FDU 操作系统 2. 内存的虚拟化

本文参考以下教材：

- Operating Systems: Three Easy Pieces (R. H. Arpaci-Dusseau & A. C. Arpaci-Dusseau) Chapter $13 \sim 23$
- 操作系统: 三个简单的部分 (王海鹏 译) 第 $13 \sim 23$ 章

欢迎批评指正!

2.1 An Introduction

2.1.1 地址空间

早期的操作系统几乎没有提供内存的抽象.
那时候机器的物理内存看起来就像这样:

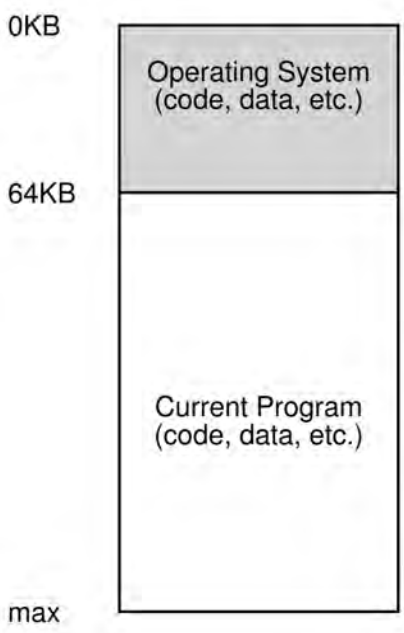


Figure 13.1: **Operating Systems: The Early Days**

一段时间之后，人们开始更有效地共享机器，多程序 (multiprogramming) 系统时代开启.
但很快，人们开始对机器要求更多，分时系统的时代便到来了.

一种实现时分共享的方法，是让一个进程单独占用全部内存运行一小段时间然后停止它，
并将它所有的状态信息保存在磁盘上 (包含所有的物理内存)，加载其他进程的状态信息，再运行一段时间.
这就实现了一种比较粗糙的机器共享.
但这个方法有个显著的问题——将全部的内存信息保存到磁盘太慢了.
因此在进程切换的时候，我们仍然将进程信息放在内存中，这样操作系统可以更有效率地实现时分共享 (如图所示)

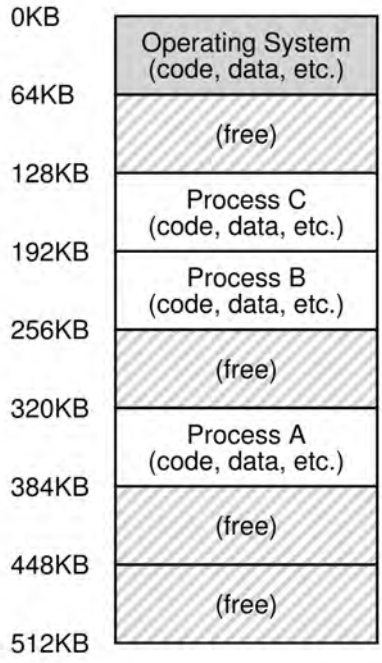


Figure 13.2: **Three Processes: Sharing Memory**

多个程序同时驻留在内存中，便使得**保护** (protection) 成为重要问题:
我们可不希望一个进程可以读取其他进程的内存，更别说修改了.
因此操作系统需要提供一个方便使用的物理内存抽象，即**地址空间** (address space)
使得运行的程序看到的是系统中的内存.

一个进程的地址空间包含运行的程序的所有内存状态，现在假设只有以下三个部分:

- ① 程序代码
- ② 栈 (stack): 用来保存当前的函数调用信息，分配空间给局部变量，传递参数和函数返回值
- ③ 堆 (heap): 用来管理动态分配的用户管理的内存

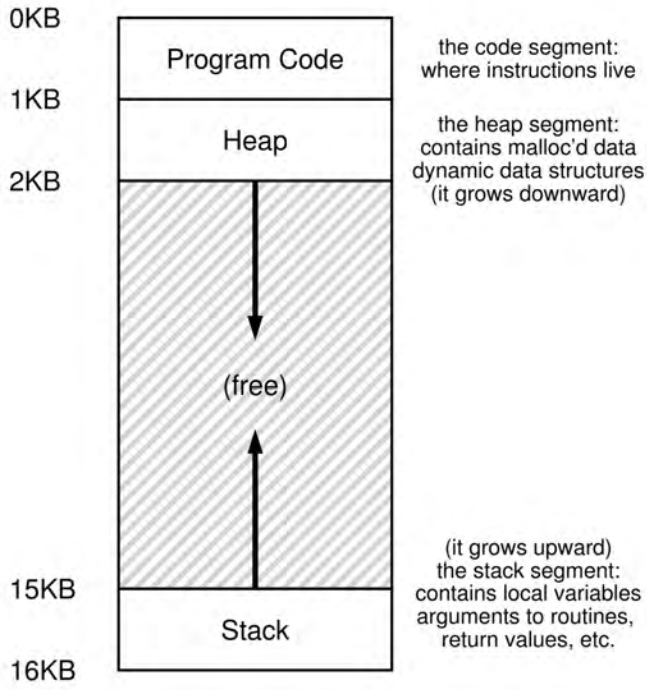


Figure 13.3: **An Example Address Space**

在程序运行时，地址空间有两个区域可能增长 (或者收缩)，它们就是堆 (在顶部) 和栈 (在底部)
(当然，栈和堆的这种放置方法只是一种约定，只要你愿意，就可以用不同的方式安排地址空间)
当用户通过 $\mathrm{malloc}()$ 请求更多内存时，堆会向下生长;
当用户进行程序调用时，栈会向上生长 (简单地移动栈指针)

Motivation for Dynamic Memory

- **Why do processes need dynamic allocation of memory?**
  - Do not know amount of memory needed at compile time
  - Must be pessimistic when allocate memory statically
    - Allocate enough for worst possible case; Storage is used inefficiently
- **Recursive procedures**
  - Do not know how many times procedure will be nested
- **Complex data structures: lists and trees**
  - `struct my_t *p = (struct my_t *) malloc(sizeof(struct my_t));`
- **Two types of dynamic allocation**
  - Stack
  - Heap

Stack Organization

- **Definition: Memory is freed in opposite order from allocation**

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

- **Simple and efficient implementation: Pointer separates allocated and freed space**
  - Allocate: Increment pointer
  - Free: Decrement pointer
- **No fragmentation**

Heap Organization

- **Definition: Allocate from any random location: malloc(), new()**
  - Heap memory consists of allocated areas and free areas (holes)
  - Order of allocation and free is unpredictable
- **Advantage**
  - Works for all data structures
- **Disadvantages**
  - Allocation can be slow
  - End up with small chunks of free space - fragmentation
  - Where to allocate 12 bytes? 16 bytes? 24 bytes??
- **What is OS's role in managing heap?**
  - OS gives big chunk of free memory to process; library manages individual allocations

事实上，当我们描述地址空间时，所描述的是操作系统提供给运行程序的抽象.
在上述例子中，程序不在物理地址 $0 \sim 16\mathrm{KB}$ 的内存中，而是加载在任意的物理地址.
如果你在 C 程序中打印地址，那就是一个虚拟地址.
它只提供地址在内存中分布的假象，只有操作系统 (或硬件) 才知道物理地址.

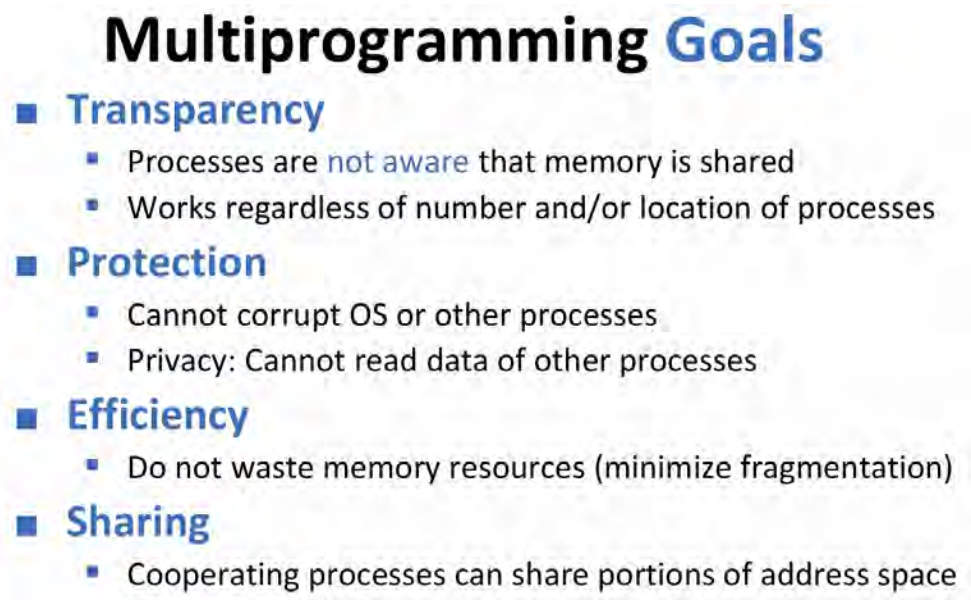
当操作系统这样做时，我们说操作系统在**虚拟化内存** (virtualizing memory)
因为运行的程序认为它被加载到特定地址 (例如 0) 的内存中，并且具有非常大的地址空间，但实际情况很不一样.

2.1.2 内存虚拟化的目标

虚拟内存 (virtual memory, VM) 系统的主要目标如下:

- **① 透明性 (transparency)**
  操作系统实现虚拟内存的方式对正在运行的程序来说是不可见的.
  换言之，程序不应该感知到内存被虚拟化的事实，相反，程序的行为就好像它拥有自己的私有物理内存.
  在幕后，操作系统 (和硬件) 完成了所有的工作，让不同的工作复用内存，从而实现这个假象.
- **② 效率 (efficiency)**
  操作系统应该追求虚拟化尽可能高效
  无论是在时间上 (即不会使程序运行得更慢) 还是在空间上 (即不需要太多额外的内存来支持虚拟化)
  在实现高效率虚拟化时，操作系统将不得不依靠硬件支持，例如 TLB 这样的硬件功能.
- **③ 保护 (protection)**
  操作系统应确保进程及其自身受到保护，不会受其他进程影响.
  当一个进程执行内存读写操作时，它不应该访问或影响任何其他进程或操作系统的内存内容 (即其地址空间之外的任何内容)

因此保护让我们能够在进程之间提供隔离 (isolation) 的特性.
换言之，每个进程都应该在自己的独立环境中运行，避免出错或恶意进程的影响.



2.2 内存操作 API

2.2.1 内存类型

在运行一个 C 程序的时候，会分配两种类型的内存.

- 第一种称为**栈内存**，它的申请和释放操作是编译器来隐式管理的，所以有时也称为**自动内存**.
  例如，假设需要在 $\mathrm{func}()$ 函数中为一个整型变量 x 申请空间
  为了声明这样的一块内存，只需要这样做:

```
void func()
{
    int x; // declares an integer on the stack
    ...
}
```

  编译器完成剩下的事情，确保在你进入 $\mathrm{func}()$ 函数的时候，在栈上开辟空间.
  当你从该函数退出时，编译器释放内存.
  因此如果你希望某些信息存在于函数调用之外，请不要将它们放在栈上.
- 正是这种对长期内存的需求，我们才需要第二种类型的内存，
  即所谓的**堆内存**，其所有的申请和释放操作都由程序员显式地完成.
  这可能会导致很多缺陷，但如果正确地使用这些接口，就没有太多的麻烦.
  下面的例子展示了如何在堆上分配一个整数，得到指向它的指针:

```
void func()
{
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

  首先当编译器看到指针的声明 `(int *x)` 时，就知道为一个整型指针分配空间.
  随后当程序调用 $\mathrm{malloc}()$ 时，它会在堆上请求整数的空间，
  如果成功，则返回这个整数的地址 (否则返回 `NULL`) 并将其存储在栈中以供程序使用.

2.2.2 malloc() & free()

malloc() 函数非常简单:
传入要申请的堆空间的大小，若成功则返回一个指向申请空间的 void 类型指针，否则返回 `NULL`
(在 C 语言中，`NULL` 就是一个值为 0 的宏)
只需包含头文件 stdlib.h 就可以使用 malloc() 了 (实际上甚至都不需要这么做)

malloc() 函数只需要一个 size_t 类型参数，表示你需要多少个字节.

- 我们通常使用各种函数 (例如 sizeof) 和宏给这个参数赋值，例如为给双精度浮点数分配空间，只要这样:

  ```
  double *d = (double *) malloc(sizeof(double));
  ```

- 你也可以传入一个变量的名字 (而不只是类型) 给 sizeof()，但有时可能得不到你要的结果，所以要小心使用.
  例如下面的例子在 64 位机器上输出的是 8 (一个整型数据的字节数) 而不是 80 (十个整型数据的字节数):

  ```
  int *x = malloc(10 * sizeof(int));
  printf("%d\n", sizeof(x));
  ```

  原因是在这种情况下，sizeof() 认为我们只是问一个整型数据的指针有多大，而不是我们动态分配了多少内存.
  不过有时 sizeof() 的输出符合我们的期望:

  ```
  int x[10];
  printf("%d\n", sizeof(x));
  ```

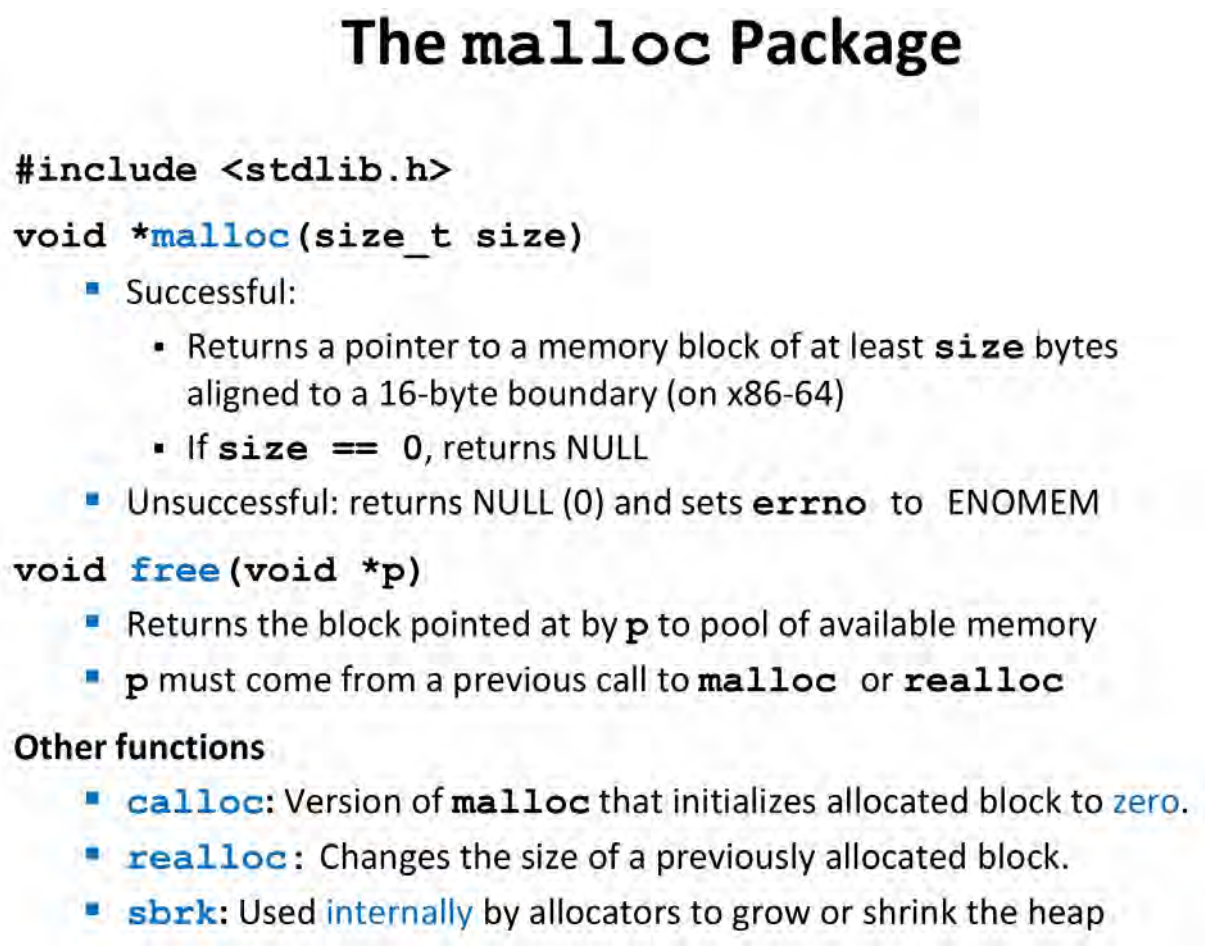
  在这种情况下，编译器有足够的静态信息，知道已经分配了 40 个字节.

- 另一个需要注意的地方是使用字符串.
  若要为一个字符串声明空间，请使用 `malloc(strlen(s) + 1)`
  它使用函数 strlen() 获取字符串 s 的长度，并加上 1，以便为字符串结束符留出空间.
  这里使用 sizeof() 可能会导致麻烦.

要释放不再使用的堆内存，程序员只需调用 free():

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

该函数接受一个参数，即一个由 malloc() 返回的指针.
值得注意的是，该指针对应的分配区域大小不会被用户传入，必须由内存分配库本身记录追踪.



2.2.3 常见错误

在使用 malloc() 和 free() 时会出现一些常见的错误.
下面所有的例子都可以通过编译器的编译并运行.
对于构建一个正确的 C 程序来说，通过编译是必要的，但还远远不够 (你会懂的, 通常在吃了很多苦头之后)

- **(忘记分配内存)**
  很多例程在被调用之前，都希望你为它们分配内存.
  例如例程 `strcpy(dst, src)` 将源字符串中的字符串复制到目标指针.
  但一不小心，你就可能会这样做:

  ```
  char *src = "hello";
  char *dst; // oops! unallocated
  strcpy(dst, src); // segfault and die
  ```

  运行这段代码时，可能会导致**段错误** (segmentation fault)
  在这个例子中，正确的代码可能像这样:

  ```
  char *src = "hello";
  char *dst = (char *) malloc(strlen(src) + 1);
  strcpy(dst, src); // work properly
  ```

  或者你可以直接用 strdup()，让会让你活得更加轻松.

- **(没有分配足够的内容)**
  另一个相关的错误是没有分配足够的内存，有时称为**缓冲区溢出** (buffer overflow)，例如:

  ```
  char *src = "hello";
  char *dst = (char *) malloc(strlen(src)); // too small!
  strcpy(dst, src); // work properly
  ```

奇怪的是，这个程序通常看起来会正确运行 (但有时该程序确实会发生崩溃)
这是因为 $\mathrm{malloc}()$ 总是分配一些额外的空间，使得你的程序不会覆写其他变量.

- **(忘记初始化分配的内存)**
  在这个错误中，你正确地调用 $\mathrm{malloc}()$，但忘记在新分配的数据类型中填写一些值.
  你的程序最终会遇到**未初始化的读取** (uninitialized read)
  它从堆中读取了一些未知值的数据，天知道那里可能会有什么?
  实际上你可以直接使用 $\mathrm{calloc}()$ 分配内存，它会在返回指针之前将分配的内存初始化为零.
- **(忘记释放内存)**
  如果忘记释放内存，就会发生**内存泄露** (memory leak)
  缓慢泄露的内存最终会导致内存不足，需要通过重新启动来解决.
  一般来说，当你用完一段内存时，应该确保释放它.
  值得注意的是，使用垃圾收集语言在这里没有什么帮助.
  如果你仍然拥有对某块内存的引用，那么垃圾收集器就不会释放它.
  因此即使在较现代的语言中，内存泄露仍然是一个问题.
  程序员的良好习惯之一便是释放显式分配的每个字节.

  > 对于短时间运行的程序，泄露内存通常不会导致任何操作问题 (尽管它可能被认为是不好的形式)
  > 即使你没有调用 $\mathrm{free}()$ (并因此泄露了堆中的内存)
  > 操作系统也会在程序结束运行时回收进程的所有内存 (包括用于代码、栈，以及相关堆的内存页)
  > 无论地址空间中堆的状态如何，操作系统都会在进程终止时收回所有这些页面，
  > 从而确保即使没有释放内存，也不会丢失内存.
  >
  > 如果你编写一个长期运行的程序 (例如 Web 服务器或数据库管理系统)
  > 泄露内存就是很大的问题，最终会导致应用程序在内存不足时崩溃.

- **(在用完之前释放内存)**
  有时候程序会在用完之前释放内存，这种错误称为**悬挂指针** (dangling pointer)
  随后的使用可能会导致程序崩溃或覆盖有效的内存
  (例如错误调用了 $\mathrm{free}()$ 之后再次调用 $\mathrm{malloc}()$ 来分配其他内容，这会重新利用先前错误释放的内存)
- **(反复释放内存)**
  程序有时还会不止一次地释放内存，这被称为**重复释放** (double free)
  这样做的结果是未定义的——内存分配库可能会感到困惑，并且会做各种奇怪的事情，崩溃是常见的结果.
- **(无效的释放)**
  $\mathrm{free}()$ 函数只期望你传入你先前从 $\mathrm{malloc}()$ 得到的一个指针.
  如果你传入了其他的值，则会产生**无效的释放** (invalid free)，这无疑是很危险的.

2.3 地址转换

在实现 CPU 虚拟化时，我们遵循的一般准则被称为**受限直接执行** (Limited Direct Execution, LDE)
即让程序运行的大部分指令直接访问硬件，只在一些关键点 (如进程发起系统调用或发生时钟中断时)
由操作系统介入 (interposing) 来确保 "在正确时间，正确的地点，做正确的事"
在实现虚拟内存时，我们将追求类似的战略，在实现高效和控制的同时，提供期望的虚拟化.

- **高效**决定了我们要利用硬件的支持，这在开始的时候非常初级 (如使用一些寄存器)，
  但逐渐会变得相当复杂 (例如我们会讲到的 TLB 和页表等)
- **控制**意味着操作系统要确保应用程序只能访问它自己的内存空间.
- 最后我们对虚拟内存还有一点要求，即**灵活性**.
  具体来说，我们希望程序能以任何方式访问它自己的地址空间，从而让系统更容易编程.

我们将使用**基于硬件的地址转换** (hardware-based address translation)，简称为**地址转换**.
它可以看成是受限直接执行的补充.
利用地址转换，硬件对每次访存操作进行处理，将指令中的虚拟地址转换为实际存储的物理地址.

当然，仅仅依靠硬件不足以实现虚拟内存，因为它只是提供了底层机制来提高效率.
操作系统必须在关键的位置介入，设置好硬件，以便完成正确的地址转换.
因此它必须管理内存，记录被占用和空闲的内存位置，并明智而谨慎地介入，保持对内存使用的控制.

所有这些工作都是为了创造一种美丽的假象: 每个程序都拥有私有的内存，那里存放着它自己的代码和数据.
实际上许多程序其实是在同一时间共享着内存，就像 CPU 在不同的程序间切换运行.
通过虚拟化， 操作系统 (在硬件的帮助下) 将丑陋的机器现实转化成一种易于使用的抽象.

2.3.1 一个例子

我们先假设用户的地址空间必须连续地放在物理内存中，且地址空间小于物理内存的大小.
并假设每个地址空间的大小完全一样.
(我们会逐步地放宽这些假设，从而得到现实的内存虚拟化)

考虑一小段 C 代码:

```
void func()
{
    int x = 3000; // thanks, Perry.
    x = x + 3; // line of code we are interested in
    ...
}
```

编译器将其转为如下的 x86 汇编:

```
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
```

假设 x 的地址已经存入寄存器 ebx 了
上述汇编指令将这个地址的值 (即 x) 加载到通用寄存器 eax，加 3 后写回内存的同一位置.

设该进程的地址空间如下:
(3 条指令序列从地址 128B 开始，变量 x 位于地址 15KB 处，且初始值为 3000)

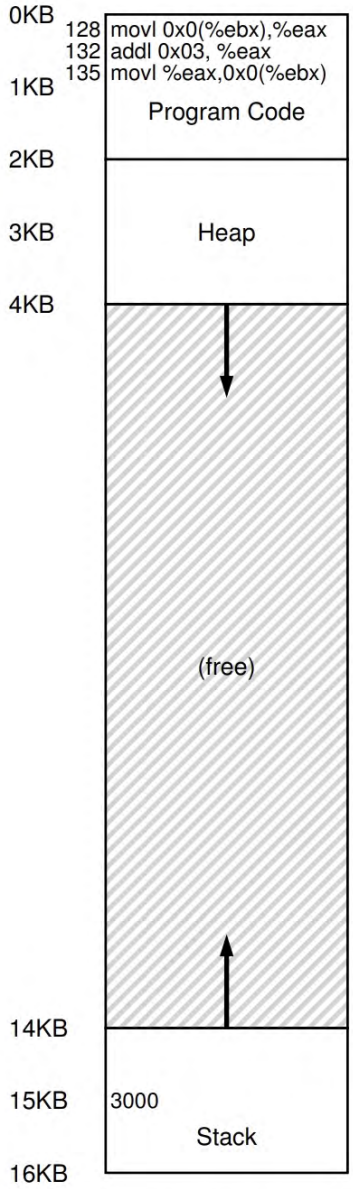


Figure 15.1: **A Process And Its Address Space**

如果这 3 条指令执行，则从进程的角度来看，发生了以下几次内存访问:

- 从地址 128 获取指令 `movl 0x0(%ebx), %eax`
- 执行指令 (从地址 15KB 加载数据 $x = 3000$)
- 从地址 132 获取命令 `addl $0x03, %eax`
- 执行命令 (没有内存访问，$3000 + 3 = 3003$)
- 从地址 135 获取指令 `movl %eax, 0x0(%ebx)`
- 执行指令 (将新值 3003 存入地址 15KB)

从程序的角度来看，它的地址空间从 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.
然而对虚拟内存来说，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.
下图展示了这个进程的地址空间被放入物理内存后可能的样子:
(上述例子中的进程地址空间被重定位到了从 32KB 开始的物理内存地址)

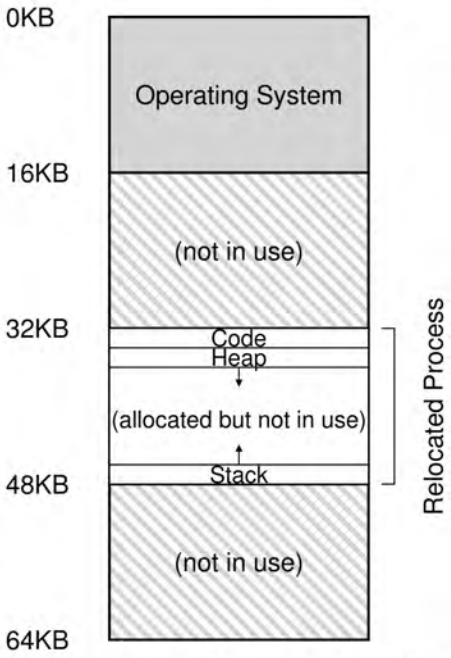


Figure 15.2: **Physical Memory with a Single Relocated Process**

Static Relocation

- Idea: OS rewrites each program before loading it as a process in memory
- Each rewrite for different process uses different addresses and pointers
- Change jumps, loads of static data

- 0x10: movl 0x8(%rbp), %edi
- 0x13: addl $0x3, %edi
- 0x19: movl %edi, 0x8(%rbp)

rewrite →
0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)

rewrite →
0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)

Disadvantages

- **No protection**
  - Process can destroy OS or other processes
  - No privacy
- **Cannot move address space after it has been placed**
  - May not be able to allocate new process
  - Need to be the same place for being switched back

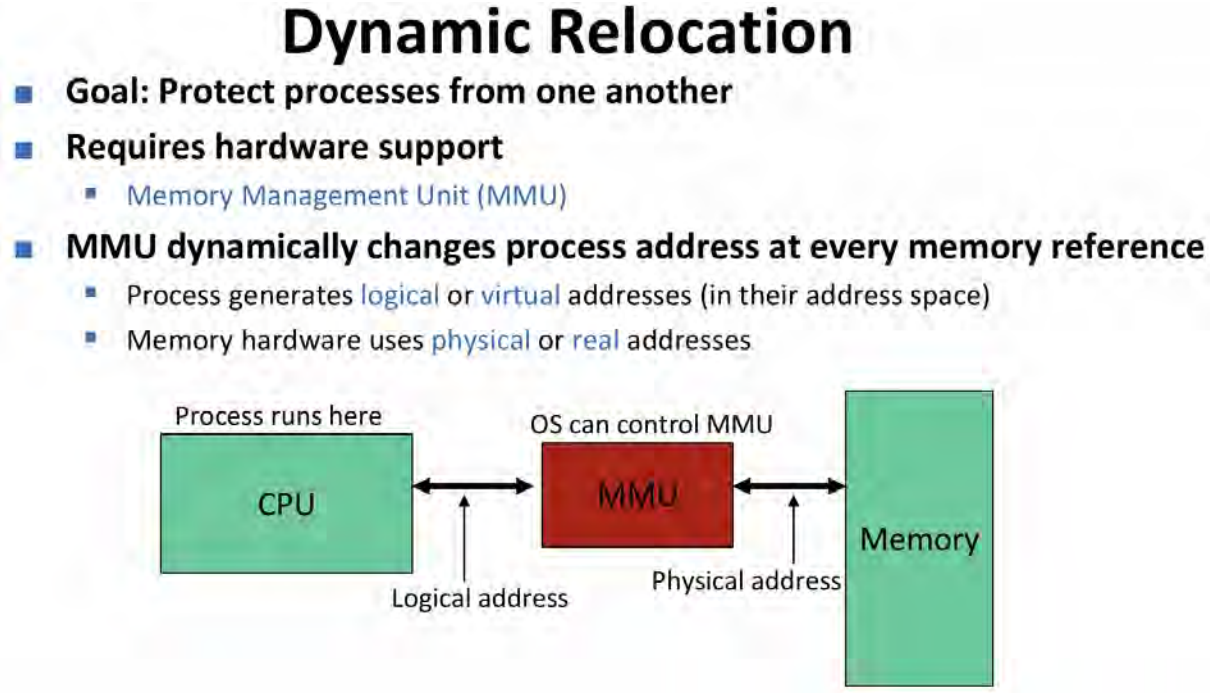
2.3.2 动态重定位

基于硬件的地址转换的一种方法便是**基址加界限机制** (base and bound)，又称为**动态重定位** (dynamic relocation)
具体来说，CPU 需要两个硬件寄存器: **基址寄存器**和**界限寄存器**.
它们让我们能够将地址空间放在物理内存的任何位置，同时又能确保进程只能访问自己的地址空间.

采用这种方式时，在编写和编译程序时地址空间从零开始.
而当程序真正执行时，操作系统会决定其在物理内存中的实际加载地址，并将起始地址记录在基址寄存器中.
(在 2.3.1 的例子中基址寄存器的值为 32KB)
当进程运行时，该进程产生的所有内存访问都会被 CPU 转换为物理地址:

$$\text{physical\_address} = \text{virtual\_address} + \text{base}$$

换言之，硬件取得进程认为它要访问的地址 (虚拟地址)，并将其转换成物理地址.
由于这种重定位是在运行时发生的，而且我们甚至可以在进程开始运行后改变其地址空间，
故这种技术一般被称为**动态重定位**.



而界限寄存器提供了访问保护 (在 2.3.1 的例子中界限寄存器为 16KB)
如果进程需要访问的虚拟地址超过这个界限或者为负数，则 CPU 将触发异常，该进程会被终止.
这个检查是在将虚拟地址与基址寄存器内容求和前进行的.
(另一种实现方式是界限寄存器中记录地址空间结束的物理地址，硬件在转化虚拟地址到物理地址之后才去检查这个界限.
这两种方式在逻辑上是等价的，简单起见，我们这里假设采用第一种方式)

这种基址寄存器配合界限寄存器的硬件结构统称为**内存管理单元** (Memory Management Unit, MMU)
随着我们开发更复杂的内存管理技术，MMU 也将有更复杂的电路和功能.

2.3.3 硬件支持

下表总结了动态重定位所需的硬件支持:

表 15.2　　动态重定位：硬件要求

硬件要求	解释
特权模式	需要，以防用户模式的进程执行特权操作
基址/界限寄存器	每个 CPU 需要一对寄存器来支持地址转换和界限检查
能够转换虚拟地址并检查它是否越界	电路来完成转换和检查界限，在这种情况下，非常简单
修改基址/界限寄存器的特权指令	在让用户程序运行之前，操作系统必须能够设置这些值
注册异常处理程序的特权指令	操作系统必须能告诉硬件，如果异常发生，那么执行哪些代码
能够触发异常	如果进程试图使用特权指令或越界的内存

- 首先我们需要两种 CPU 模式.
  操作系统在内核模式下可以访问整个机器资源.
  应用程序在用户模式下运行，只能做有限的操作.

只要处理器状态字的一个位，就能说明当前的 CPU 的运行模式.
在一些特殊的时刻 (例如系统调用、异常或中断)，CPU 会切换状态.

- 硬件还必须提供 CPU 的内存管理单元 MMU (由基址寄存器和界限寄存器构成).
  用户程序运行时，MMU 会转换每个地址，即将用户程序产生的虚拟地址加上基址寄存器的内容.
  硬件也必须能检查地址是否有用，通过界限寄存器和 CPU 内的一些电路来实现.
- 硬件应该提供一些特殊的指令，用于修改基址寄存器和界限寄存器，允许操作系统在切换进程时改变它们.
  这些指令应当是特权指令，以保证只有在内核模式下才能修改这些寄存器.
- 最后，在用户程序尝试非法访问内存 (越界访问) 时，CPU 必须能够产生异常.
  在这种情况下，CPU 应该阻止用户程序的执行，并安排操作系统的 "越界" 异常处理程序去处理.
  操作系统的处理程序会做出正确的响应，例如直接终止进程.
  类似地，当用户程序尝试修改基址或者界限寄存器时，CPU 也应该产生异常，
  并调用 "用户模式尝试执行特权指令" 的异常处理程序.
  CPU 还必须提供一种方法，来通知它这些处理程序的位置，因此又需要另一些特权指令.

Hardware Support for Dynamic Relocation

- **Two operating modes**
  - **Privileged (protected, kernel) mode: OS runs**
    - When enter OS (trap, system calls, interrupts, exceptions)
    - Allows certain instructions to be executed
      - Can manipulate contents of MMU
    - Allows OS to access all of physical memory
  - **User mode: User processes run**
    - Perform translation of logical address to physical address

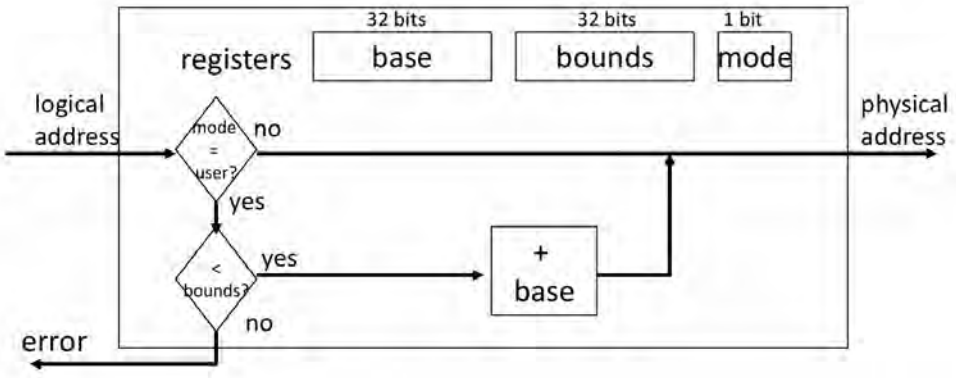
Dynamic with Base+Bounds

- **Idea: limit the address space with a bounds register**
- **Base register: smallest physical addr (or starting location)**
- **Bounds register: size of this process's virtual address space**
  - Sometimes defined as largest physical address (base + size)
- **OS kills process if process loads/stores beyond bounds**

MMU Implementation

Translation on every memory access of user process

- MMU compares logical address to bounds register
  - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address



2.3.4 软件支持

为支持动态重定位，操作系统需要实现以下功能:

- 第一，当进程被创建时，操作系统必须在物理内存中的位置为进程的地址空间安排一个位置.
  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，故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，标记空闲或已用.
  当新进程创建时，操作系统检索**空闲列表** (free list)，为新地址空间找到位置，并将其标记为已用.

> 操作系统必须记录哪些空闲内存没有使用，以便能够为进程分配内存.
> 很多的数据结构可以用于这项任务，其中最简单的是空闲列表.
> 它就是一个列表， 记录当前没有使用的物理内存的范围.

- 第二，当进程正常退出或被强制终止时，操作系统必须回收它的所有内存.
  操作系统会将这些内存放回到空闲列表，并根据需要清除相关的数据结构.
- 第三，在上下文切换时，操作系统必须保存和恢复基址和界限寄存器.
  具体来说，当操作系统决定中止当前的运行进程时，
  它将当前基址和界限寄存器中的内容保存某种每个进程都有的结构中，
  例如**进程结构** (process structure) 或**进程控制块** (Process Control Block，PCB)
  当操作系统恢复执行 (或首次执行) 某个进程时，也必须给基址和界限寄存器设置正确的值.

  值得注意的是，当进程停止时，操作系统可以改变其地址空间的物理位置.
  操作系统首先让进程停止运行，然后将地址空间拷贝到新位置，最后更新进程结构中保存的基址，指向新位置.
- 第四，操作系统必须提供**异常处理程序** (exception handler) 及 2.3.3 节中提到的一些特权指令.

表 15.3　　动态重定位：操作系统的职责

操作系统的要求	解释
内存管理	需要为新进程分配内存 从终止的进程回收内存 一般通过空闲列表（free list）来管理内存
基址/界限管理	必须在上下文切换时正确设置基址/界限寄存器
异常处理	当异常发生时执行的代码，可能的动作是终止犯错的进程

下表按时间线展示了一个例子:

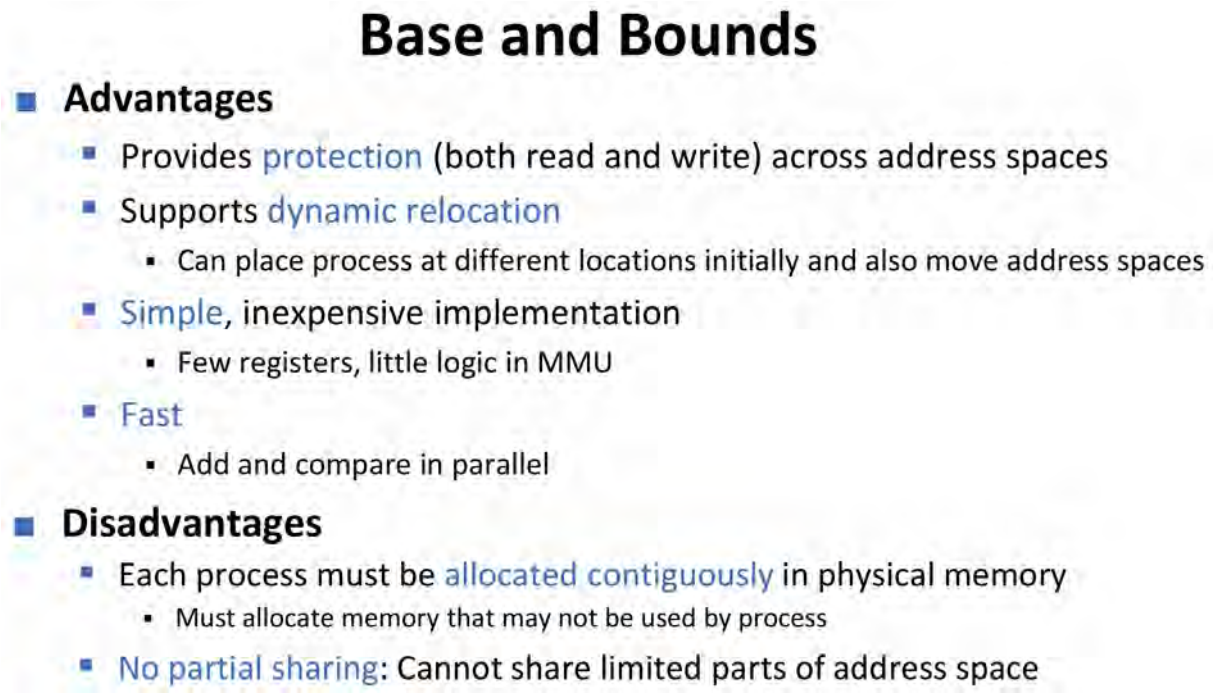
表 15.4　　受限直接执行协议（动态重定位）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 　系统调用处理程序 　时钟处理程序 　非法内存处理程序 　非常指令处理程序	
开始中断时钟		
	开始时钟，在 x ms 后中断	
初始化进程表 初始化空闲列表		
操作系统@运行（核心模式）	硬件	程序（用户模式）
为了启动进程 A： 　在进程表中分配条目 　为进程分配内存 　设置基址/界限寄存器 　从陷阱返回（进入 A）		
	恢复 A 的寄存器 转向用户模式 跳到 A（最初）的程序计数器	
		进程 A 运行 　获取指令
	转换虚拟地址并执行获取	
		执行指令
	如果显式加载/保存 　确保地址不越界 　转换虚拟地址并执行 　加载/保存	
		……
	时钟中断 转向内核模式 跳到中断处理程序	
处理陷阱 调用 switch()例程 　将寄存器（A）保存到进程结构（A） 　（包括基址/界限） 　从进程结构（B）恢复寄存器（B） 　（包括基址/界限） 从陷阱返回（进入 B）		
	恢复 B 的寄存器 转向用户模式 跳到 B 的程序计数器	
		进程 B 运行 　执行错误的加载
	加载越界 转向内核模式 跳到陷阱处理程序	

处理本期报告 决定终止进程 B 回收 B 的内存 移除 B 在进程表中的条目		

2.4 分段

遗憾的是，2.3 节所描述的简单的动态重定位技术有效率低下的问题.
由于进程地址空间的栈区和堆区并不很大，故这块内存区域中有大量的空间被浪费.
这种浪费通常称为**内部碎片** (internal fragmentation)，指的是已经分配的内存单元内部有未使用的空间，造成了浪费.



如果我们要求将地址空间完整地存放在固定大小的槽块中，那么内部碎片的出现是不可避免的.
因此我们需要更复杂的机制，以便更好地利用物理内存，避免出现内部碎片.
我们的第一次尝试便是将基址/界限的概念稍稍泛化，得到**分段** (segmentation) 的概念.

2.4.1 分段

分段的基本思想很简单，即在内存管理单元 MMU 中为地址空间内的每个**段**各引入一对基址/界限寄存器对.
一个段只是地址空间里的一个连续定长的区域.
在典型的地址空间里有 3 个逻辑不同的段: 代码、栈和堆.
分段的机制使得操作系统能够将不同的段放到不同的物理内存区域，
从而避免了虚拟地址空间中的未使用部分占用物理内存.

考虑以下地址空间.
通过一组 3 对基址/界限寄存器，我们可以将每个段独立地放入物理内存.

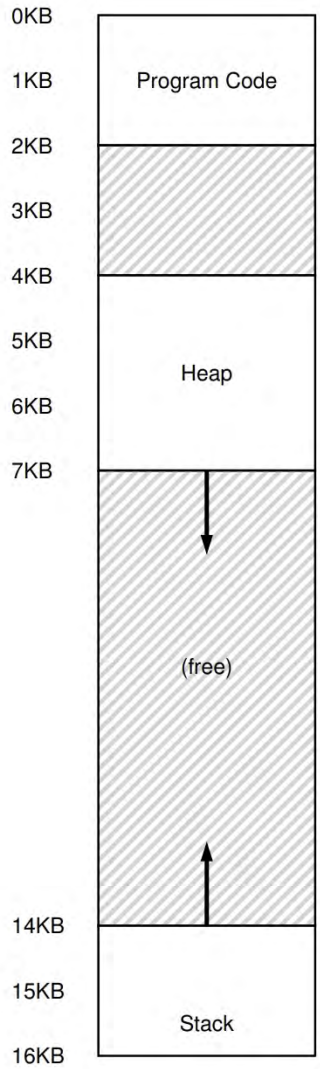


Figure 16.1: **An Address Space (Again)**

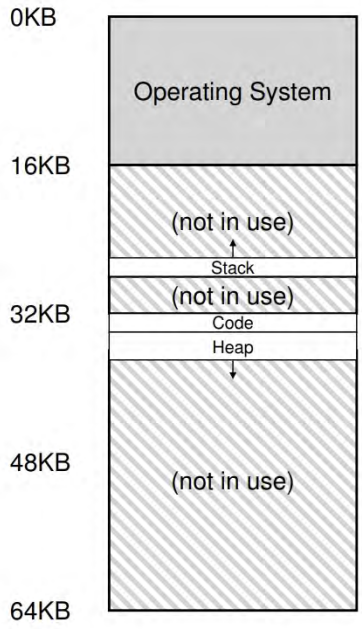


Figure 16.2: **Placing Segments In Physical Memory**

Segment	Base	Size (max 4K)	Grows Positive?
Code_{00}	32K	2K	1
Heap_{01}	34K	3K	1
Stack_{11}	28K	2K	0

Figure 16.4: **Segment Registers (With Negative-Growth Support)**

我们来看几个地址转换的例子:

- 假设现在要引用虚拟地址 100B (在代码段中)，它相对于代码段的虚拟基址 0 的偏移量为 100B
  MMU 会检查偏移量是否在界限内 (100B 小于 2KB)
  发现是的，便将基址值加上偏移量 (100B) 得到实际的物理地址: $32\text{KB} + 100\text{B} = 32868\text{B}$
  然后发起对物理地址 32868B 的引用.
- 假设现在要引用虚拟地址 4200B (在堆段中)，它相对于堆段的虚拟基址 4KB 的偏移量为 104B
  MMU 会检查偏移量是否在界限内 (104B 小于 3KB)
  发现是的，便将基址值加上偏移量 (104B) 得到实际的物理地址: $34\text{KB} + 104\text{B} = 34920\text{B}$
  然后发起对物理地址 34920B 的引用.
- (值得注意的是，栈是反向增长的，因此其地址转换与代码段和堆段不同)
  假设现在要引用虚拟地址 15KB (在堆段中)，它相对于栈段的虚拟基址 16KB 的偏移量为 -1KB
  MMU 会检查偏移量是否在界限内 (-1KB 大于 -2KB)
  发现是的，便将基址值加上偏移量 (-1KB) 得到实际的物理地址: $28\text{KB} - 1\text{KB} = 27\text{KB}$
  然后发起对物理地址 27KB 的引用.
- 如果我们试图访问非法的地址，例如 7KB (在地址空间的未分配段中)
  那么硬件会发现该虚拟地址越界，因此陷入操作系统，执行异常处理程序.
  这便是**段错误** (Segmentation Fault)，即在支持分段的机器上发生了非法的内存访问
  (有趣的是，在不支持分段的机器上，这个术语依然保留)

Segmentation

- Divide address space into logical segments
  - Each segment corresponds to logical entity in address space
    - code, stack, heap
- Each segment can independently:
  - be placed separately in physical memory
  - grow and shrink
  - be protected (separate read/write/execute protection bits)

Segmented Addressing

- Process now specifies base and offset within segment
- How does process designate a particular segment?
  - Use part of logical address
    - Top bits of logical address select segment
    - Low bits of logical address select offset within segment
- What if small address space, not enough bits?
  - Implicitly by type of memory reference
  - Special registers
- MMU contains Segment Table (per process)
  - Each segment has own base and bounds, protection bits
  - Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

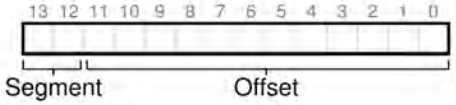
Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

remember:
1 hex digit->4 bits

2.4.2 段的区分

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?

一种常见的实现方式称为**显式方式**，即用虚拟地址的开头几位来标识不同的段 (及其生长方向).
假设 16KB 地址空间有 3 个段 (代码、堆和栈)，则需要两位来标识，此时虚拟地址如下所示:



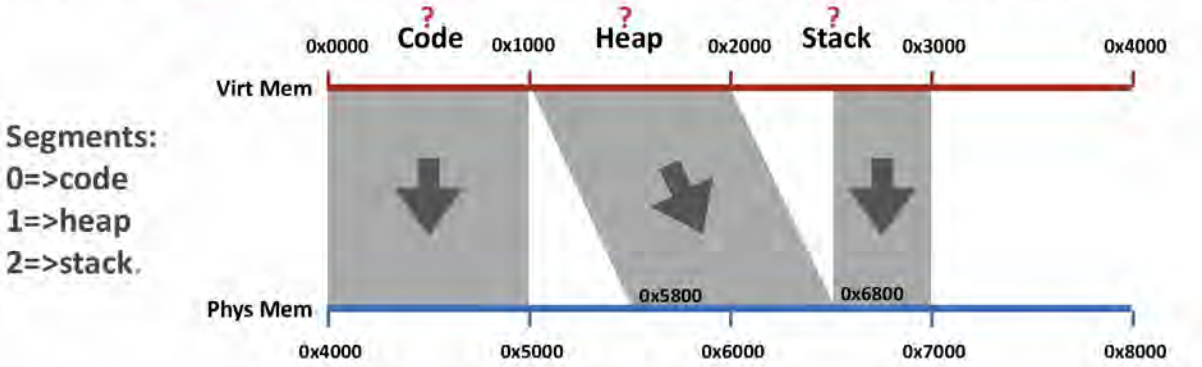
- 如果前两位是 00，那么硬件就知道这是属于代码段的虚拟地址，并使用代码段的基址和界限来重定位到正确的物理地址.
  (由于第一位是 0，故它的地址正向生长)

- 如果前两位是 01，则是堆地址，使用堆段的基址和界限来重定位到正确的物理地址.
  (由于第一位是 0，故它的地址正向生长)
- 如果前两位是 10，则是栈地址，使用栈段的基址和界限来重定位到正确的物理地址.
  (由于第一位是 1，故它的地址反向生长)

假设基址和界限放在数组中，为获取所需的物理地址，硬件会做以下事情:
(对于本例来说，$\text{SEG\_MASK} = 0\text{x}3000$, $\text{SEG\_SHIFT} = 12$, $\text{OFFSET\_MASK} = 0\text{xFFF}$)

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT;
// now get offset
Offset = VirtualAddress & OFFSET_MASK;

// Check if segment is valid (i.e., not 2'b11)
if (Segment == 2'b11) begin
    // Raise an exception if the segment is undefined
    RaiseException(SEGMENT_FAULT);
end
else if (Offset >= Bounds[Segment]) begin
    // Raise an exception if the offset exceeds the segment bounds
    RaiseException(PROTECTION_FAULT);
end
else begin
    // Handle address growth based on segment type
    if (Segment == 2'b10) begin
        // Stack segment (reverse growth)
        PhysAddr = Base[Segment] - Offset;
    end
    else begin
        // Code or heap segment (forward growth)
        PhysAddr = Base[Segment] + Offset;
    end

    // Access the physical memory at the calculated address
    Register = AccessMemory(PhysAddr);
end
```

