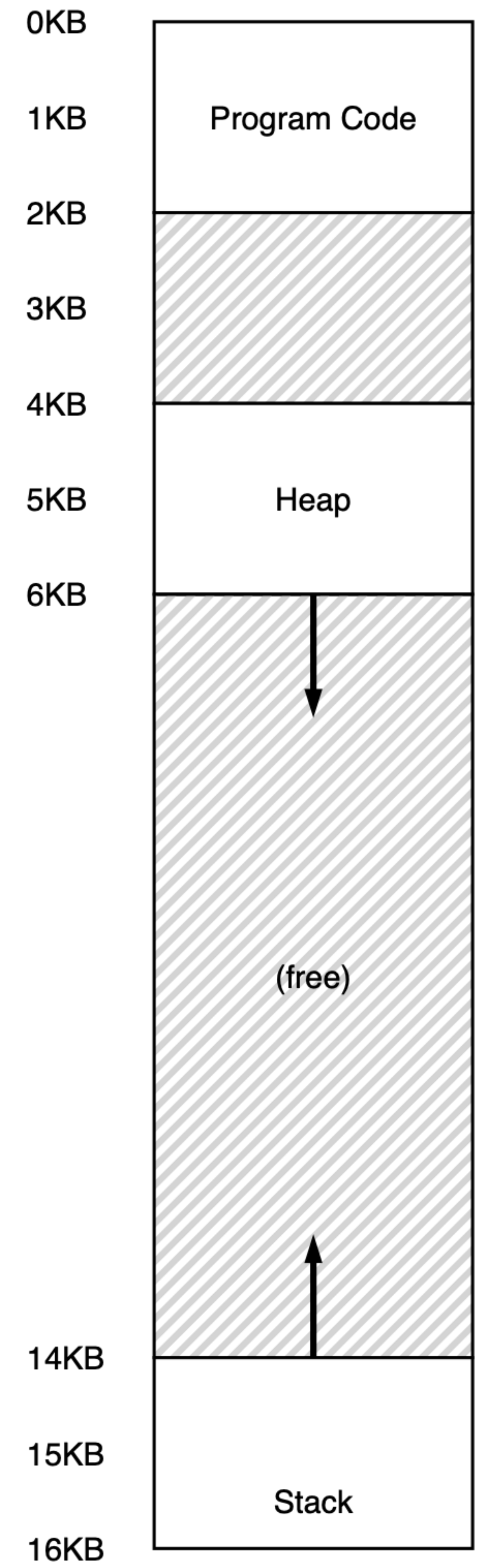
- Base-and-bounds also offers **protection**; the OS and hardware combine to ensure no process can generate memory references outside its own address space.

- However, it may lead to **internal fragmentation**, as the space inside the allocated unit is not all used.

- Our first attempt will be a slight generalization of base and bounds known as **segmentation**, which we discuss next.

Segmentation

- Big chunk of “free” space
- “free” space takes up physical memory.
- Hard to run when an address space does not fit into physical memory



关键问题：怎样支持大地址空间

怎样支持大地址空间，同时栈和堆之间（可能）有大量空闲空间？在之前的例子里，地址空间非常小，所以这种浪费并不明显。但设想一个 32 位（4GB）的地址空间，通常的程序只会使用几兆的内存，但需要整个地址空间都放在内存中。

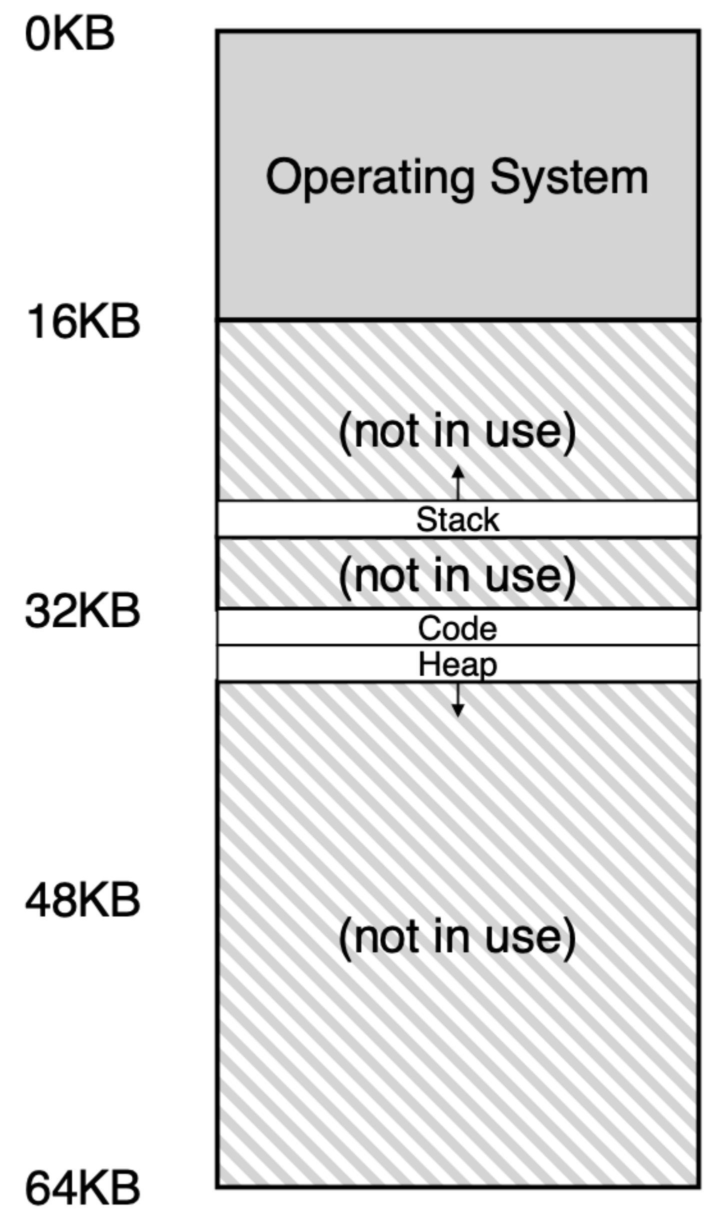
Segmentation: Generalized Base/Bounds

- To solve this problem, an idea called **segmentation** was born. An old idea, dating back to the very early **1960**' s.
- Segment is just a contiguous portion of the address space of a particular length.
  - Logically-different segment: code, stack, heap
- Each segment can be placed in different part of physical memory.
  - **Base** and **bounds** exist **per each segment**.

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

Segment Register Values

- For example, see the Figure; there you see a 64KB physical memory with those three segments within it.
- In this case, the hardware required to support segmentation needs a set of **three** base and bounds register pairs.



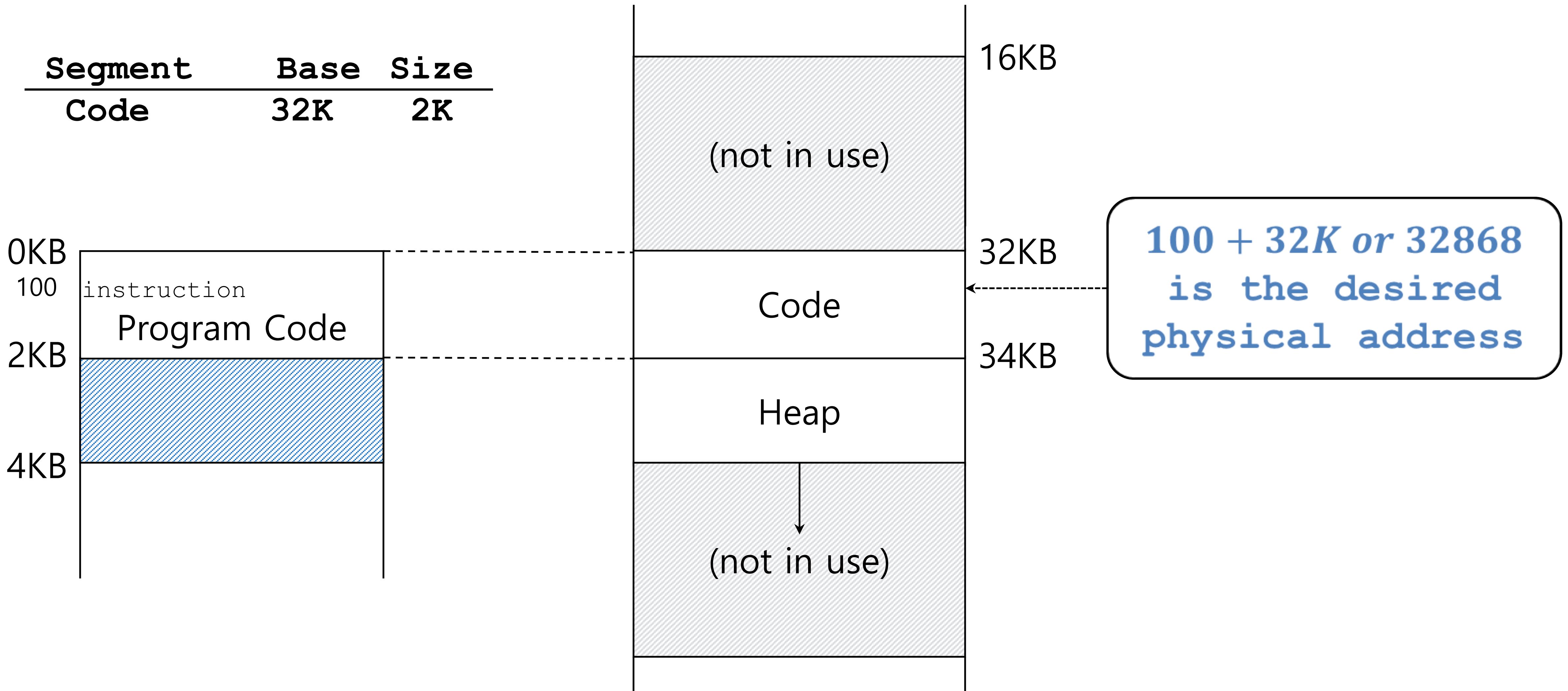
Placing Segments In Physical Memory

Address Translation on Segmentation

$$physical\ address = offset + base$$

- The `offset` of virtual address 100 is `100`.
  - The code segment **starts at virtual address 0** in address space.

Segment	Base	Size
Code	32K	2K

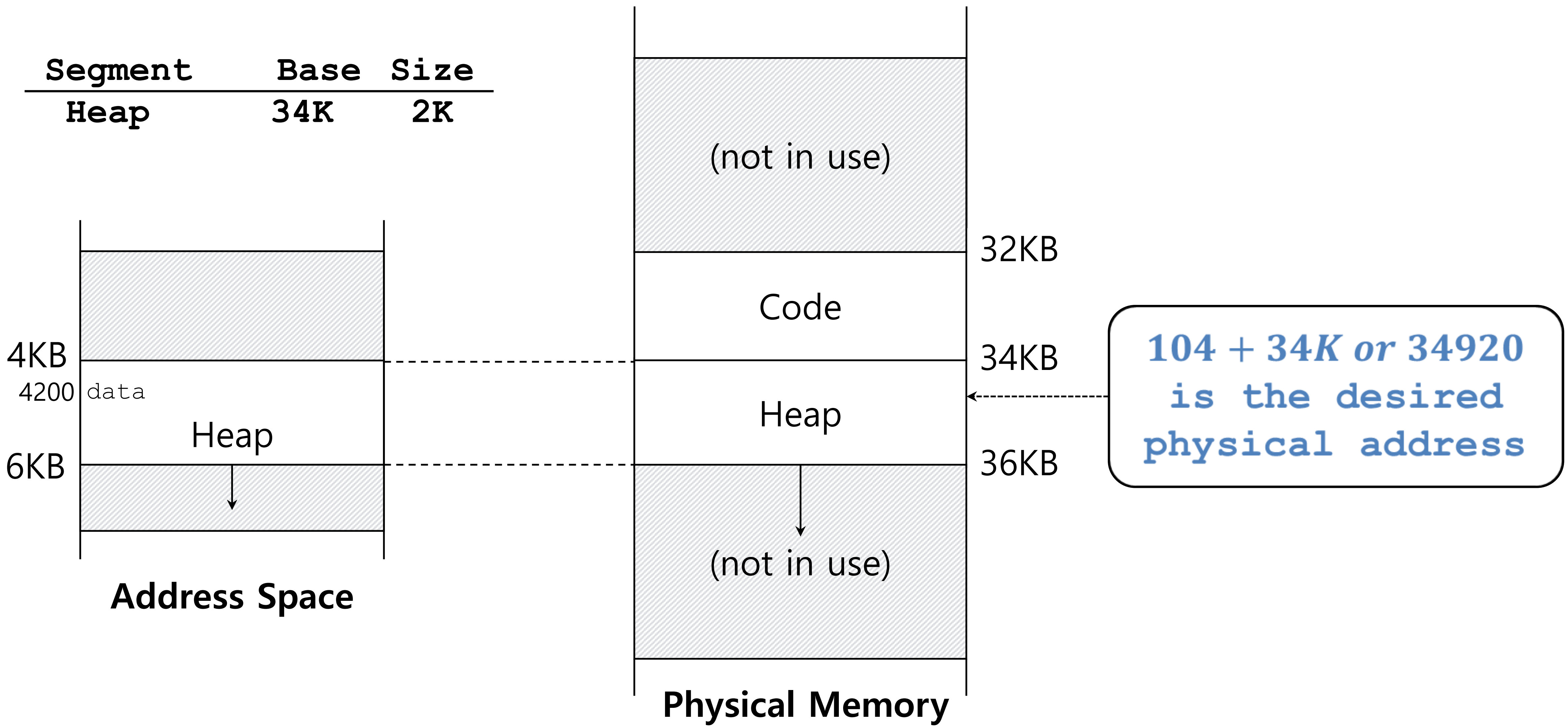


Address Translation on Segmentation(Cont.)

***Virtual address + base* is not the correct physical address.**

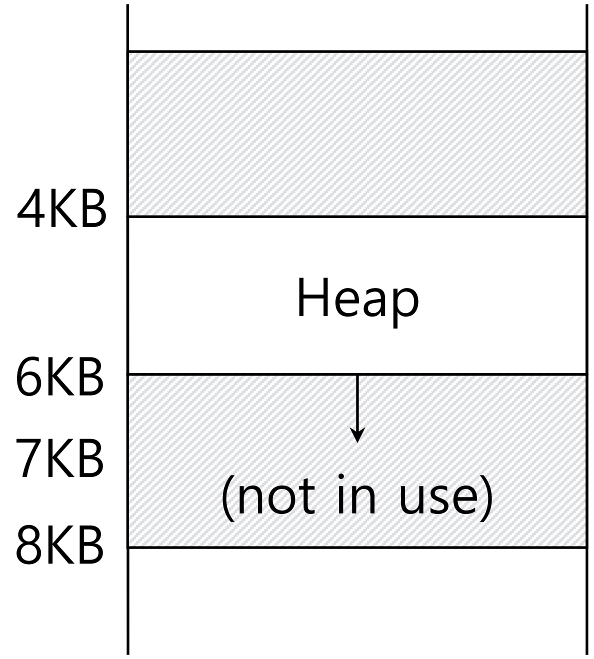
- The `offset` of virtual address 4200 is `104`.
  - The heap segment **starts at virtual address 4096** in address space.

Segment	Base	Size
Heap	34K	2K



Segmentation Fault or Violation

- If an illegal address such as 7KB which is beyond the end of heap is referenced, the OS occurs segmentation fault.
  - The hardware detects that address is out of bounds.



Address Space

补充：段错误

段错误指的是在支持分段的机器上发生了非法的内存访问。有趣的是，即使在不支持分段的机器上这个术语依然保留。但如果你弄不清楚为什么代码老是出错，就没那么有趣了。

Segmentation Fault

In computing, a **segmentation fault** (often shortened to **segfault**) or **access violation** is a fault raised by hardware with memory protection, notifying an operating system (OS) about a memory access violation; on x86 computers this is a form of general protection fault. The OS kernel will, in response, usually perform some corrective action, generally passing the fault on to the offending process by sending the process a signal. Processes can in some cases install a custom signal handler, allowing them to recover on their own, but otherwise the OS default signal handler is used, generally causing abnormal termination of the process (a program crash), and sometimes a core dump.

```c
#include <stdlib.h>
int main()
{
    char *c = "hello world";
    c[1] = 'H';
}
```

```c
#include <stdlib.h>
int main()
{
    char *c = "hello world";
    c[1] = 'H';
}
```

char* p = “hello world"; 这个声明，声明了一个指针，而这个指针指向的是全局的const内存区，const内存区当然不会让你想改就改的

一个由c/C++编译的程序占用的内存分为以下几个部分

代码区：存放程序的代码，即CPU执行的机器指令，并且是只读的。

常量区：存放常量(程序在运行的期间不能够被改变的量，例如: 10，字符串常量”abcde”等

静态区（全局区）：静态变量和全局变量的存储区域是一起的，一旦静态区的内存被分配, 静态区的内存直到程序全部结束之后才会被释放

堆区：由程序员调用malloc()函数来主动申请的，需使用free()函数来释放内存，若申请了堆区内存，之后忘记释放内存，很容易造成内存泄漏

栈区：存放函数内的局部变量，形参和函数返回值。

```c
#include <stdlib.h>
int main()
{
    char c[] = "hello world";
    c[0] = 'H';
}
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int* p = (int*)0xC0000fff;
     *p = 10;
}
```

int *p中的 p 必须赋予一个地址，如果不是地址，编译器则会报错，在此直接指定了一个地址，非法

试图向内存地址0存放一个值

```c
int  i=0;
scanf ("%d", i);  /* should have used &i */
printf ("%d\n", i);
return 0;
```

```c
int  main()
{
    int b = 10;
    printf("%s\n", b);
    return 0;
}
```

在打印字符串的时候，实际上是打印某个地址开始的所有字符，但是当你想把整数当字符串打印的时候，这个整数被当成了一个地址，然后printf从这个地址开始去打印字符，直到某个位置上的值为\0。所以，如果这个整数代表的地址不存在或者不可访问，自然也是访问了不该访问的内存——segmentation fault。

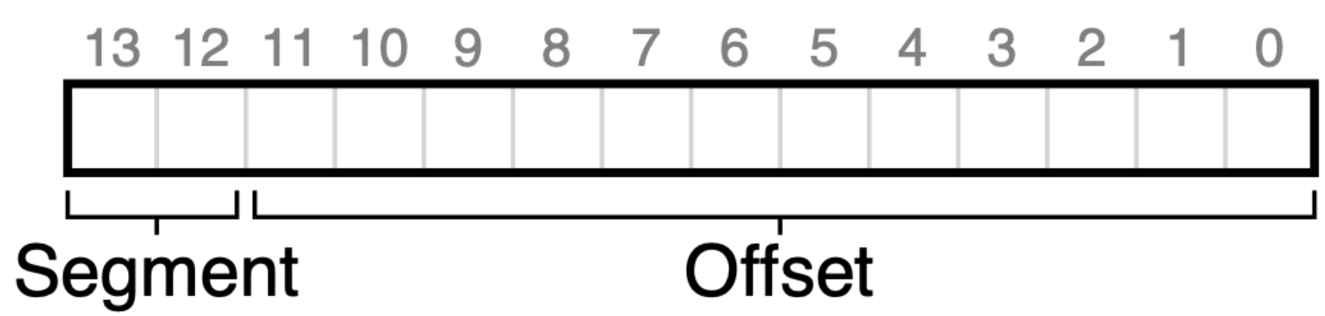
```c
int main(void)
{
    main();
    return 0;
}
```

“堆栈”是一块预留的内存，用来放置局部变量和函数调用的返回地址。当堆栈耗尽时，保留区域以外的内存将被访问。但是应用程序没有向内核请求这个内存，因此会生成一个SegFault来保护内存。

What is the difference between a segmentation fault and a stack overflow?