Assume 14-bit virtual addresses, high 2 bits indicate segment



Where doe segment table live?

All registers, MMU

Seg	Base	Bounds
0	0x4000	0xfff
1	0x5800	0xfff
2	0x6800	0x7ff

```
0x0010:  movl   (0x1100), %edi
0x0013:  addl   $0x3, %edi
0x0019:  movl   %edi, (0x1100)
```

%rip: 0x0010

Physical Memory Accesses?

1) Fetch instruction at logical addr 0x0010

- Physical addr: 0x4010

Exec, load from logical addr 0x1100

- Physical addr: 0x5900

2) Fetch instruction at logical addr 0x0013

- Physical addr: 0x4013

Exec, no load

3) Fetch instruction at logical addr 0x0019

- Physical addr: 0x4019

Exec, store to logical addr 0x1100

- Physical addr: 0x5900

Total of 5 memory references (3 instruction fetches, 2 movl)

硬件还有其他方法来决定特定地址在哪个段.
在隐式方式中，硬件通过虚拟地址产生的方式来确定段.
例如，如果虚拟地址由程序计数器产生，那么该地址在代码段.
如果基于栈或基址指针，那么它一定在栈段.
其他地址则在堆段.

Advantages of Segmentation

- **Enables sparse allocation of address space**
  - Stack and heap can grow independently
  - Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
  - Stack: OS recognizes reference outside legal segment, extends stack implicitly
- **Different protection for different segments**
  - Read-only status for code
- **Enables sharing of selected segments**
- **Supports dynamic relocation of each segment**

Disadvantages of Segmentation

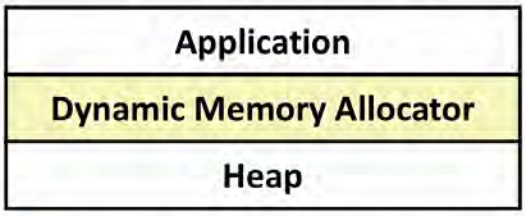
- **Each segment must be allocated contiguously**
  - May not have sufficient physical memory for large segments
- **Fix in next lecture with paging…**

2.5 动态内存分配

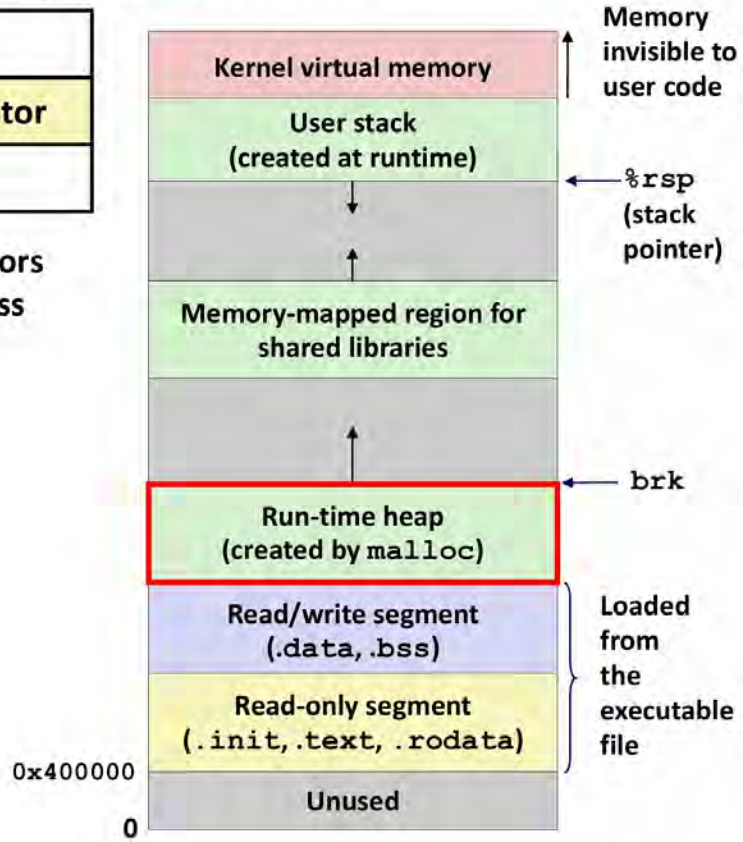
2.5.1 概念

Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
  - *Explicit allocator*: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - *Implicit allocator:* application allocates, but does not free space
    - E.g., `new` and garbage collection in Java



- Dynamic memory allocators manage an area of process VM known as the *heap*.



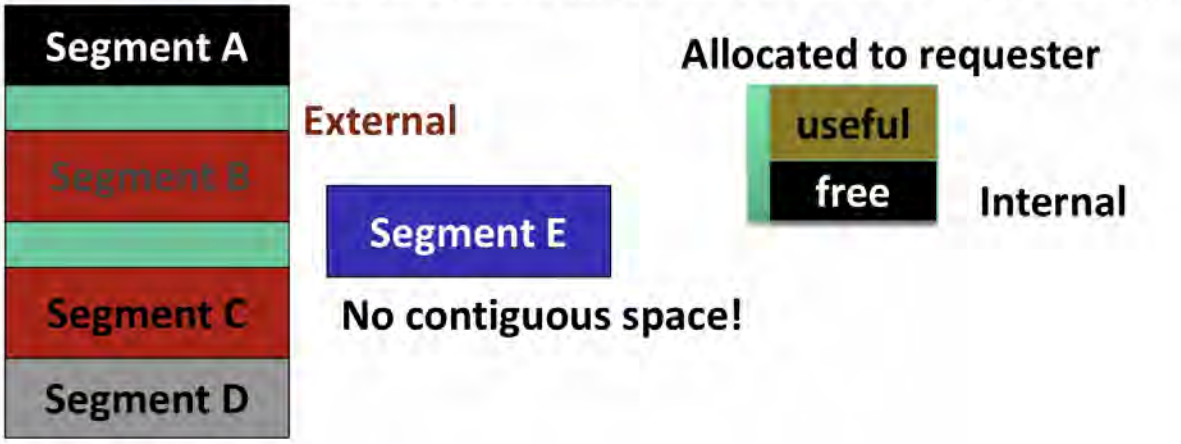
本节我们介绍 `malloc` 和 `free` 的底层机制.