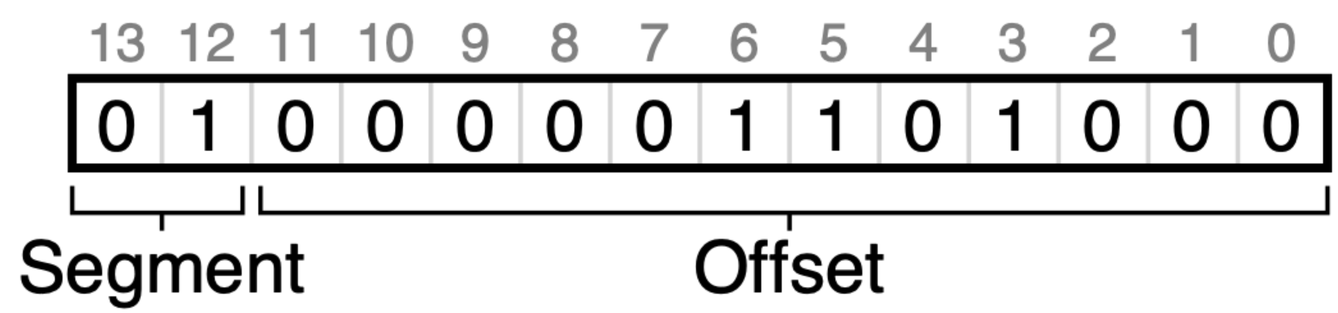
Which Segment Are We Referring To?

- 硬件在地址转换时使用段寄存器。它如何知道段内的偏移量，以及地址引用了哪个段?
- 一种常见的方式，有时称为显式（explicit）方式，就是用虚拟地址的开头几位来标识不同的段
- The size of the address space in our example is **16KB**, which can be represented with 2^{14}



- Let's take our example heap virtual address from above **4200** (01000001101000) and translate it, just to make sure this is clear. The virtual address 4200, in binary form, can be seen here:

Segment	bits
Code	00
Heap	01
Stack	10
-	11



- As you can see from the picture, the top two bits (**01**) tell the hardware **which segment** we are referring to. The bottom **12** bits are the **offset** into the segment: **0000 0110 1000**, or hex **0x068**, or **104** in decimal.