- `void * malloc(size_t size)` 返回一个指针 (没有具体的类型， 在 C 语言的术语中是 void 类型) 指向 `size` 大小 (或稍大一点) 的一块空间.
- `void free (void *ptr)` 接受一个指针，释放对应的内存块.
  用户不需告知库这块空间的大小 (这由库自己来追踪这块空间的大小).

该库管理的空间由于历史原因被称为堆，在堆上管理空闲空间的数据结构通常称为空闲列表 (free list)
当然它不一定真的是列表，只要能够管理内存区域中所有空闲块的引用即可.

Problem: Fragmentation

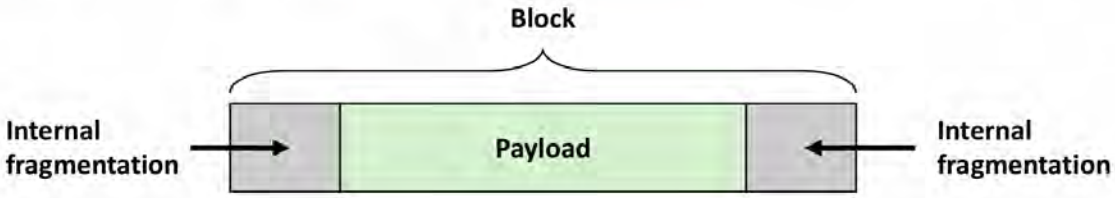
- **Definition: Free memory that can't be usefully allocated**
- **Why?**
  - Free memory (hole) is too small and scattered
  - Rules for allocating memory prohibit using this free space
- **Types of fragmentation**
  - External: Visible to allocator (e.g., OS)
  - Internal: Visible to requester (e.g., if must allocate at some granularity)



简单起见，我们主要关心的是**外部碎片** (external fragmentation) 问题.
另一种形式的空间浪费是**内部碎片** (internal fragmentation) 问题，即已分配单元的内部有未使用的空间.

Internal Fragmentation

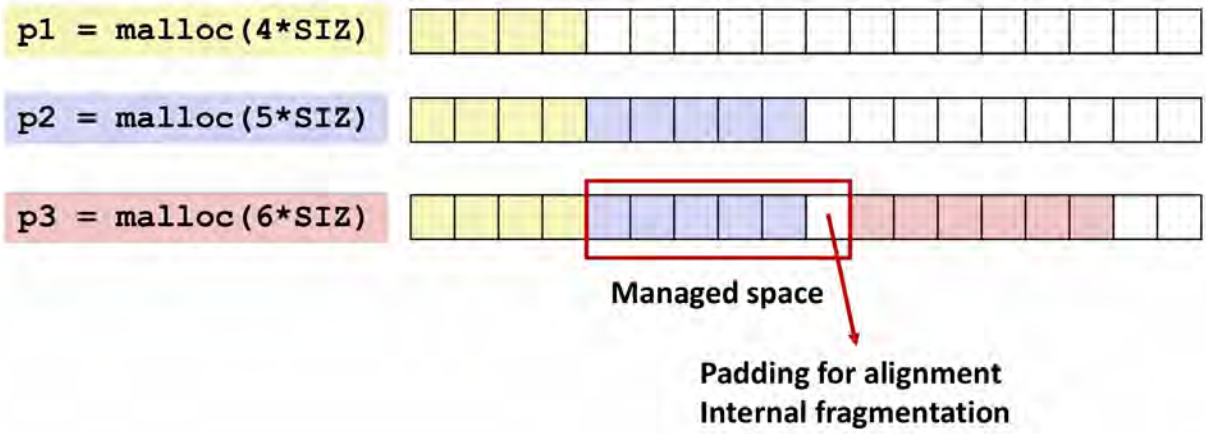
- For a given block, *internal fragmentation* occurs if payload is smaller than block size

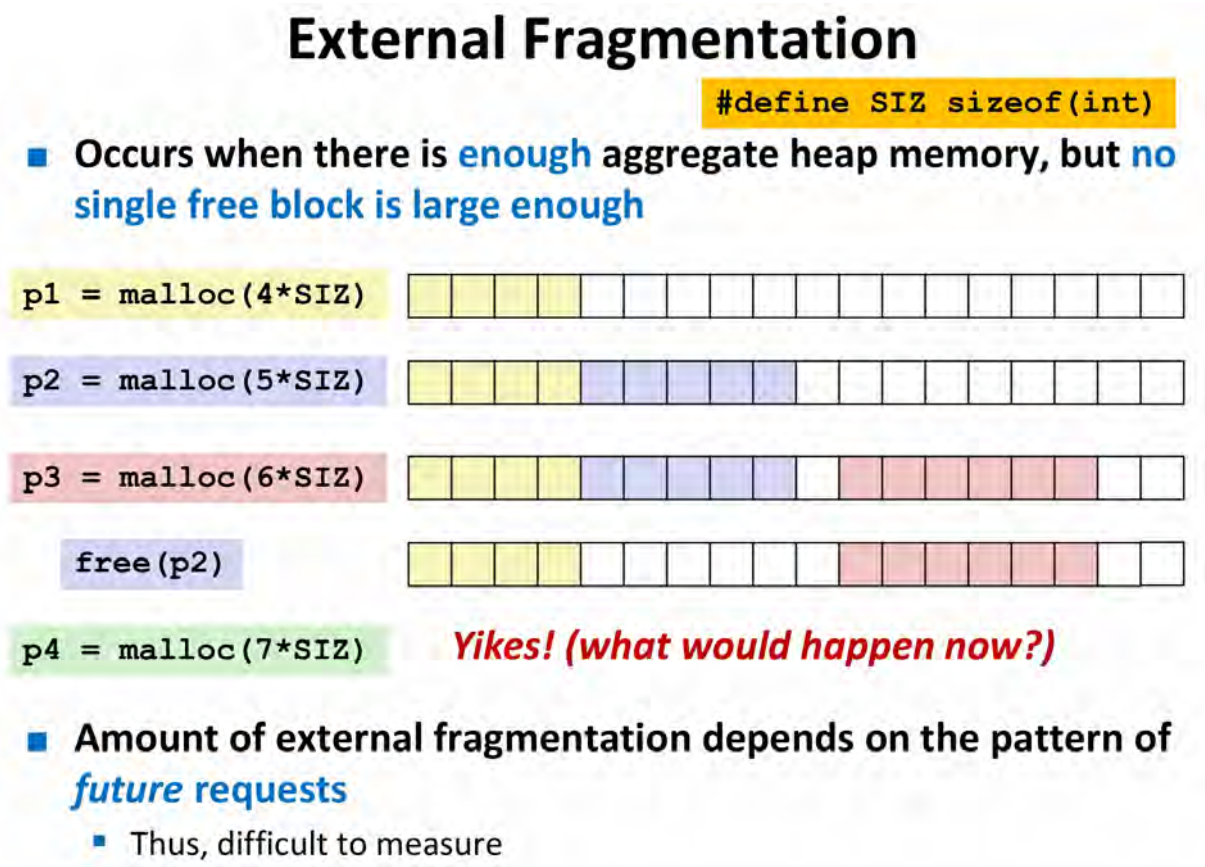


- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
  - Thus, easy to measure

```
#define SIZ sizeof(int)
```

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

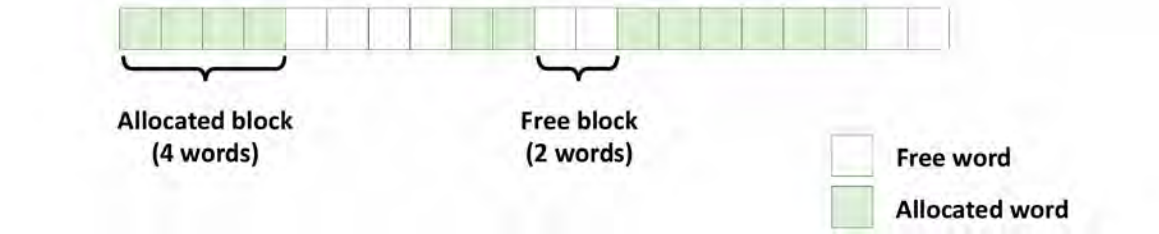




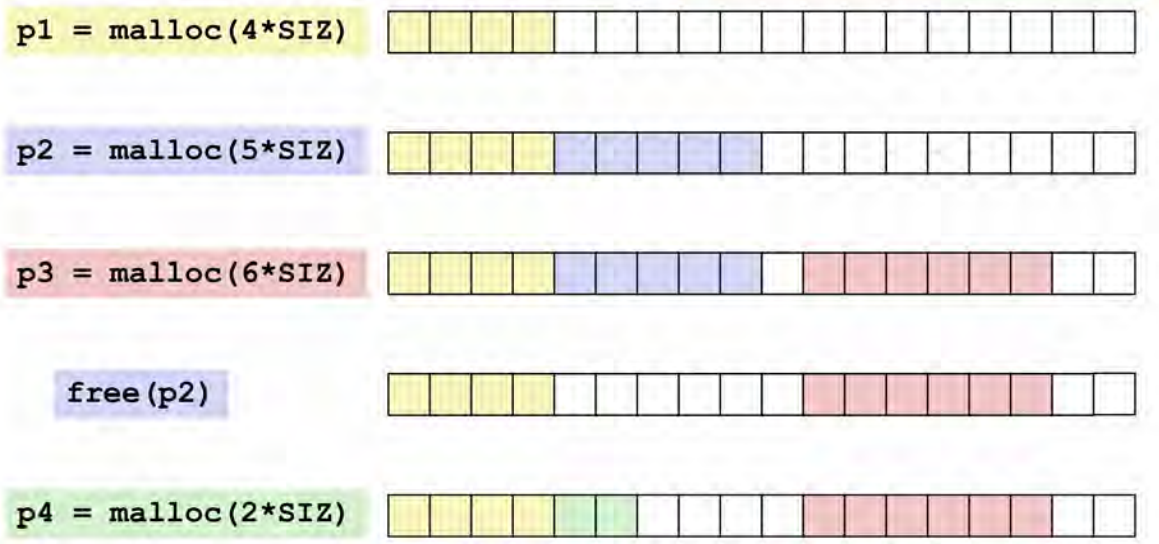
简单起见，我们做以下假设:

- ① 内存一旦通过 `malloc` 函数分配给用户，就不可以被重定位到其他位置，直到程序调用 `free` 函数将它归还. 因此不可能进行压缩 (compaction) 空闲空间的操作来减少外部碎片.
  (不过在实现分段的机器上，或在带垃圾收集的语言中，可以通过压缩技术减少外部碎片)
- ② 分配程序所管理的是连续的字节区域.
- ③ 分配程序所管理的空间在整个生命周期内大小固定.
  但实际上分配程序可以要求这块区域增长，例如通过 `sbrk` 等系统调用
  操作系统会找到空闲的物理内存页，将它们映射到请求进程的地址空间中去，并返回新的堆的末尾地址.
- ④ 内存是按字寻址的 (word addressed)
- ⑤ 字是整型大小的 (int-sized)
- ⑥ 内存分配是双字对齐的 (double-word aligned)

Allocation Example




```
#define SIZ sizeof(int)
```

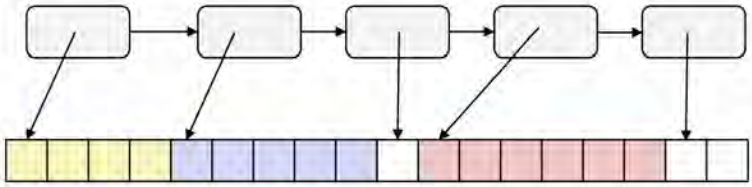


Constraints

- **Applications**
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**'d block
- **Explicit Allocators**
  - Can't control number or size of allocated blocks
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 16-byte (x86-64) alignment on Linux
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are **`malloc`**'d
    - *i.e.*, compaction is not allowed. *Why not?*

How to Implement

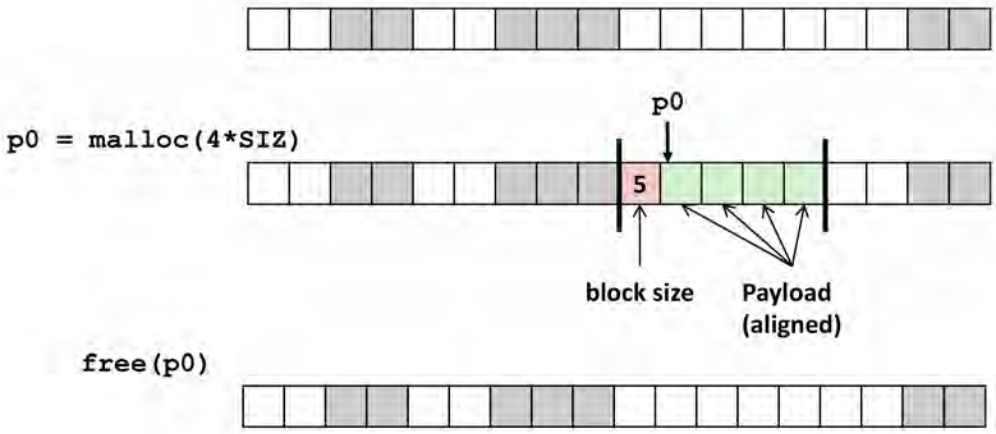
- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we reclaim freed block?
- How about a new linked list to maintain the allocated and free blocks?



- Where to allocate the nodes?
  - New nodes in allocation or delete node when free
  - From Heap? But we are managing heap
  - Allocate() -> new node in linked list -> Allocate space for node -> new node in linked list -> ...
  - Nice to manage the heap without extra data strutures

Knowing How Much to Free

- Standard method
  - Keep the length of a block in the word *preceding* the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block



Keeping Track of Free Blocks

Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks

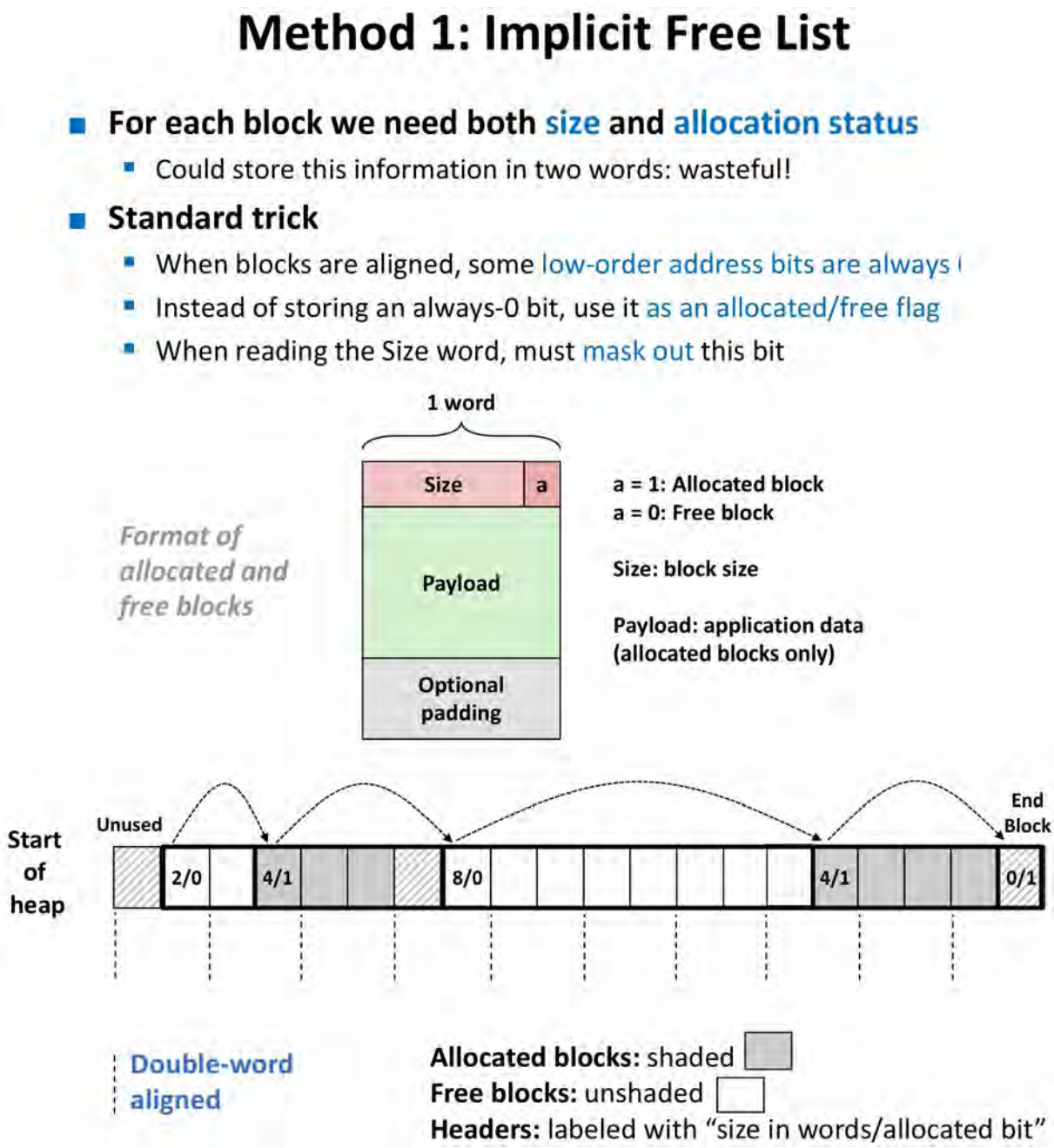
Need to tag each block as allocated/free

- Method 2: *Explicit list* among the free blocks using pointers

Need space for pointers

- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

2.5.2 隐式空闲列表



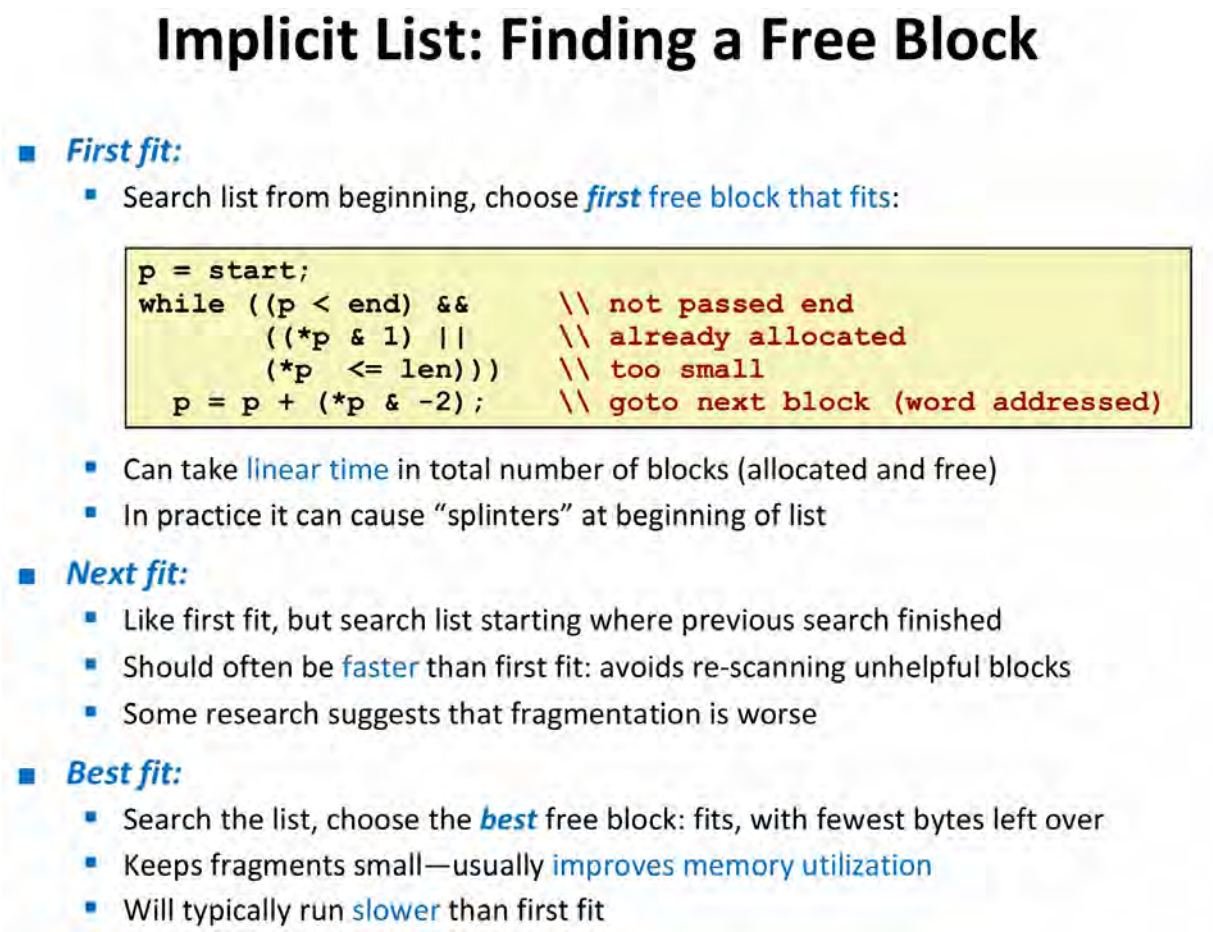
(1) 寻找空闲块

理想的分配程序需要同时保证快速和碎片最小化.
遗憾的是，由于内存分配及释放的请求序列是任意的 (毕竟它们由用户程序决定),
任何特定的策略在某组不匹配的输入下都会变得非常差.
因此没有一种策略是 "最好" 的，因此我们只介绍一些基本策略，并讨论它们的优缺点.

- **① 最佳匹配 (best fit)**
  遍历整个空闲列表，找到满足请求大小的空闲块，然后返回这组候选者中最小的一块.
  其想法很简单: 选择最接近用户请求大小的块，从而尽量避免空间浪费.
  但上述策略需要付出较高的性能代价.
- **② 最差匹配 (worst fit)**
  与最优匹配相反，它遍历整个空闲列表，找到满足请求大小的空闲块，然后返回这组候选者中最大的一块.
  最差匹配尝试在空闲列表中保留较大的块， 而不是像最优匹配那样可能剩下很多难以利用的小块.
  但最差匹配同样需要遍历整个 空闲列表.
  更糟糕的是，大量研究表明它的表现非常差，导致过量的碎片，同时还有很高的性能开销.
- **③ 首次匹配 (first fit)**
  从空闲列表的开始查找，只要找到第一个足够大的空闲块，就将其分配给用户.
  这个策略具有速度优势 (不需要遍历所有空闲块)，但有时会让空闲列表开头的部分有很多小块.
  此时分配程序如何管理空闲列表的顺序就变得很重要.

一种方式是**基于地址排序** (address-based ordering)
通过保持空闲块按内存地址排序，这会让合并操作变得容易，从而减少外部碎片.

- **④ 下次匹配 (next fit)**
  不同于首次匹配每次都从空闲列表的开始查找，下次匹配策略多维护一个指针，指向上一次查找结束的位置.
  其想法是将对空闲空间的查找操作扩散到整个列表中去，避免对列表开头频繁的分割.
  这种策略的性能与首次匹配很接近，同样避免了遍历查找.



(2) 追踪已分配空间的大小

分配程序需要追踪已分配空间的大小 (这样用户无需传入大小参数)，从而在释放时能够将它完整地放回空闲列表.
要完成这个任务，大多数分配程序都会在**头块** (header) 中保存一点额外的信息，头块通常位于返回的内存块之前.
头块至少包含所分配空间的大小，还可能包含一个幻数来提供正常性检查 (甚至一些额外的指针来加速空间释放):

```
typedef struct header_t {
 int size;
 int magic;
} header_t;
```

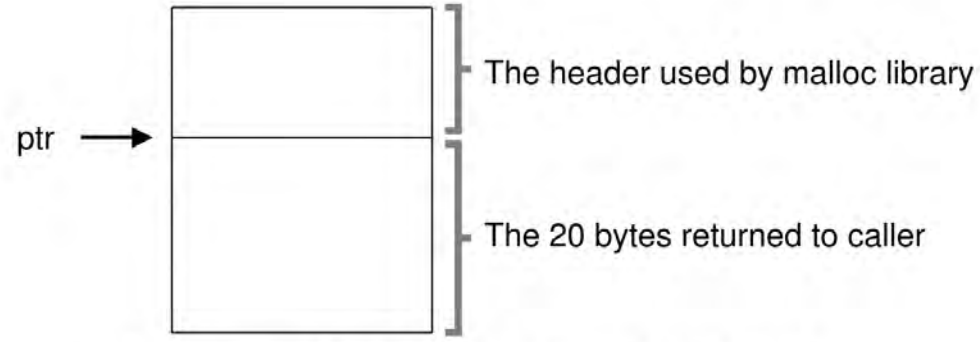


Figure 17.1: **An Allocated Region Plus Header**

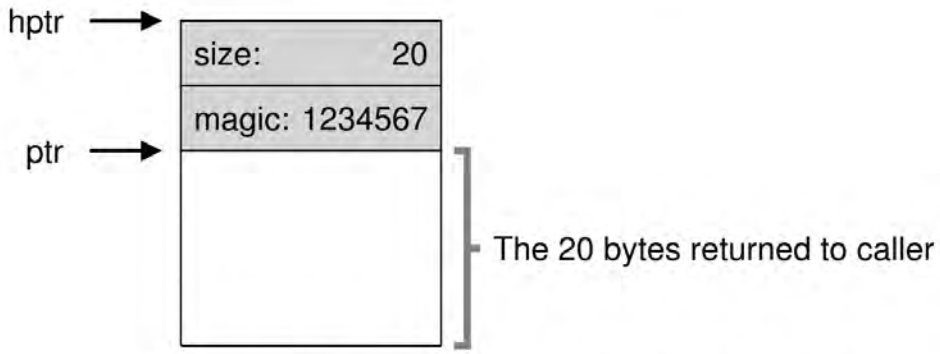


Figure 17.2: **Specific Contents Of The Header**

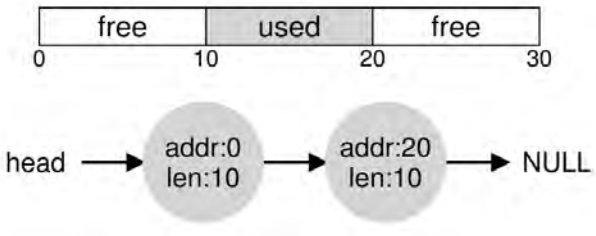
用户调用 `free(ptr)` 时，库会通过简单的指针运算找到头块的位置:

```
header_t *hptr = (void *)ptr - sizeof(header_t);
```

获得头块的指针后，库可以很容易地确定幻数是否符合预期的值以作为正常性检查 (本例中为 `assert (hptr->magic == 1234567)`)
并简单计算要释放的空间大小 (即头块的大小加上头块记录的 `size` 参数，即分配给用户的空间的大小)
因此，如果用户请求 N 字节的内存，那么分配程序实际寻找的是大于等于 N 加上头块大小的空闲块.

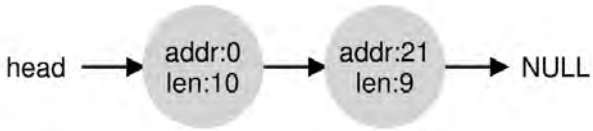
(3) 简单实现

(分割 & 合并)
空闲列表记录了堆中哪些空间还没有分配:



在上述例子中，任何大于 10 字节的分配请求都会失败，因为没有足够的连续可用空间.
而恰好 10 字节的需求可以由两个空闲块中的任何一个满足.

- 现假设我们只申请 1 个字节的内存.
  此时分配程序会找到一块可以满足请求的空闲空间，将其**分割** (split)，第一块返回给用户，第二块留在空闲列表中.



- 现假设我们调用 `free(10)` 释放了从地址 10 开始的 10 字节已分配空间.
  如果只是简单地将这块空闲空间加入空闲列表，则会得到如下结果:

  尽管整个堆现在完全空闲，但它被分割成了 3 个 10 字节的区域.
  为了避免这个问题，当一块内存被释放时，分配程序会将其与邻近空闲块**合并** (coalesce) 为一个较大的空闲空间.

我们该如何在空闲内存的内部构建空闲列表呢?
(虽然听起来有点奇怪，但别担心，这是可以这到的)

假设我们需要管理一个 $4\text{KB} = 4096\text{B}$ 的内存块.
刚开始，空闲列表仅有一个节点，记录了大小为 4096B 的空间 (减去头块的大小):

```
typedef struct node_t {
    int size;
    struct node_t *next;
} node_t;
```

我们使用系统调用 `mmap()` 初始化堆，并将空闲列表的第一个节点放在该空间上:

```
// Allocate a 4KB block of memory using mmap
node_t *head = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);

// Initialize the first node in the free list
head->size = 4096 - sizeof(node_t);  // Calculate usable size
head->next = NULL;                   // Initialize the next pointer to NULL
```

其中 `prot = PROT_READ | PROT_WRITE` 表明这块内存可以读写
`flags = MAP_ANON | MAP_PRIVATE` 表示私有匿名映射，代表这块内存不关联文件且仅供当前进程使用.
执行上述代码后，空闲列表仅有一个节点，记录的大小为 `4088`

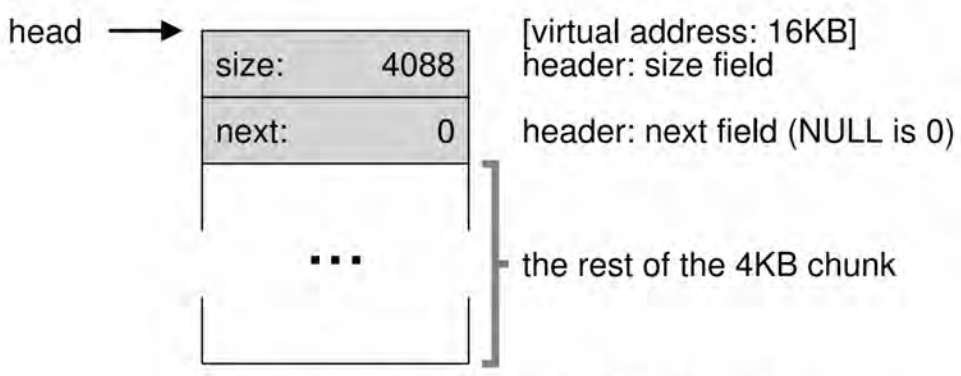


Figure 17.3: **A Heap With One Free Chunk**

现假设有一个 100 字节的内存请求，我们需要实际分配 $100 + 8 = 108$ 字节的内存空间
(其中 8 个字节用于记录头块信息，前 4 字节记录大小，后 4 字节存放幻数，以供未来的 `free` 函数使用)
最后空闲列表上的空闲节点缩小为 $4088 - 108 = 3980$ 字节.

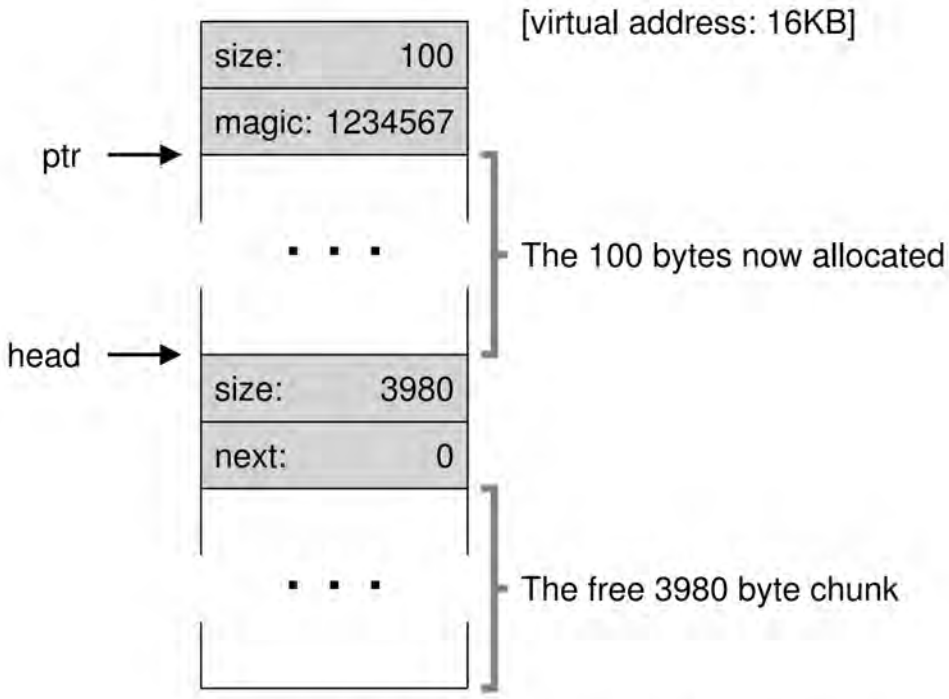


Figure 17.4: **A Heap: After One Allocation**

现假设我们已经处理了三个 100 字节的内存请求
此时空闲列表上的空闲节点缩小为 $4088 - 108 \times 3 = 3764$ 字节.
我们即将释放分配的第二个内存块:

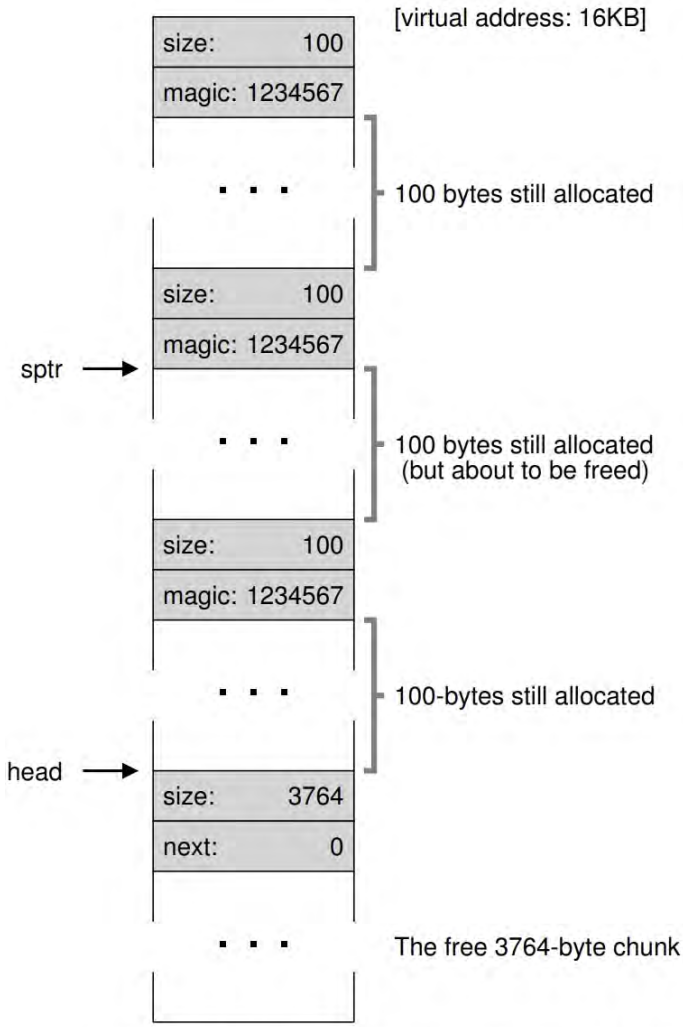


Figure 17.5: **Free Space With Three Chunks Allocated**

第二个内存块的起始位置是 $16\text{KB} + 108\text{B} + 8\text{B} = 16384\text{B} + 116\text{B} = 16500\text{B}$
我们调用 `free(16500)` 来释放分配的第二个内存块.
分配程序很快计算出要释放的空间的大小为 $100\text{B} + 8\text{B} = 108\text{B}$，将空闲块放回空闲列表 (假设插到头位置)
此时的空闲列表便包括一个小空闲块 (100 字节，由列表的头指向) 和一个大空闲块 (3764 字节)

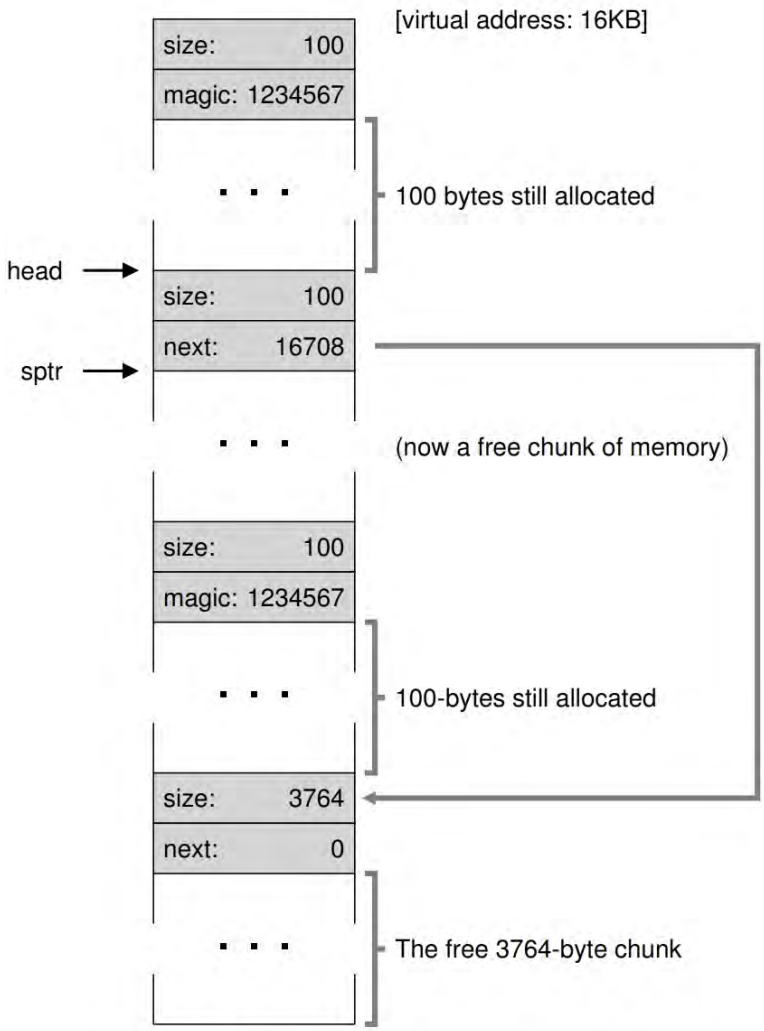


Figure 17.6: **Free Space With Two Chunks Allocated**

现在我们分别释放已分配的第一个内存块和第三个内存块.
前者调用 `free(16384)`，后者调用 `free(16608)` (因为 $16384 + 2 \times 108 + 8 = 16608$) (假设插到头位置)
若不进行合并，则空闲列表有四个节点:

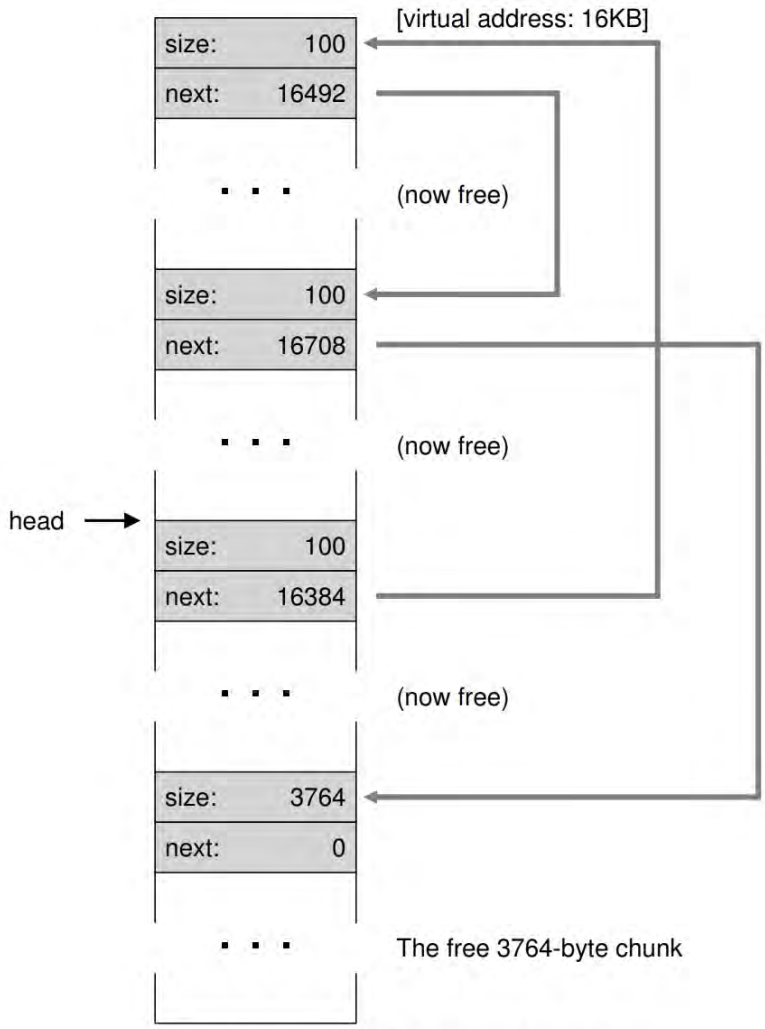


Figure 17.7: **A Non-Coalesced Free List**

遍历列表，合并相邻块之后，堆由成了一个整体:

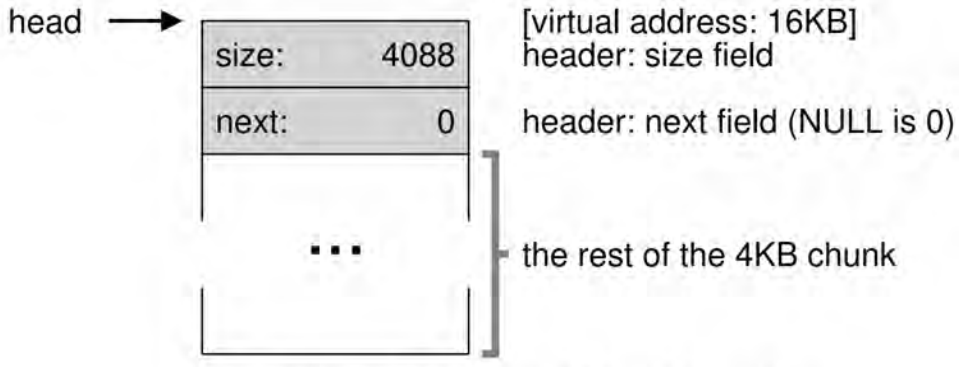
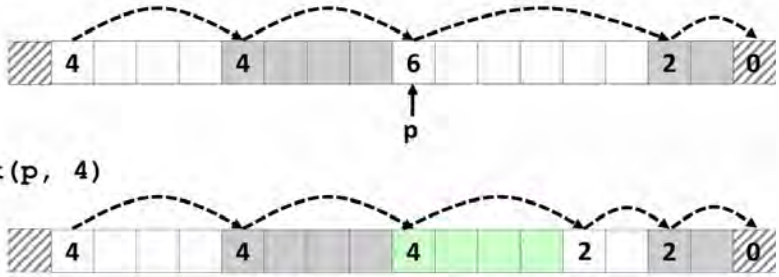


Figure 17.3: **A Heap With One Free Chunk**

(4) 进阶实现

Implicit List: Allocating in Free Block

- **Allocating in a free block: *splitting***
  - Since allocated space might be smaller than free space, we might want to split the block



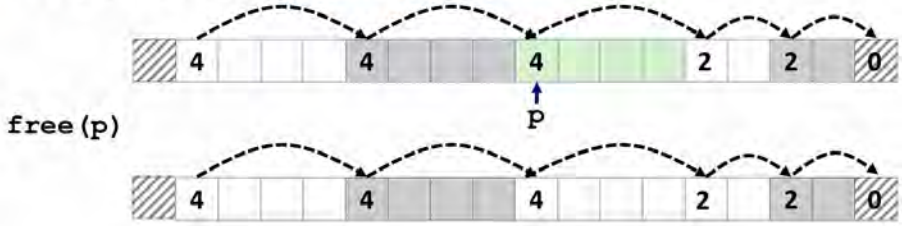

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

Implicit List: Freeing a Block

- **Simplest implementation:**
  - Need only clear the "allocated" flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to "false fragmentation"

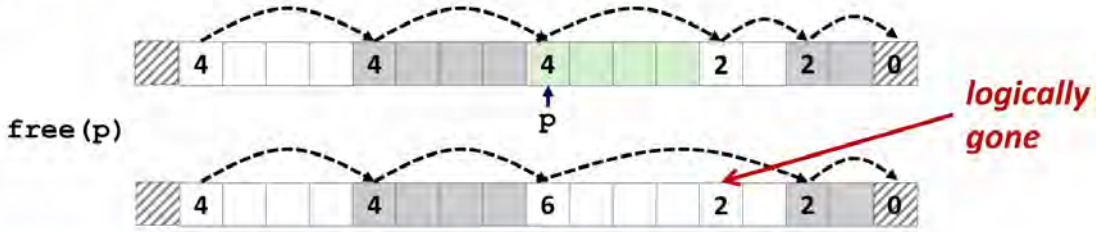


`malloc(5*SIZ)` ***Yikes!***

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

Implicit List: Coalescing

- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block

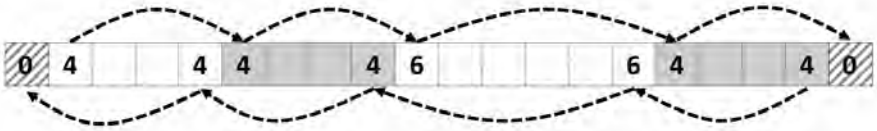



```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated flag
    next = p + *p;          // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;      // add to this block if
}                           //    not allocated
```

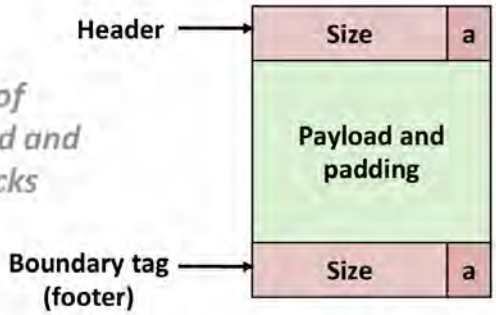
  - But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

- ***Boundary tags*** [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!



Format of allocated and free blocks



a = 1: Allocated block
a = 0: Free block

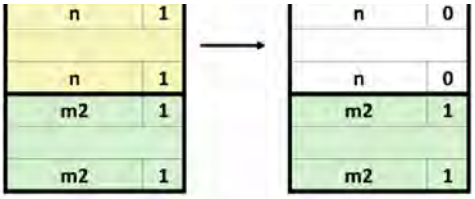
Size: Total block size

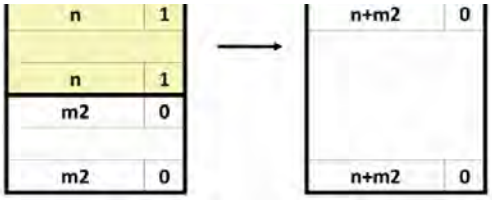
Payload: Application data (allocated blocks only)

Constant Time Coalescing

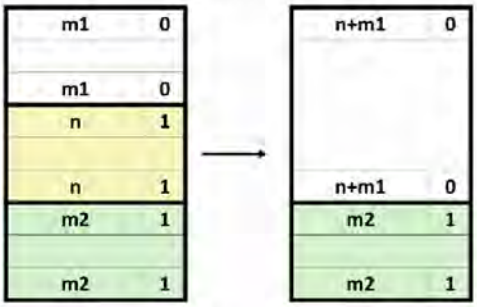
Case 1

Case 2

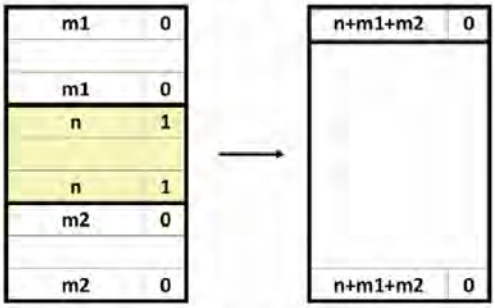

n 1
n 1
m2 1
m2 1
n 0
n 0
m2 1
m2 1


n 1
n 1
m2 0
m2 0
n+m2 0
n+m2 0

Case 3


m1 0
m1 0
n 1
n 1
m2 1
m2 1
n+m1 0
n+m1 0
m2 1
m2 1

Case 4

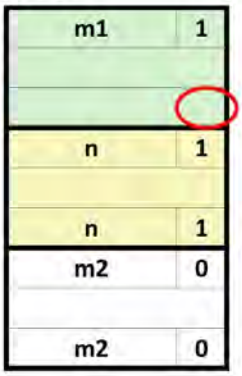

m1 0
m1 0
n 1
n 1
m2 0
m2 0
n+m1+m2 0
n+m1+m2 0

Disadvantages of Boundary Tags

- Internal fragmentation
- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?

No Boundary Tag for Allocated Blocks

- Boundary tag needed only for free blocks
- How to know if the previous block is free or allocated?
  - Last bit means nothing if there is no bottom boundary tag

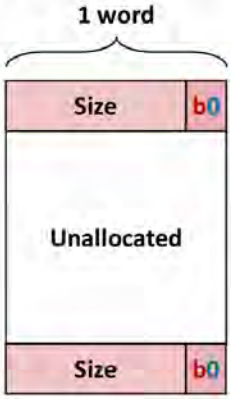


- When sizes are multiples of 4 or more, have 2+ spare bits
  - An extra bit to record if the allocation status of the previous block

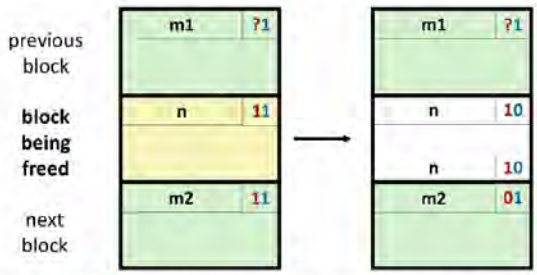
a = 1: Allocated block
a = 0: Free block
b = 1: Previous block is allocated
b = 0: Previous block is free

Size: block size

Payload: application data

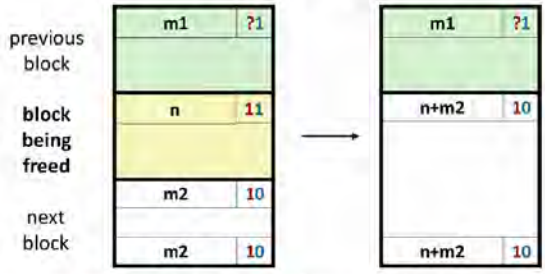


Case 1



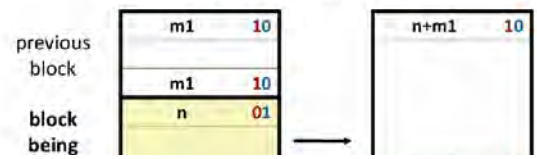
Header: Use 2 bits (address bits always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

Case 2

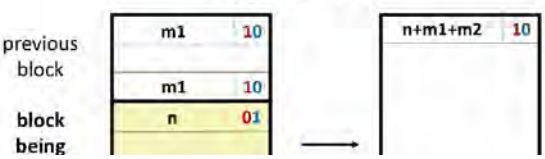


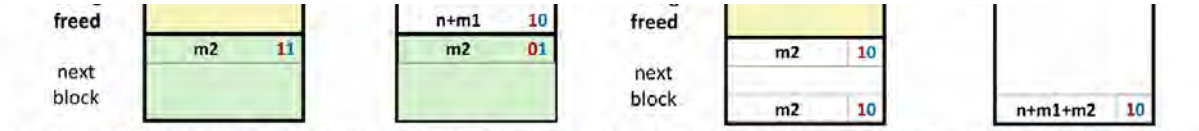
Header: Use 2 bits (address bits always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

Case 3



Case 4





Header: Use 2 bits (address bits always zero due to alignment): (previous block allocated)<<1 | (current block allocated)

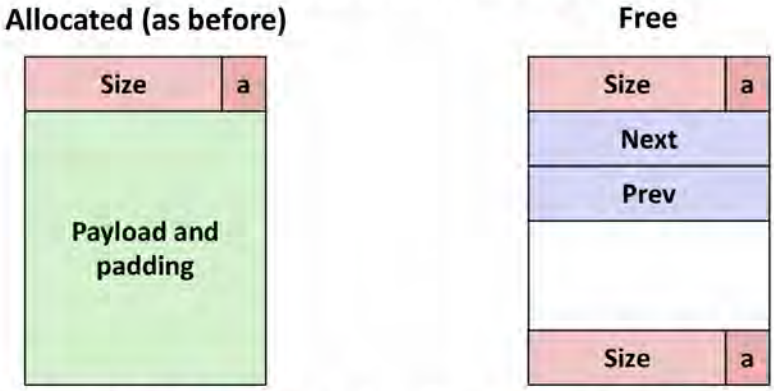
Header: Use 2 bits (address bits always zero due to alignment): (previous block allocated)<<1 | (current block allocated)

Summary of Key Allocator Policies

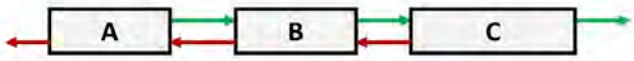
- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - Segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list
- **Splitting policy:**
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- **Coalescing policy:**
  - ***Immediate coalescing:*** coalesce each time `free` is called
  - ***Deferred coalescing:*** try to improve performance of `free` by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for `malloc`
    - Coalesce when the amount of external fragmentation reaches some threshold
- **Implementation: very simple**
- **Allocate cost:**
  - linear time worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory usage:**
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- **Not used in practice for `malloc/free` because of linear-time allocation**
  - used in many special purpose applications
- **However, the concepts of splitting and boundary tag coalescing are general to *all* allocators**

2.5.3 显式空闲列表

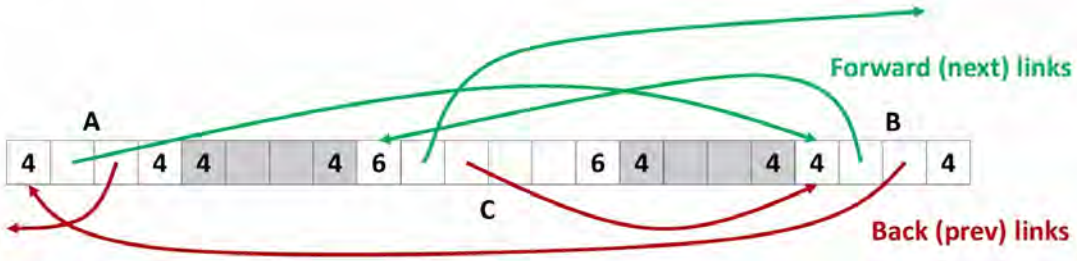
Explicit Free Lists



- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area
- **Logically:**

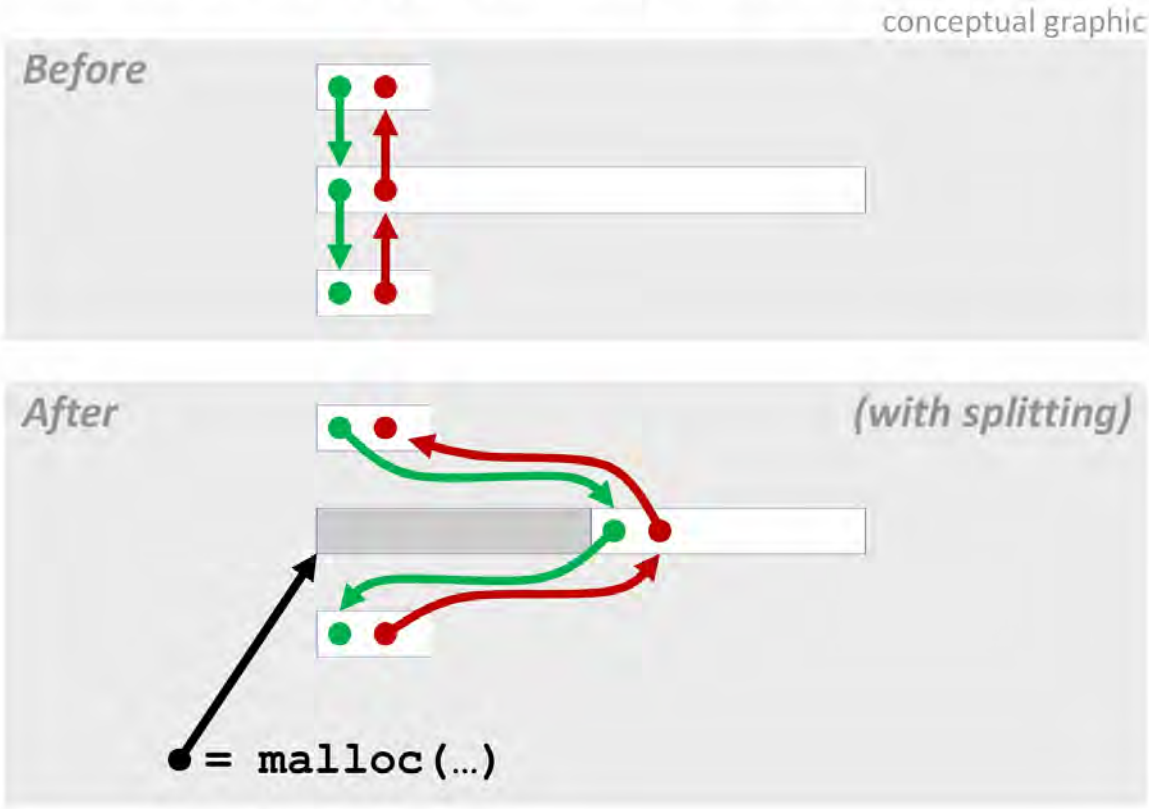


- **Physically: blocks can be in any order**

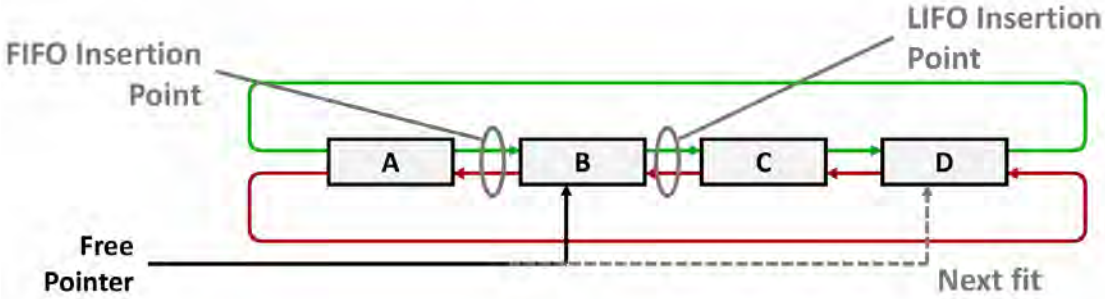


- **Comparison to implicit list:**
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free because need to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?
- **Most common use of linked list approach is in conjunction with segregated free lists**
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Allocating From Explicit Free Lists