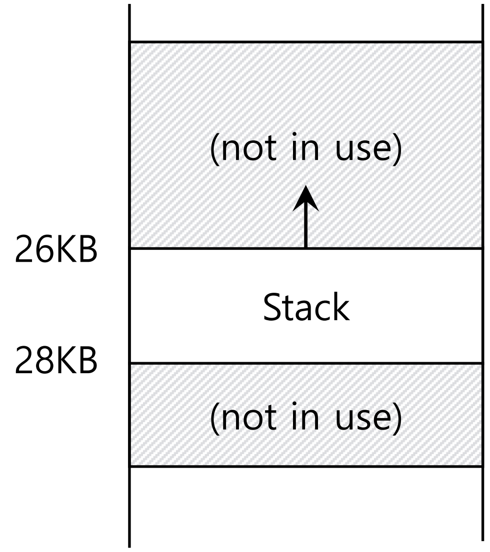
```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset  = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

- 在上例中
- SEG_MASK 为 0x3000
- SEG_SHIFT 为 12
- OFFSET_MASK 为 0xFFF。

What About The Stack?

- Stack grows **backward.**
- **Extra hardware support** is need.
  - The hardware checks which way the segment grows.
  - 1: positive direction, 0: negative direction

In our example, the stack is relocated to physical address **28KB**, but with one **critical difference**: **it grows backwards**. In physical memory, it starts at **28KB** and grows back to **26KB**, corresponding to virtual addresses **16KB** to **14KB**; **translation must be processed differently**.



Physical Memory

Segment Register(with Negative-Growth Support)

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

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

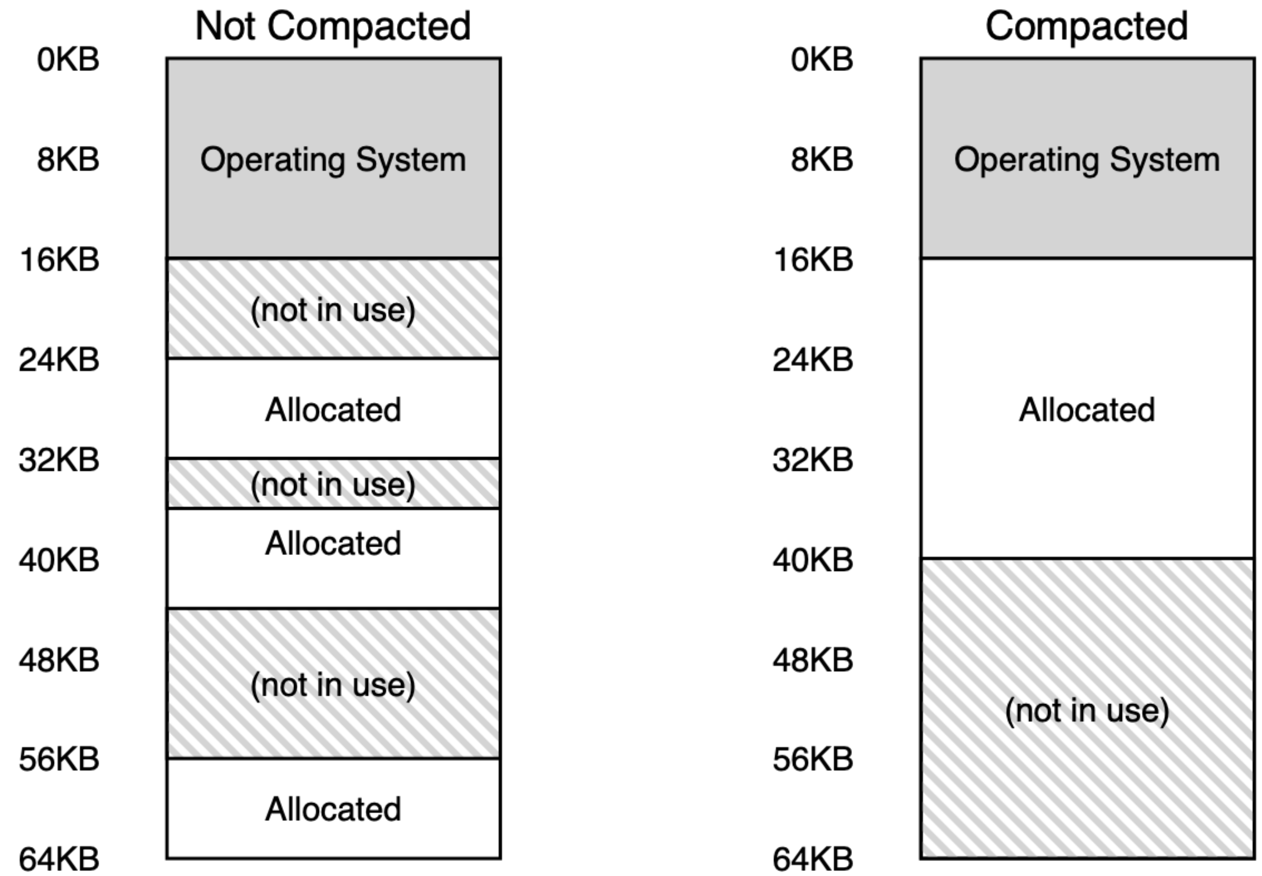
Segment Registers (With Negative-Growth Support)

- Assume we wish to access virtual address **15KB**, which should map to physical address **27KB**.
- 假设要访问虚拟地址 15KB，它应该映射到物理地址 27KB。
- 该虚拟地 址的二进制形式是：11 1100 0000 0000（十六进制 0x3C00）。硬件利用前两位（11）来指定 段，但然后我们要处理偏移量 3KB。为了得到正确的反向偏移，我们必须从 3KB 中减去最 大的段地址：在这个例子中，假设段的大小是 4KB，因此正确的偏移量是 3KB 减去 4KB，即−1KB。只要用这个反向偏移量（−1KB）加上基址（28KB），就得到了正确的物理地址 27KB。

OS Support

- What should the OS do on a context switch?
- 第一个 问题：操作系统在上下文切换时应该做什么？各个段寄存器中的内容必须保存和恢复。显然，每个进程都有自己独立的虚拟地址空间，操作系统必须在进 程运行前，确保这些寄存器被正确地赋值。