Some Advice: An Implementation Trick

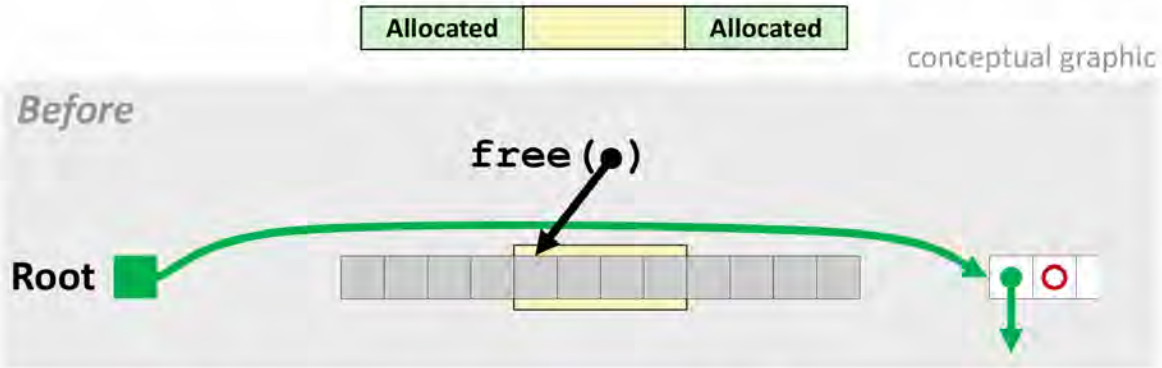


- Use circular, doubly-linked list
- Support multiple approaches with single data structure
- First-fit vs. next-fit
  - Either keep free pointer fixed or move as search list
- LIFO vs. FIFO
  - Insert as next block (LIFO), or previous block (FIFO)

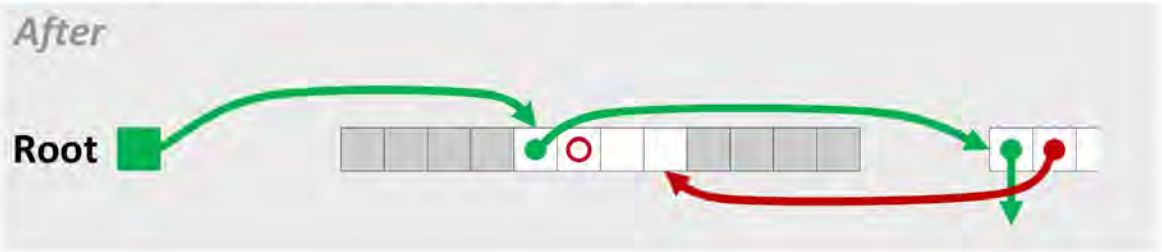
Freeing With Explicit Free Lists

- ***Insertion policy*: Where in the free list do you put a newly freed block?**
- **Unordered**
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
  - FIFO (first-in-first-out) policy
    - Insert freed block at the end of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered
- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO/FIFO

Freeing With a LIFO Policy (Case 1)

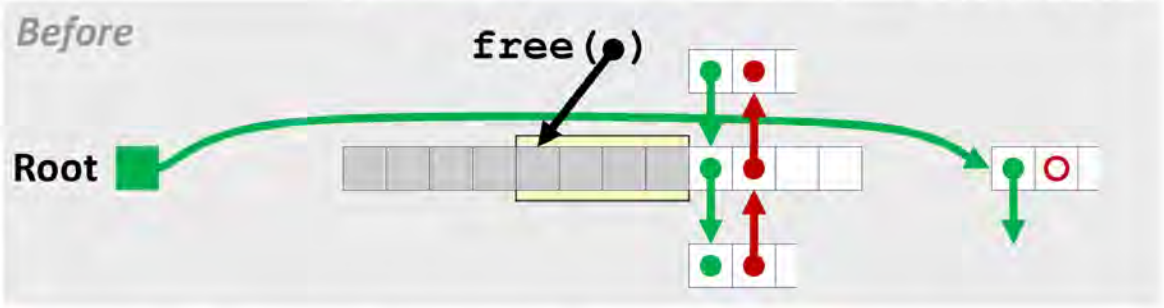


- **Insert the freed block at the root of the list**

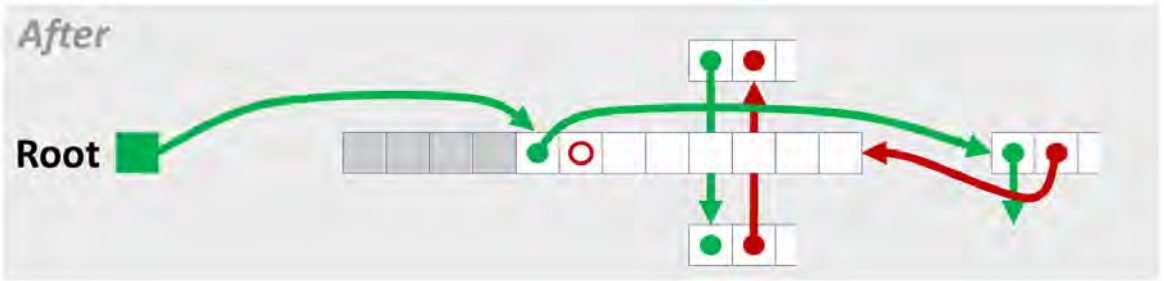


Freeing With a LIFO Policy (Case 2)

conceptual graphic

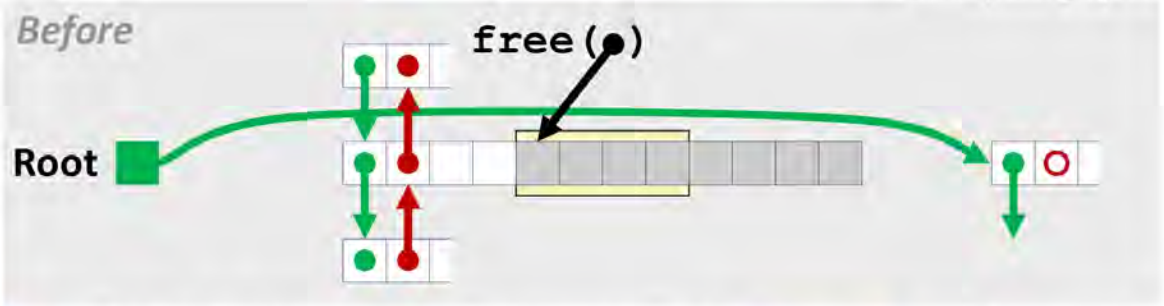


- Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list

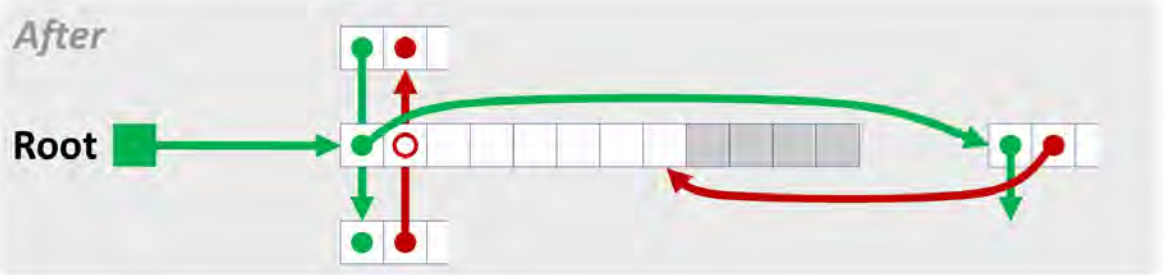


Freeing With a LIFO Policy (Case 3)

conceptual graphic

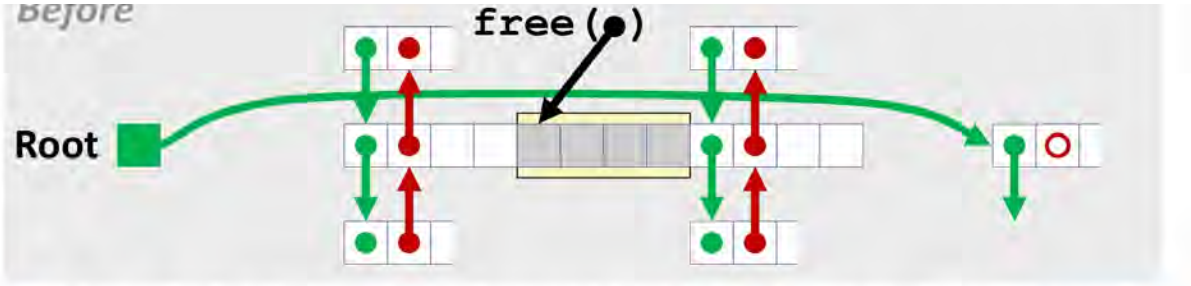


- Splice out adjacent predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

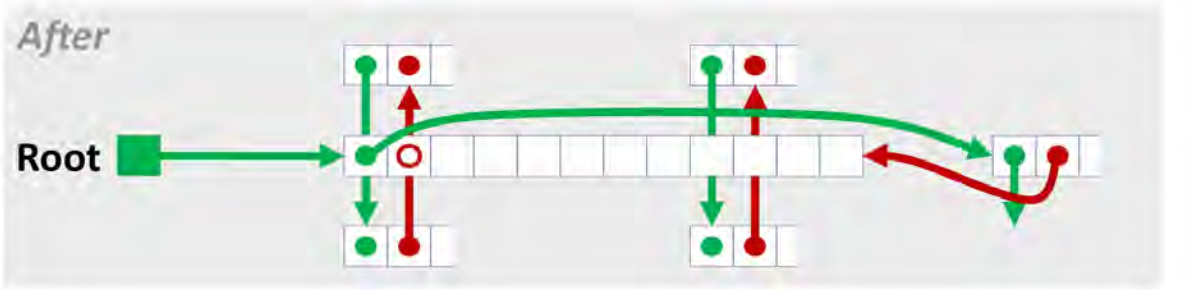


Freeing With a LIFO Policy (Case 4)

conceptual graphic



- Splice out adjacent predecessor and successor blocks, coalesce all 3 blocks, and insert the new block at the root of the list



2.6 分页

操作系统通过将空间分割成固定长度的分片来解决大多数空间管理问题，我们称这种思想为**分页** (paging)
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、堆和栈) (这样会使空间碎片化)
而是分割成固定大小的单元，每个单元称为一**页** (page).
相应地，我们把物理内存看成是定长槽块的阵列，我们称其槽块为**页帧** (page frame)
每个页帧都可以存放一个虚拟内存页.

2.6.1 一个例子

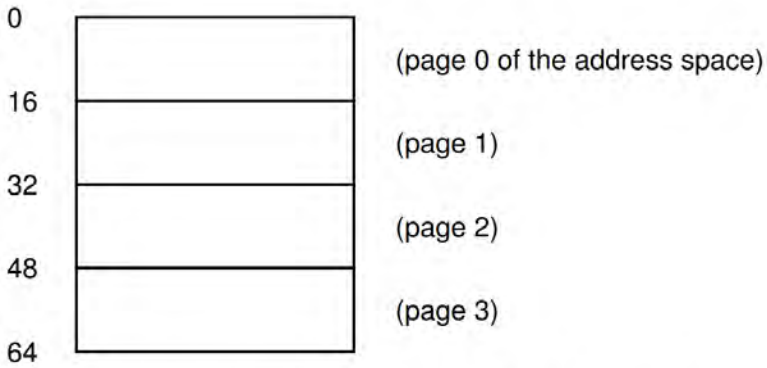


Figure 18.1: **A Simple 64-byte Address Space**

考虑一个仅有 64B 的小地址空间，分为 4 个 16B 的页 (记为虚拟页 $0, 1, 2, 3$)
(当然，真实的地址空间会大得多，例如在 32 位字节寻址的情况下地址空间的大小为 4GB)
现在我们将其放入 128B 的物理内存中:

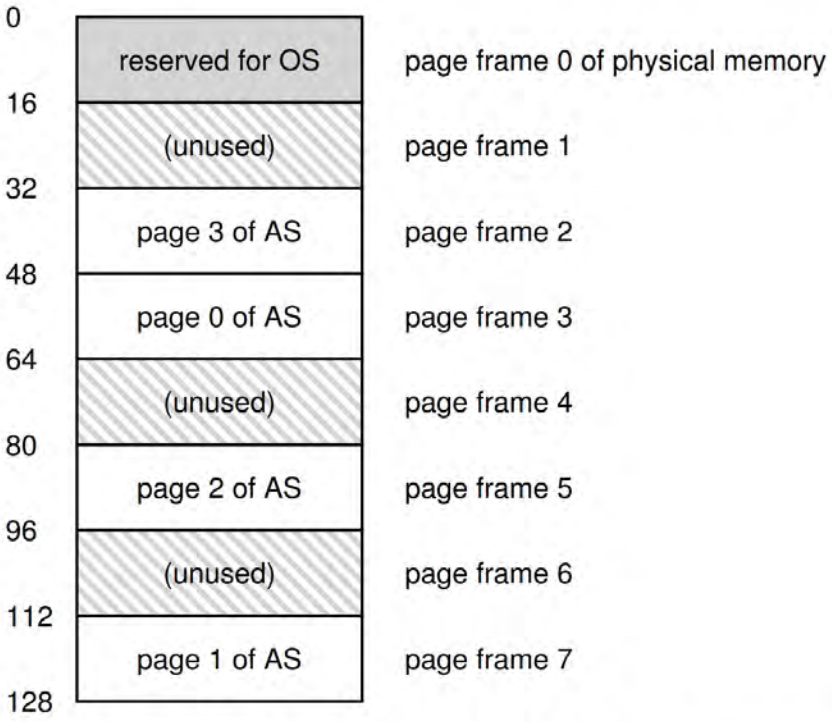


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

与我们以前的方法相比，分页方法有许多优点:

- 最大的改进就是**灵活性** (flexibility):
  通过完善的分页方法，操作系统能够高效地提供地址空间的抽象，不管进程如何使用地址空间.
  例如我们不用假定堆和栈的增长方向以及它们如何使用.
- 另一个优点是分页提供的空闲空间管理的**简单性** (simplicity):
  在上述例子中，要将 $64\mathrm{B}$ 的小地址空间放入物理内存，只需从空闲列表中拿出 4 个空闲页即可.

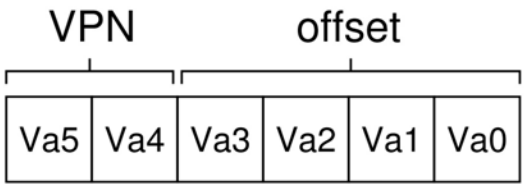
为了记录地址空间的每个虚拟页放在物理内存中的位置，
操作系统通常为每个进程保存一个数据结构，称为**页表** (page table)
页表的主要作用是为地址空间的每个虚拟页面保存**地址转换** (address translation)
从而让我们知道每个页在物理内存中的位置.

在上述例子中，页表具有以下四个条目:
(我们将 "虚拟页" 和 "页帧" 分别简记为 VP 和 PF)

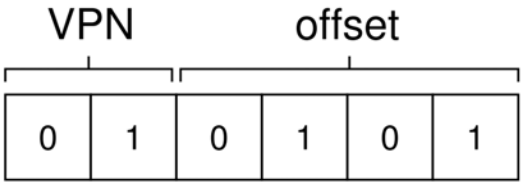
- $\mathrm{VP0} \to \mathrm{PF3}$
- $\mathrm{VP1} \to \mathrm{PF7}$
- $\mathrm{VP2} \to \mathrm{PF5}$
- $\mathrm{VP3} \to \mathrm{PF2}$

操作系统会为不同的进程管理不同的页表，将其虚拟页映射到不同的物理页面 (共享的情况除外)

假设拥有这个 $64\mathrm{B}$ 的小地址空间的进程正在访问内存.
其虚拟地址的位数为 $\log_2(64) = 6$
高 $\log_2(4) = 2$ 位用于确定**虚拟页号** (Virtual Page Number, VPN)
低 $\log_2(16) = 4$ 位用于确定**页内偏移量** (page offset)



当进程生成虚拟地址时，操作系统和硬件必须协作，将它转换为有意义的物理地址.
假设进程访存使用的虚拟地址为 $0\mathrm{x}15$，我们将其表示为二进制形式:



因此 $0\text{x}15$ 指向的是第 $0\text{b}01 = 1$ 个虚拟页的第 $0\text{b}0101 = 5$ 个字节.
根据上面的页表可知 $\text{VP}1 \rightarrow \text{PF}7$
因此**物理页号** (Physical Page Number, PPN) 为 $7 = 0\text{b}111$，页内偏移量仍为 $5 = 0\text{b}0101$
于是我们可以用 PPN 代替 VPN (而偏移量不变)
来将虚拟地址 $0\text{x}15 = 0\text{b}010101$ 转换为物理地址 $0\text{b}1110101 = 0\text{x}75$，然后发送给物理内存.