[R69] “A note on storage fragmentation and program segmentation”
Brian Randell
Communications of the ACM
Volume 12(7), pages 365-372, July 1969
One of the earliest papers to discuss fragmentation.



Non-compacted and Compacted Memory

- 第二个问题：物理内存的空闲空间。新的地址空间被创建时，操作系统 需要在物理内存中为它的段找到空间。我们假设所有的地址空间大小相同，物理内存可以被认为是一些槽块，进程可以放进去。现在，每个进程都有一些段，每个段的大小也可能不同。 一般会遇到的问题是，物理内存很快充满了许多空闲空间的小洞，因而很难分配给新的段，或扩大已有的段。这种问题被称为外部碎片（external fragmentation）

- 该问题的一种解决方案是紧凑（compact）物理内存，重新安排原有的段。

- 另一种方案是空闲列表管理算法.

Paging

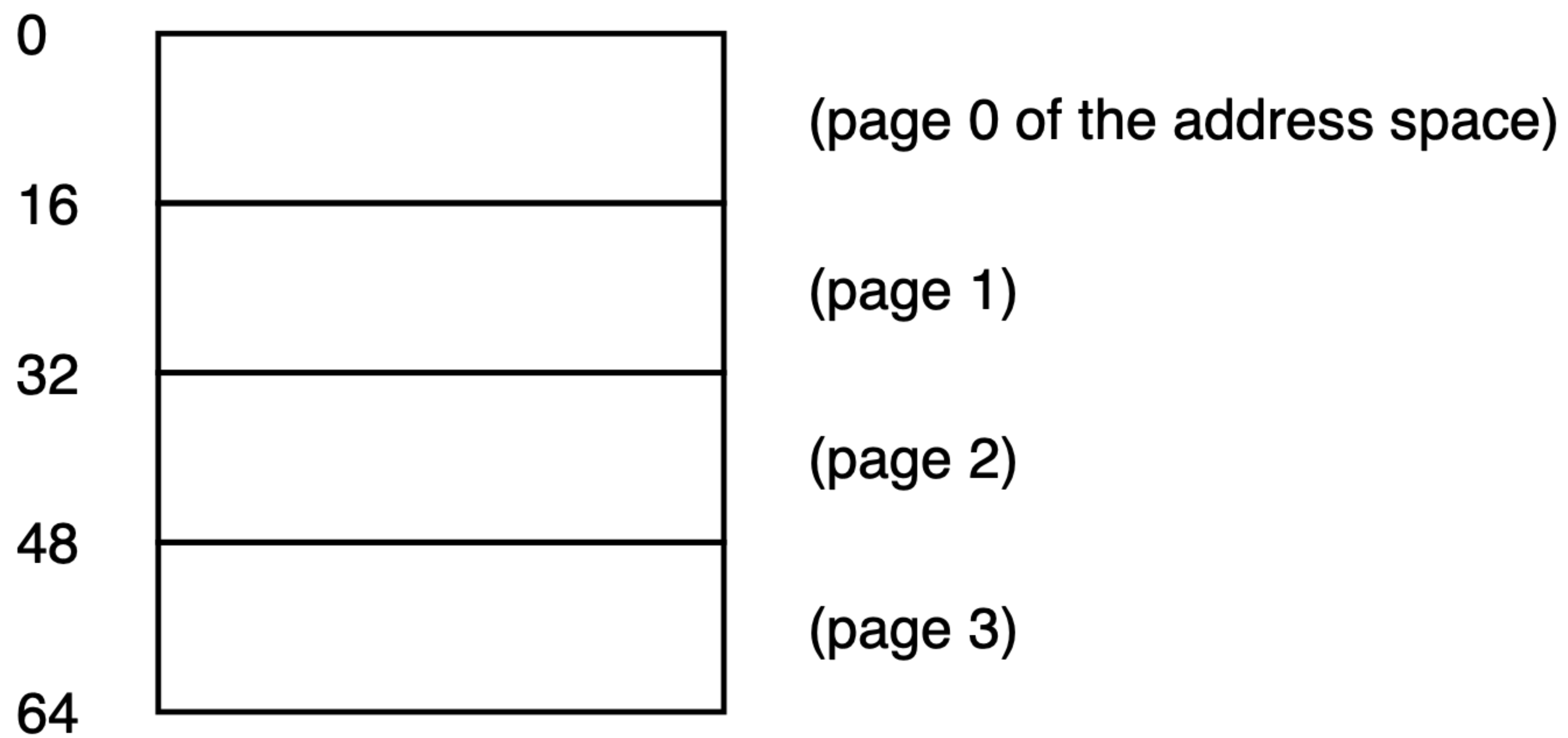
Introduction

- Remember **our goal**: to virtualize memory.
- **Segmentation helped us do this, but has some problems**; in particular, managing free space becomes quite a pain as memory becomes fragmented and segmentation is not as flexible as we might like. （空闲空间碎片化问题）
- Is there a better solution?

关键问题：如何通过页来实现虚拟内存

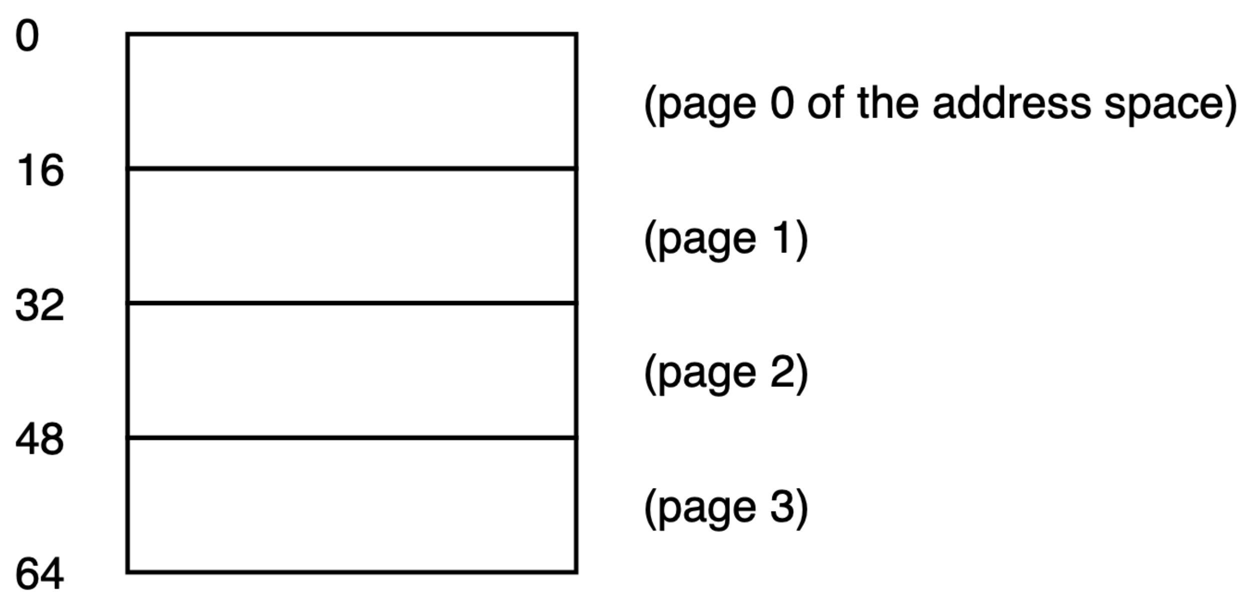
如何通过页来实现虚拟内存，从而避免分段的问题？基本技术是什么？如何让这些技术运行良好，并尽可能减少空间和时间开销？

- 分页不是将一个进程的地址空间分割成几个不同长度的逻辑段（即代码、堆、段），而是分割成固定大小的单元，每个单元称为一页。

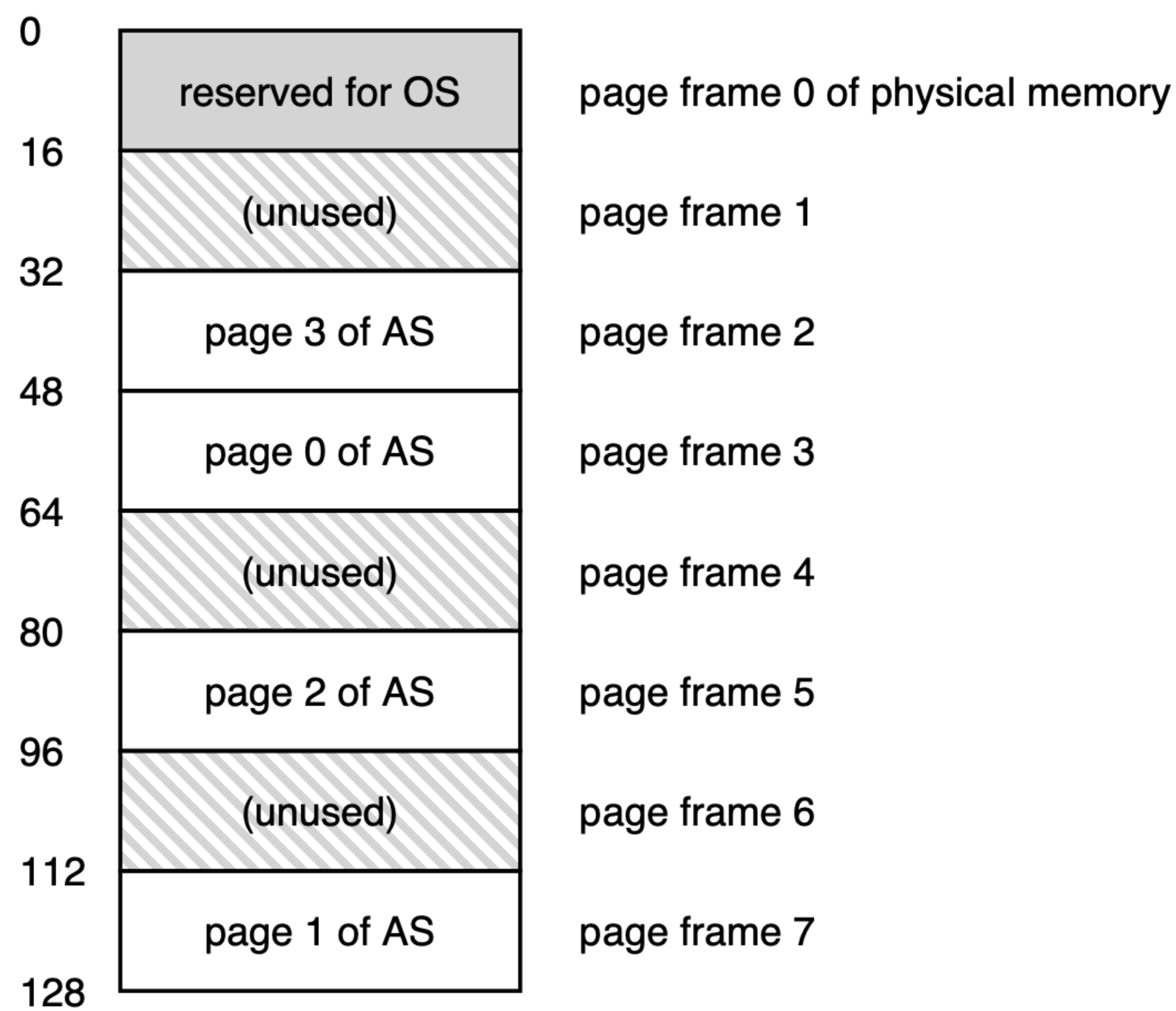


A Simple 64-byte Address Space

- 相应地，我们把物理内存看成是定长槽块的阵列，叫作页帧（page frame）。每个这样的页帧包含一个虚拟内存页.



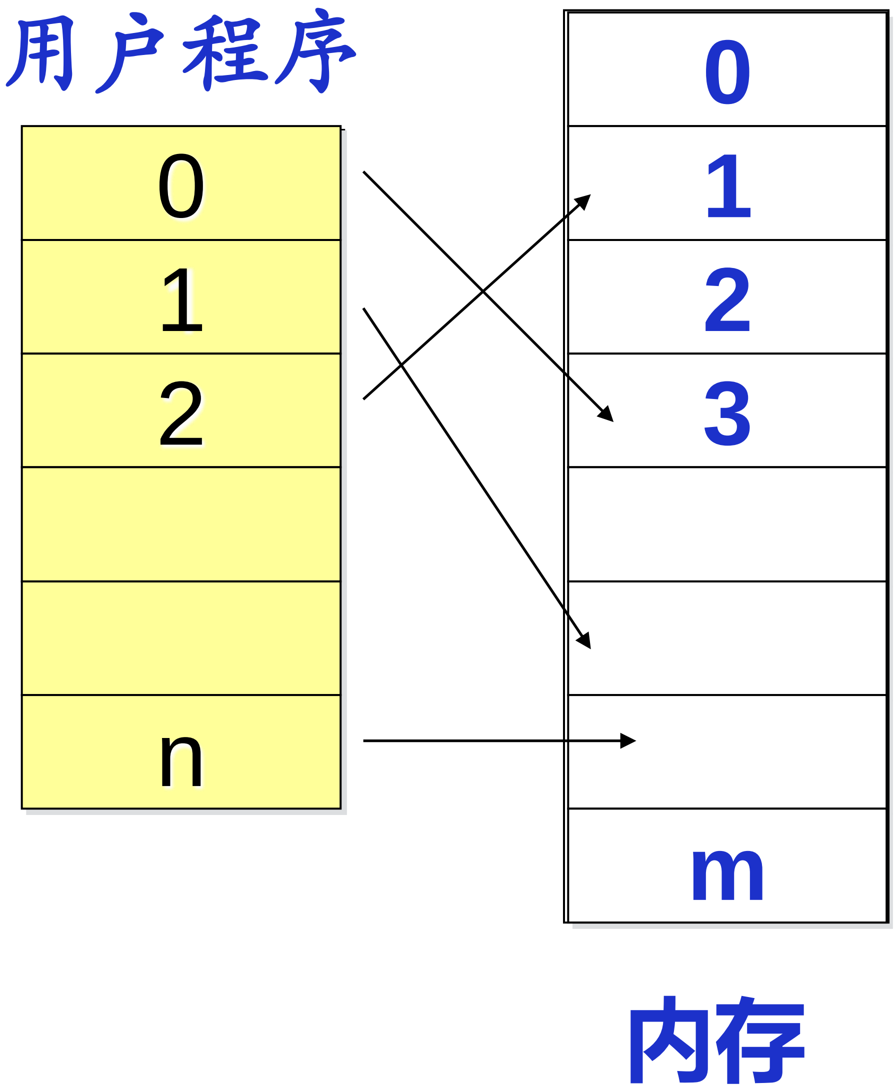
A Simple 64-byte Address Space

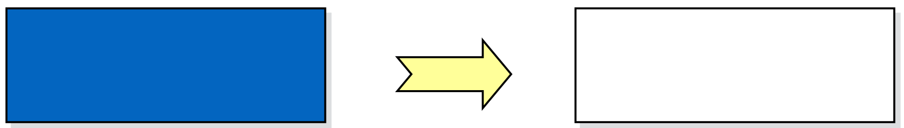


64-Byte Address Space Placed In Physical Memory

分页存储管理基本思想

- 分页存储管理基本思想
- 1）非连续分配的基础
  - 分页：将程序地址空间分页
  - 分块：将内存空间分块（帧）
  - 页/块：几K~几十K字节
- 2）非连续分配的体现
  - 内存一块可以装入程序一页
  - 连续的多个页不一定装入连续的多个块中
  - 注：系统中页块的大小是不变的。
- 3）非连续分配的特点
  - 没有外零头
    - 不受连续空间限制，每块都能分出去
  - 仅有小于一个页面的内零头
    - 程序大小一般不是页大小的整数倍





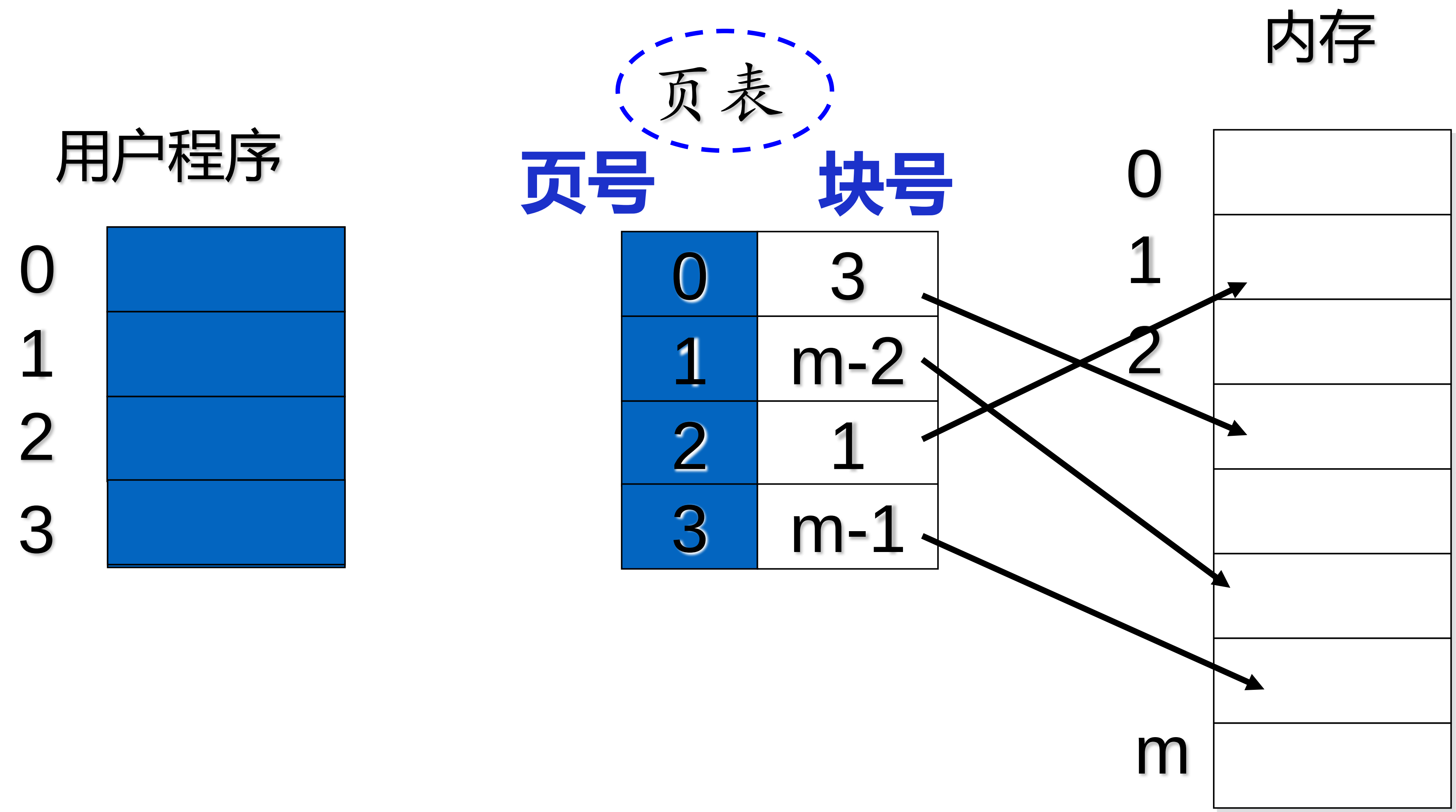
- Paging, as we will see, has a number of **advantages** over our previous approaches.
- Probably the most important improvement will be **flexibility**: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively.
- Another advantage is the **simplicity** of free-space management that paging affords.

分页存储管理实现的方法

- 分页存储管理的基本方法

  基本问题：
  - 如何建立虚拟空间与物理空间的映射
  - 如何进行地址变换
    - 从程序逻辑地址到内存物理地址

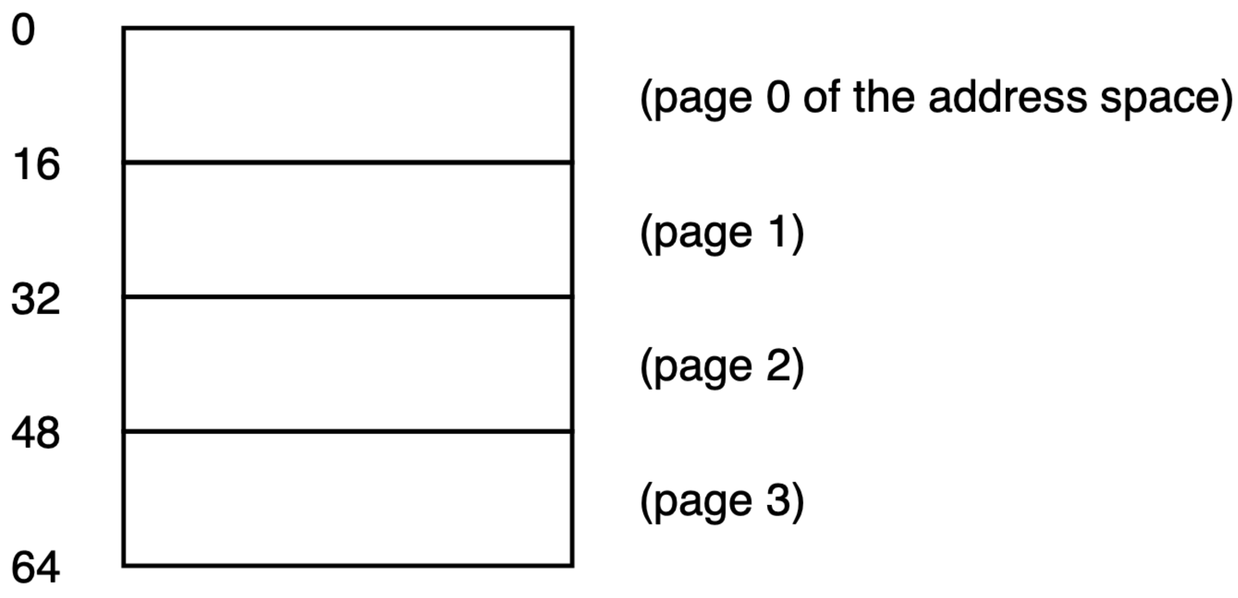
程序空间	内存空间
逻辑空间	物理空间
相对地址	绝对地址



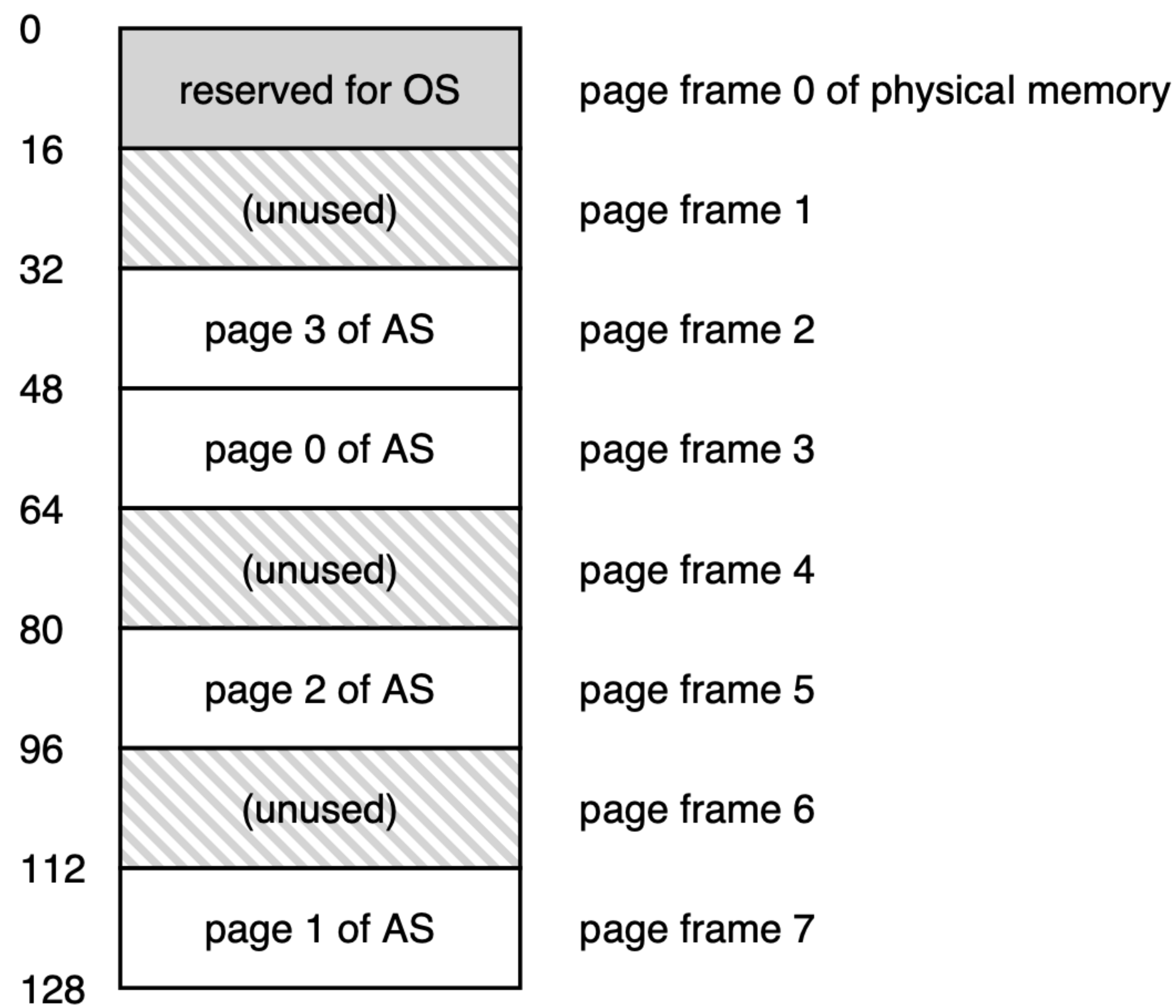
Page Table

- 页表的主要作用是为地址空间的每个虚拟页面保 存地址转换（address translation），从而让我们知道每个页在物理内存中的位置
- 每个进程都有一张自己的页表
- 如果运行另一个进程，OS将不得不为进程管理一个不同的页表，因为不同进程的虚拟页显然映射到不同的物理页

- 相应地，我们把物理内存看成是定长槽块的阵列，叫作页帧（page frame）。每个这样的页帧包含一个虚拟内存页.



A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

页表具有以下 4 个条目：

（VP 0→ PF 3）、

（VP 1→PF 7）、

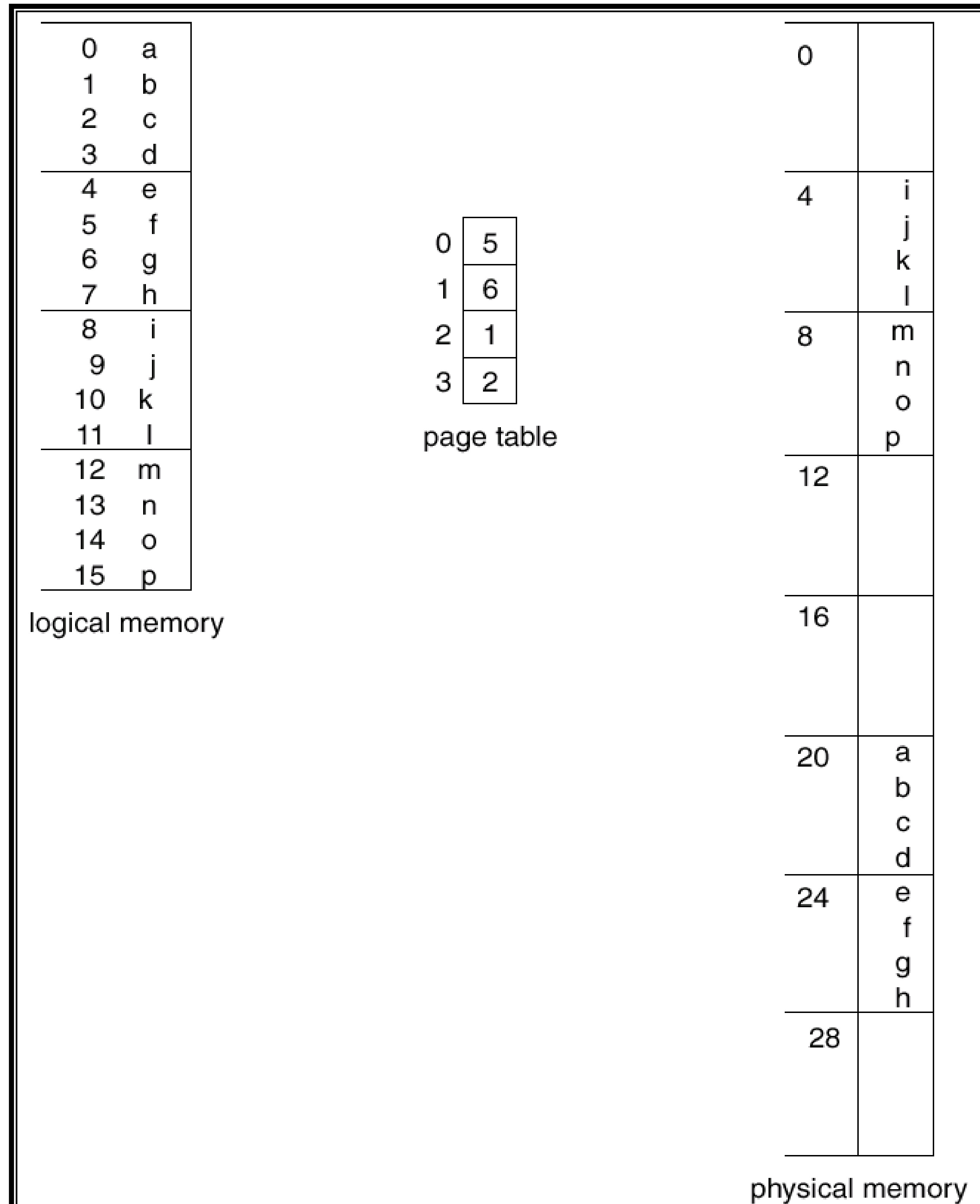
（VP 2→PF 5）、

（VP 3→PF 2）。

Paging Example

page number	page offset
p	d
m - n	n

设定页面大小为4 byte，物理内存是32 byte（8 页）。逻辑地址0 在第0页，页偏移为0。根据索引页表，我们发现第0页在第5帧。这样，逻辑地址0 映射到物理地址20 (= (5 × 4) + 0)。逻辑地址3 (第0 页，页偏移为3）映射到物理地址23 (= (5 × 4) + 3)。逻辑地址4 在第1页，偏移量为0；根据页表，第1 页被映射到第6 帧。这样，逻辑地址4 映射到物理地址24 (= (6 × 4) + 0)。逻辑地址13映射到物理地址9。



- Let’s imagine the process with that tiny address space (64 bytes) is performing a memory access:

```
movl <virtual address>, %eax
```

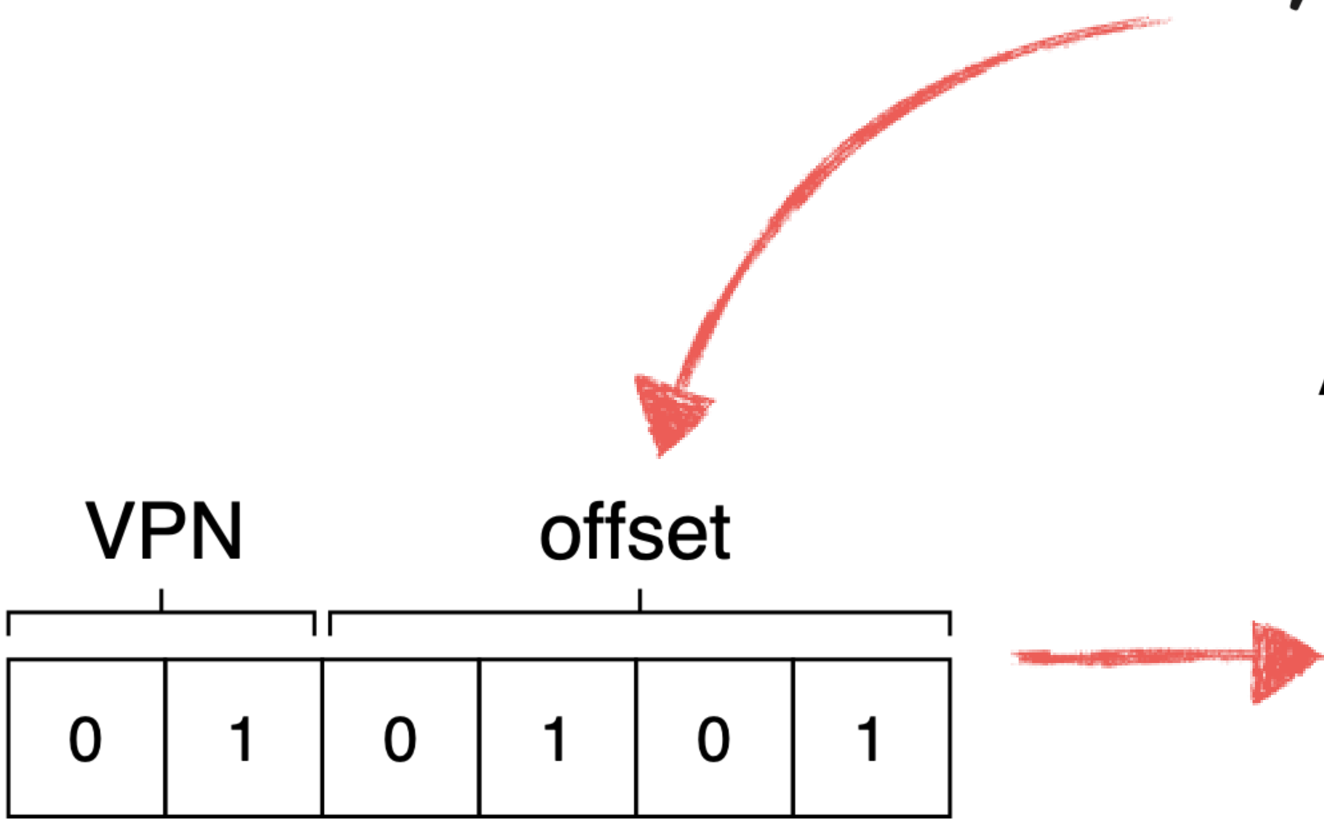
- To **translate** this virtual address, we have to first split it into two components: the **virtual page number (VPN)**, and the **offset** within the page.

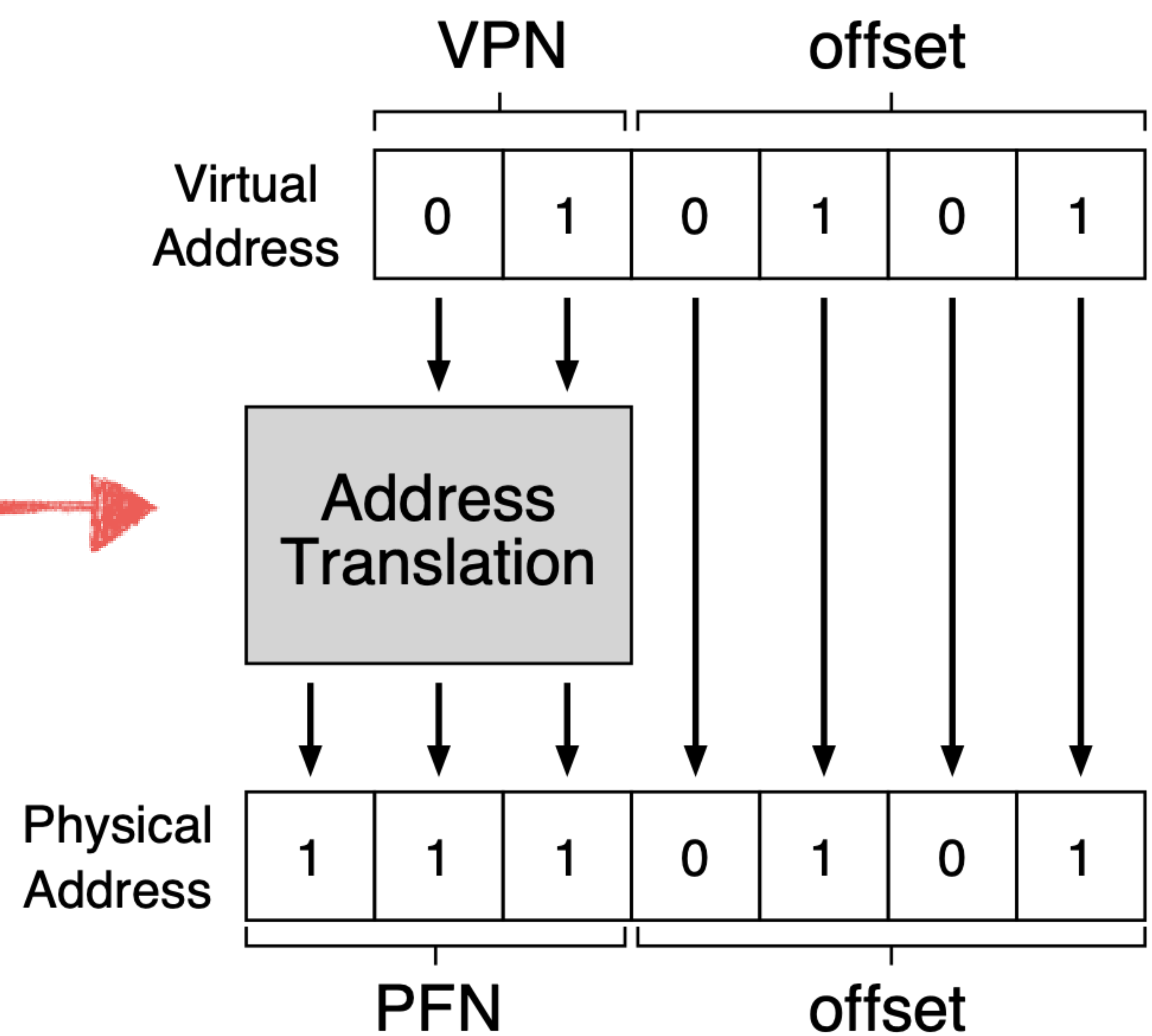
- For this example, because the virtual address space of the process is **64 bytes**, we need **6 bits** total for our virtual address (**$2^6 = 64$**). Thus, our virtual address:

Va5	Va4	Va3	Va2	Va1	Va0

- When a process generates a virtual address, the **OS** and **hardware** must combine to translate it into a meaningful physical address. For example:

`movl 21, %eax`





The Address Translation Process

- Note the **offset stays the same** (i.e., it is not translated), because the offset just tells us which byte **within** the page we want.

存储器的用户空间共有 32 个页面，每页 1KB，内存 16KB。假定某时刻系统为用户的第 0、1、2、3 页分别分配的物理块号为 5、10、4、7，试将逻辑地址 0A5C 和093C 变换为物理地址。

解：逻辑地址为：页号 5 位（2^5=32），页内位移 10 位（2^{10}=1024）；物理地址为：物理块号 4 位（2^4=16），块内位移（2^{10}=1024）10 位。

逻辑地址 0A5C 对应的二进制为：00010 1001011100，即逻辑地址 0A5C 的页号为 2，页内位移为 1001011100，由题意知对应的物理地址为：0100 1001011100 即 125C。

同理可求 093C 的物理地址为 113C。

Where Are Page Tables Stored?

- Page tables can get awfully large, much bigger than the small segment table or base/bounds pair we have discussed previously.

- For example, imagine a typical **32-bit** address space, with **4KB** pages. This virtual address splits into a **20-bit** VPN and **12-bit** offset (recall that 10 bits would be needed for a 1KB page size, and just add two more to get to 4KB).

- Estimate how much memory do we need to store the page table?

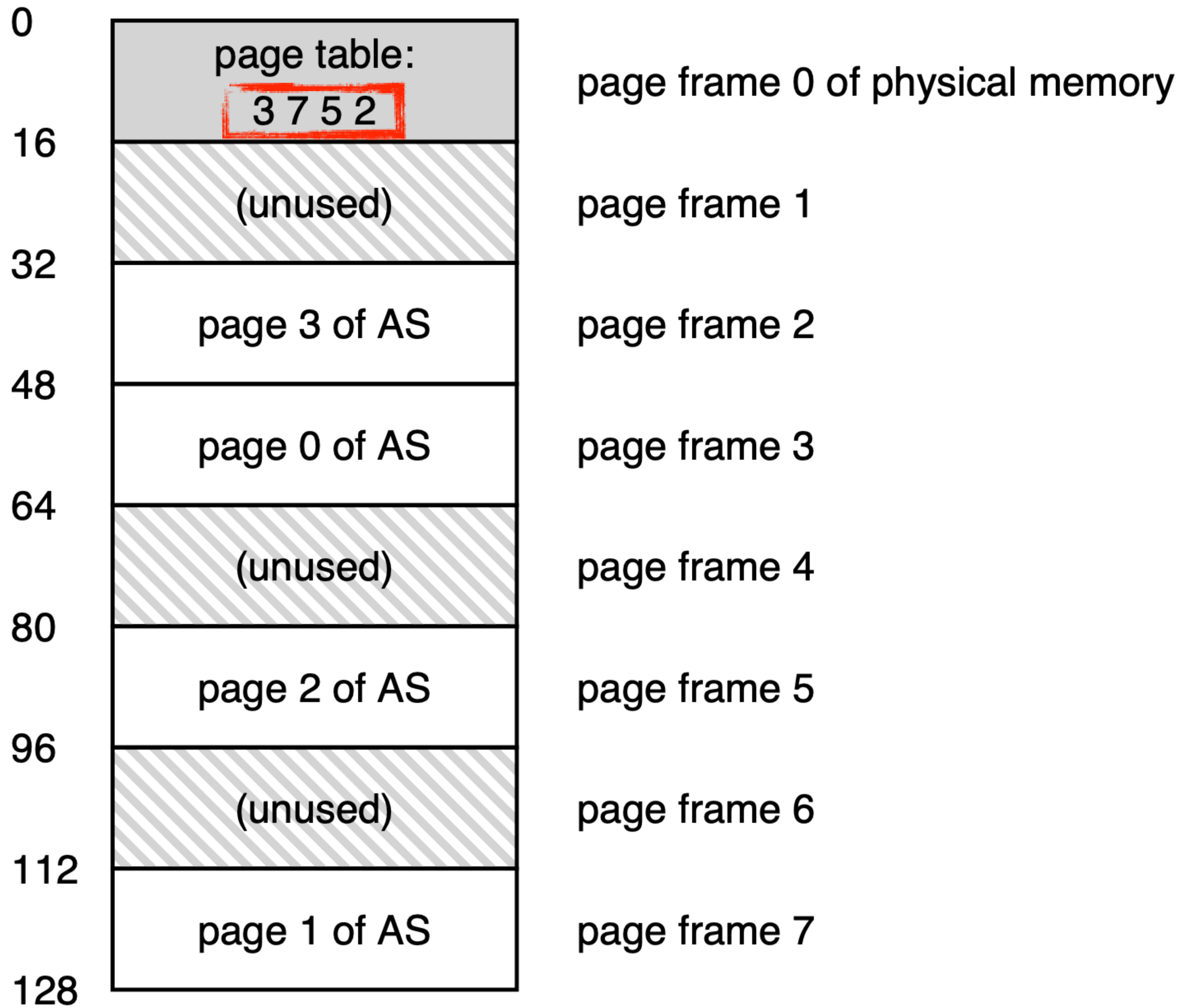
- A **20-bit** VPN implies that there are 2^{20} translations that the OS would have to manage for each process (that's roughly a million).

- Assuming we need **4 bytes** per **page table entry (PTE)** to hold the physical translation plus any other useful stuff, we get an immense **4MB** of memory needed for each page table! That is pretty big.

- Now imagine there are **100** processes running: this means the OS would need **400MB** of memory just for all those address translations!

- Because page tables are **so big**, we **don’t** keep any special **on-chip hardware** in the **MMU** to store the page table of the currently-running process.
- Instead, we store the page table for each process in memory somewhere.
- Let’s assume for now that the **page tables live in physical memory** that the OS manages.

由于页表如此之大，我们没有在 MMU 中利用任何特 殊的片上硬件来存储当前正在运行的进程的页表，而是将每个进程的页表存储在内存中。现在让我们假设页表存 在于操作系统管理的物理内存中，稍后我们会看到，很多 操作系统内存本身都可以虚拟化，因此页表可以存储在操 作系统的虚拟内存中（甚至可以交换到磁盘上）



Example: Page Table in Kernel Physical Memory

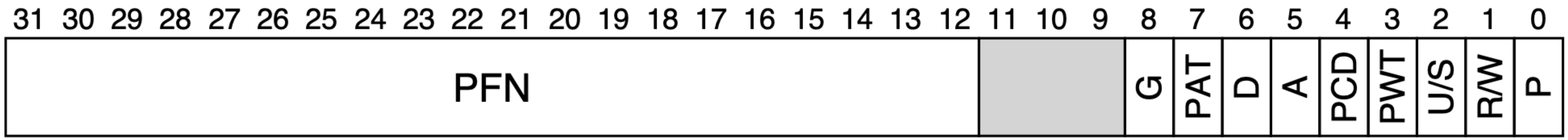
What’s Actually In The Page Table?

- The page table is just **a data structure** that is used to map virtual addresses (virtual page numbers) to physical addresses (physical page numbers).

- Thus, any data structure could work. The simplest form is called a **linear page table**, which is just an **array**.

- The OS **indexes** the array by the VPN, and looks up the **page-table entry (PTE)** at that index in order to find the desired PFN.

- As for the contents of each PTE, we have a number of **different bits** in there worth understanding at some level.



An x86 Page Table Entry (PTE)

- Read the **Intel Architecture Manuals** for more details on x86 paging support.

P: present
R/W: read/write bit
U/S: supervisor
A: accessed bit
D: dirty bit
PFN: the page frame number

- A **valid bit** is common to indicate whether the particular translation is valid.
    - 当一个程序开始运行时，它的代码和堆在其 地址空间的一端，栈在另一端。所有未使用的中间空间都将被标记为无效（invalid），如果进程尝试访问这种内存，就会陷入操作系统，可能会导致该进程终止。因此，有效位对于支持稀疏地址空间至关重要。通过简单地将地址空间中所有未使用的页面标记为无效，我 们不再需要为这些页面分配物理帧，从而节省大量内存。
- **protection bits**, indicating whether the page could be read from, written to, or executed from.
- A **present bit** indicates whether this page is in physical memory or on disk (**swapped out**).
- A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.
- A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed, and is useful in determining **which pages are popular** and thus should be kept in memory; such knowledge is critical during **page replacement**, a topic we will study in great detail later.

Paging: Also Too Slow

- With page tables in memory, we already know that they might be too big. Turns out they can slow things down too.

```
movl 21, %eax
```

- To fetch the desired data, the system must first translate the virtual address (**21**) into the correct physical address (**117**).

- Thus, before issuing the load to address 117, the system must **first** fetch the proper page table entry from the process’ s page table, perform the translation, and **then** finally get the desired data from physical memory.

- To do so, the hardware must know where the page table is for the process. Assume that a single **page-table base register** contains the physical address of the starting location of the page table. To find the location of the PTE, the hardware will perform the following functions:

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

In our example, `VPN_MASK` would be set to 0x30 (hex 30, or binary 110000) which picks out the VPN bits from the full virtual address; `SHIFT` is set to 4 (the number of bits in the offset), such that we move the VPN bits down to form the correct integer virtual page number. For example, with virtual address 21 (010101), and masking turns this value into 010000; the shift turns it into 01, or virtual page 1, as desired. We then use this value as an index into the array of PTEs pointed to by the page table base register.

- Once this physical address（PTEAddr） is known, the hardware can fetch the PTE（页表项） from memory, extract the PFN（页帧号）, and concatenate it with the offset from the virtual address to form the desired physical address.
- PTBR:页表基址寄存器

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

Accessing Memory With Paging

- For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform **one extra memory reference** in order to first fetch the translation from the page table.

- That is a lot of work! **Extra memory references are costly**, and in this case will likely slow down the process by a factor of two or more.

- Without careful design of both hardware and software, page tables will cause the system to run **too slowly**, as well as take up **too much memory**.

- While seemingly a great solution for our memory virtualization needs, these **two crucial problems** must first be overcome.

Event	Latency	Scaled
1 CPU Cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50 - 150 us	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1-3 s	105-317 years
OS virtualization system reboot	4 s	423 years
SCSI command timeout	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 min	32 millennia

补充：数据结构——页表

现代操作系统的内存管理子系统中最重要的数据结构之一就是页表（page table）。通常，页表存储虚拟—物理地址转换（virtual-to-physical address translation），从而让系统知道地址空间的每个页实际驻留在物理内存中的哪个位置。由于每个地址空间都需要这种转换，因此一般来说，系统中每个进程都有一个页表。页表的确切结构要么由硬件（旧系统）确定，要么由 OS（现代系统）更灵活地管理。

Summary

- Paging has many advantages over previous approaches (such as segmentation).
- First, it does not lead to external fragmentation, as paging (by design) divides memory into fixed-sized units.
- Second, it is quite flexible, enabling the sparse use of virtual address spaces.
- However, implementing paging support without care will lead to a **slower** machine as well as memory **waste**.