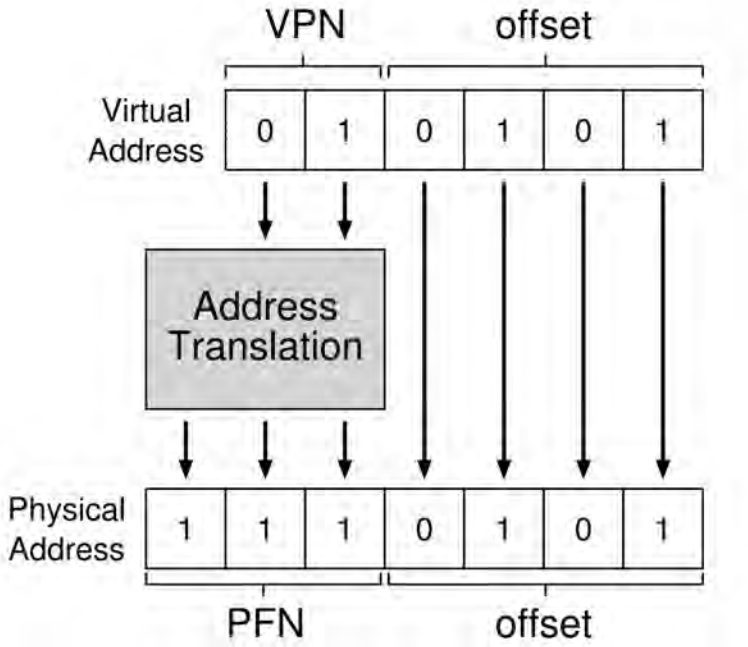


Figure 18.3: **The Address Translation Process**

2.6.2 页表

页表就是一种数据结构，用于将虚拟地址映射到物理地址 (实际上是将虚拟页号映射到物理页号)
我们暂且假设其采用最简单的形式，即一个数组，称为**线性页表** (linear page table)
操作系统通过虚拟页号 VPN 检索该数组，并在该索引处查找页表项 PTE，以便找到期望的物理页号 PPN

每个页表项 PTE 都有许多不同的位:

- **有效位** (valid bit): 用于指示特定地址转换是否有效
  通过简单地将地址空间中所有未使用的页面标记为无效，我们便不再需要为这些页面分配物理页，从而节省大量内存.
- **保护位** (protection bit): 表明页是否可以进行读写等操作
- **存在位** (present bit): 表示该页是在物理内存还是在磁盘上 (即它是否已被换出)
- **修改位** (dirty bit): 表明页面被放入物理内存后是否被修改过
- **访问位** (accessed bit): 与页面替换相关
  用于追踪页是否被访问，也用于确定哪些页很受欢迎 (因而应当保留在物理内存中)

下图展示了一个来自 x86 架构的示例页表项:

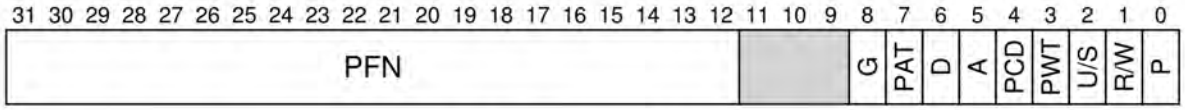


Figure 18.5: **An x86 Page Table Entry (PTE)**

它包含物理页号 PPN、访问位 A、修改位 D、保护位 R/W、存在位 P、用户模式位 U/S
(物理页号又称页帧号, Page Frame Number, PFN)
以及几个用于确定硬件缓存如何为这些页面工作的位 (PWT,PCD,PAT,G)

不幸的是，分页机制会使进程的运行变慢.

首先，硬件必须知道当前正在运行的进程的页表的位置 (我们暂且假设页表存储于操作系统管理的物理内存上)
我们假设一个**页表基址寄存器** (page-table base register) 包含页表的起始位置的物理地址.
为找到想要的页表项 PTE 的位置，硬件将执行以下功能:

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

其中 VPN_MASK 将被设置为合适的值以从完整的虚拟地址中挑选出 VPN 位
SHIFT 也将被设置为合适的值，这样我们就可通过逻辑右移获得虚拟页号 VPN
然后我们用该值作为页表基址寄存器指向的 PTE 数组的索引.

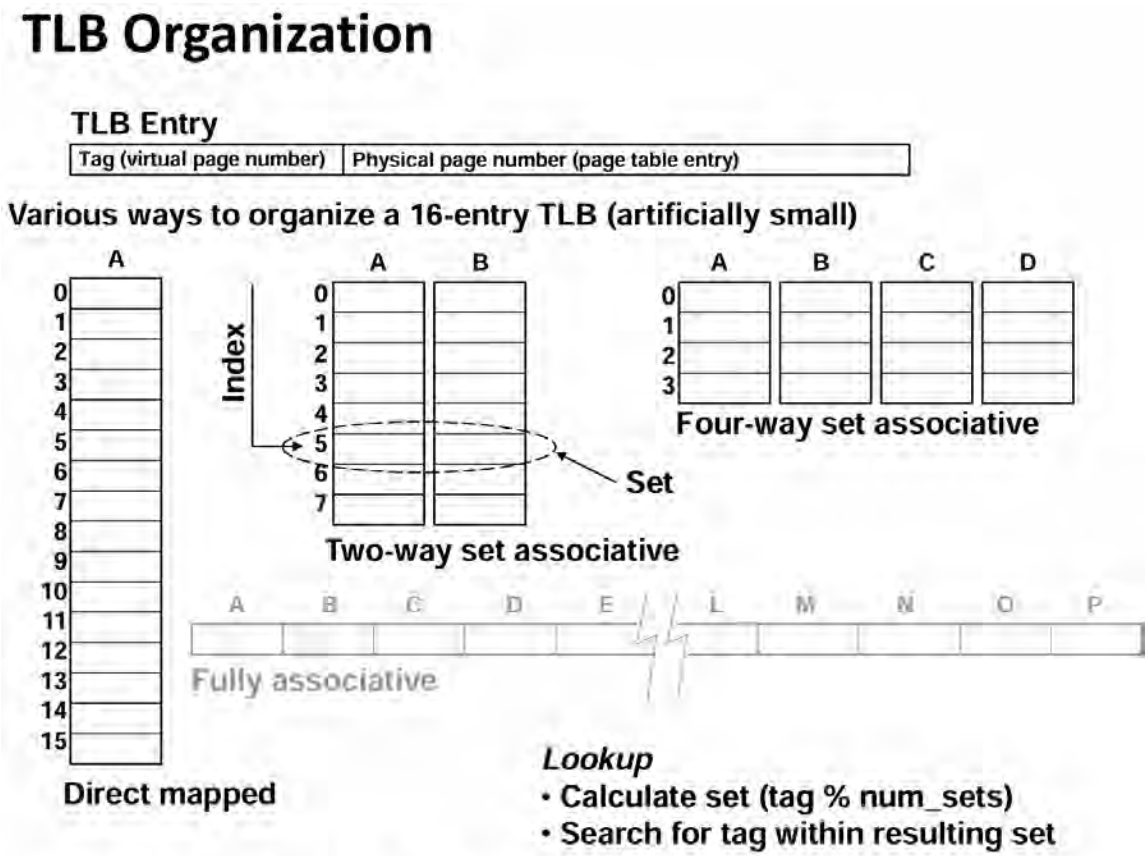
接下来我们从页表项 PTE 中读取物理页号 PPN，左移 SHIFT 位，
并与来自虚拟地址的页内偏移量 OFFSET 连接起来得到所需的物理地址.

```
// Extract the VPN (Virtual Page Number) from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT;  // Ensure the correct bit shift

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE));  // PTBR is the Page Table Base Register

// Fetch the PTE (Page Table Entry)
PTE = AccessMemory(PTEAddr);  // Access memory to get the PTE

// Check if the process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT);  // If the page is invalid, raise segmentation fault
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT);    // If access is not allowed, raise protection fault
else
    // Access is OK: form the physical address and fetch the data
    offset = VirtualAddress & OFFSET_MASK;  // Extract the offset from the virtual address
    PhysAddr = (PTE.PPN << PPN_SHIFT) | offset;  // Form the physical address using PPN and
offset
    Register = AccessMemory(PhysAddr);  // Access the memory at the physical address
```

可以看到，对于每个内存引用，分页都需要我们执行一个额外的内存引用以将虚拟地址转换为物理地址.
在这种情况下，可能会使进程减慢两倍甚至更多.

2.6.3 TLB

想让某些东西更快，操作系统通常需要一些来自硬件的帮助.
我们要增加**地址转换缓存** (address-translation cache)
(由于历史原因，它通常被称为**地址转换旁路缓冲器**, translation-lookside buffer, TLB)



对每次内存访问，硬件先检查 TLB，看看其中是否有期望的转换映射.
如果有，就完成转换 (很快)，不用访问页表 (尽管那里有全部的转换映射)

假定使用最简单的线性页表 (即数组) 和硬件管理的 TLB (即硬件承担许多页表访问的责任)
那么 TLB 参与的地址转换算法大致如下:

```
// Step 1: Extract VPN (Virtual Page Number) from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT;

// Step 2: Perform TLB lookup for the VPN
(Success, TlbEntry) = TLB_Lookup(VPN);

if (Success == True) // TLB Hit
{
    // Step 3: Check access permissions using ProtectBits in the TLB entry
    if (CanAccess(TlbEntry.ProtectBits) == True)
    {
        // Step 4: Extract the offset from the virtual address and form the physical address
        Offset = VirtualAddress & OFFSET_MASK;
        PhysAddr = (TlbEntry.PPN << SHIFT) | Offset;
```

```
        // Step 5: Access the memory at the physical address
        Register = AccessMemory(PhysAddr);
    }
    else
        // Step 6: Raise a protection fault if access is denied
        RaiseException(PROTECTION_FAULT);
}
else  // TLB Miss
{
    // Step 7: Calculate the Page Table Entry (PTE) address
    PTEAddr = PTBR + (VPN * sizeof(PTE));

    // Step 8: Access memory to retrieve the PTE
    PTE = AccessMemory(PTEAddr);

    // Step 9: Check if the page is valid
    if (PTE.Valid == False)
        // Step 10: Raise a segmentation fault if the page is invalid
        RaiseException(SEGMENTATION_FAULT);
    else if (CanAccess(PTE.ProtectBits) == False)
        // Step 11: Raise a protection fault if access is denied
        RaiseException(PROTECTION_FAULT);
    else
    {
        // Step 12: Insert the valid PTE into the TLB (for faster future lookups)
        TLB_Insert(VPN, PTE.PPN, PTE.ProtectBits);

        // Step 13: Retry the instruction after inserting into the TLB
        RetryInstruction();
    }
}
```

以前造硬件的人不太相信那些搞操作系统的人，因此硬件全权处理 TLB 未命中.
为了做到这一点，硬件必须知道页表在内存中的确切位置 (通过页表基址寄存器)，以及页表项的确切格式.
当发生 TLB 未命中时， 硬件会 "遍历" 页表，找到正确的页表项，取出想要的转换映射，用它更新 TLB，并重试该指令.
这样的 "旧" 体系结构有硬件管理的 TLB
这要求硬件有复杂的指令集 (又称**复杂指令集计算机**, Complex Instruction Set Computer, CISC)

> 值得说明的是，更新 TLB 是为了未来更高的命中率
> 这得益于大多数程序运行过程中内存访问的**空间局部性** (spatial locality) 和**时间局部性** (temporal locality)

更现代的体系结构都是**精简指令集计算机** (Reduced Instruction Set Computer, RISC)
它们有所谓的软件管理的 TLB
当发生 TLB 未命中时，硬件系统会抛出一个异常 (`RaiseException(TLB_MISS)`)
这会暂停当前的指令流，将用户模式提升为内核模式，跳转至陷阱处理程序.
这个陷阱处理程序是操作系统的一段代码，用于处理 TLB 未命中.
这段代码在运行时，会查找页表中的转换映射，然后更新 TLB，并从陷阱返回.
最后硬件会重试该指令 (自然会发生 TLB 命中)

软件管理的 TLB 的主要优势是**灵活性**:
操作系统可以用任意数据结构来实现页表，而不需要改变硬件.
另一个优势是**简单性**:
硬件不需要对 TLB 未命中做太多工作，它只负责抛出异常，然后操作系统的未命中处理程序会负责剩下的工作.

但软件管理的 TLB 有几个细节需要注意:

- ① 与服用于系统调用的陷阱返回 (继续运行陷入操作系统之后的那条指令) 不同的是，
  处理 TLB 未命中的陷阱返回必须从导致陷阱的指令继续执行.
  因此根据陷阱或异常的原因，系统在陷入内核时必须保存不同的程序计数器，以便将来能够正确地继续执行.
- ② 操作系统要避免 TLB 未命中的无限递归，即 TLB 未命中陷阱处理程序自身也 TLB 未命中 (有点好笑，不是吗?)
  这个问题有很多解决方案:

例如可以把 TLB 未命中陷阱处理程序直接放到物理内存中 (由于它们没有映射过，故不用地址转换) 或者在 TLB 中保留一些项，记录永久有效的地址转换，并将其中一些留给陷阱处理代码.

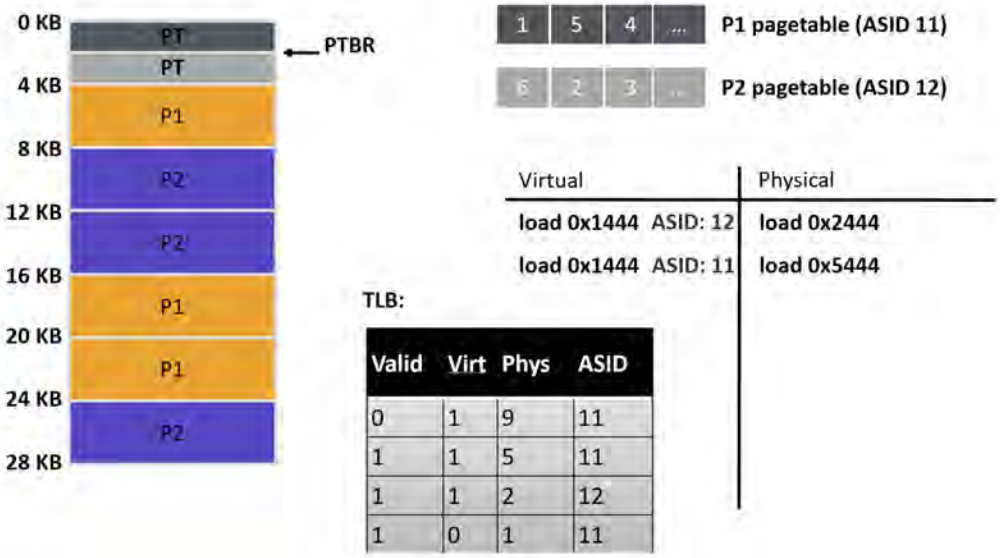
为避免在进程切换时清空 TLB，我们在 TLB 的表项中加入 asid，用 VPN 和 asid 的组合来进行匹配:

Context Switches

- **What happens if a process uses cached TLB entries from another process?**
  - Two processes access their own VPN 0
- **Solutions?**

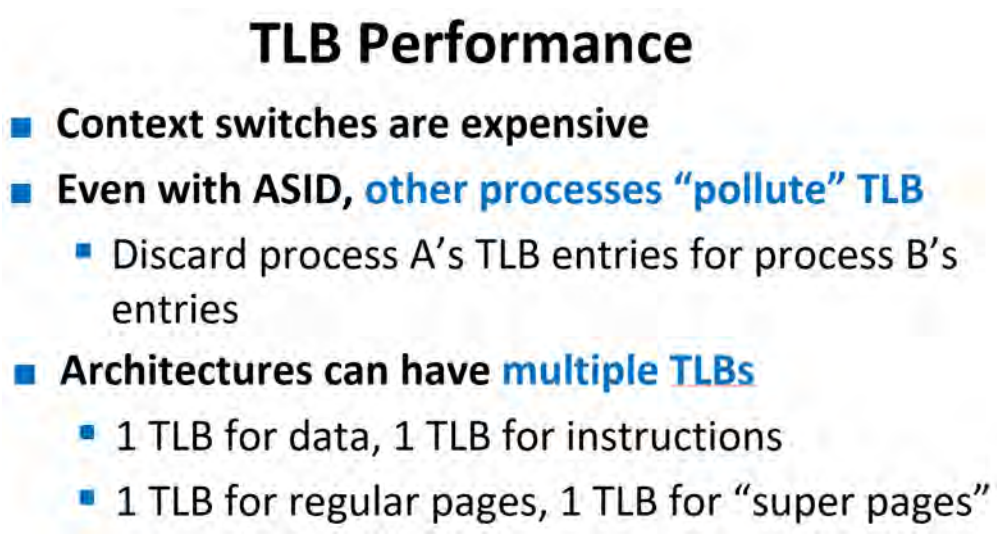
1. **Flush TLB on each switch**
   - Costly; lose all recently cached translations
2. **Track which entries are for which process**
   - Address Space Identifier
   - Tag each TLB entry with an 8-bit ASID
     - how many ASIDs do we get?
     - why not use PIDs? (e.g., 8 bits for the ASID versus 32 bits for a PID)

TLB Example with ASID



Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

但是进程切换不可避免地会导致 TLB 的表项被 "污染"，即一个进程可能会驱逐其他进程的表项.



2.6.4 多级页表

我们现在来解决分页引入的第二个问题: 线性页表太大了.

一个典型的 32 位地址空间带有 4KB 的页
因此其 32 位虚拟地址的高 20 位为虚页号 VPN，低 12 位为页内偏移量.

一个 20 位的 VPN 意味着，操作系统必须为每个进程管理 $2^{20} \approx 10^6$ 个地址转换.
如果每个**页表条目** (Page Table Entry, PTE) 需要 4 个字节来保存物理页号及其他标记，
那么每个页表就需要 4MB 内存，这非常大.
想象一下有 100 个进程 (这在现代系统中并不罕见) 在运行:
这意味着操作系统会需要 400MB 内存，仅仅是为了这些地址转换!
我们甚至不敢想 64 位地址空间的页表该有多大.

一种简单的解决方法就是使用更大的页:
页的大小每增大一倍，页表大小便缩小到原来的二分之一.
但这个方法的缺点也很明显:
过大的内存页会导致页内空间的浪费 (因为进程可能只用这些大页面的一小部分)，这称为**内部碎片**问题.
因此大多数系统都使用较小的页面，例如 4KB (如 x86)

如何去掉页表中的所有无效区域，而不是将它们全部保留在内存中?
一种非常有效的方法称为**多级页表** (multi-level page table)
其基本思想很简单: 将页表分成页大小的单元.
如果整页的页表项 PTE 无效，就不分配该页的页表.

为了追踪页表的页是否有效 (以及如果有效，它在内存中的位置)
我们使用一个名为**页目录** (page directory) 的新结构.
一个简单的两级页表如图所示:

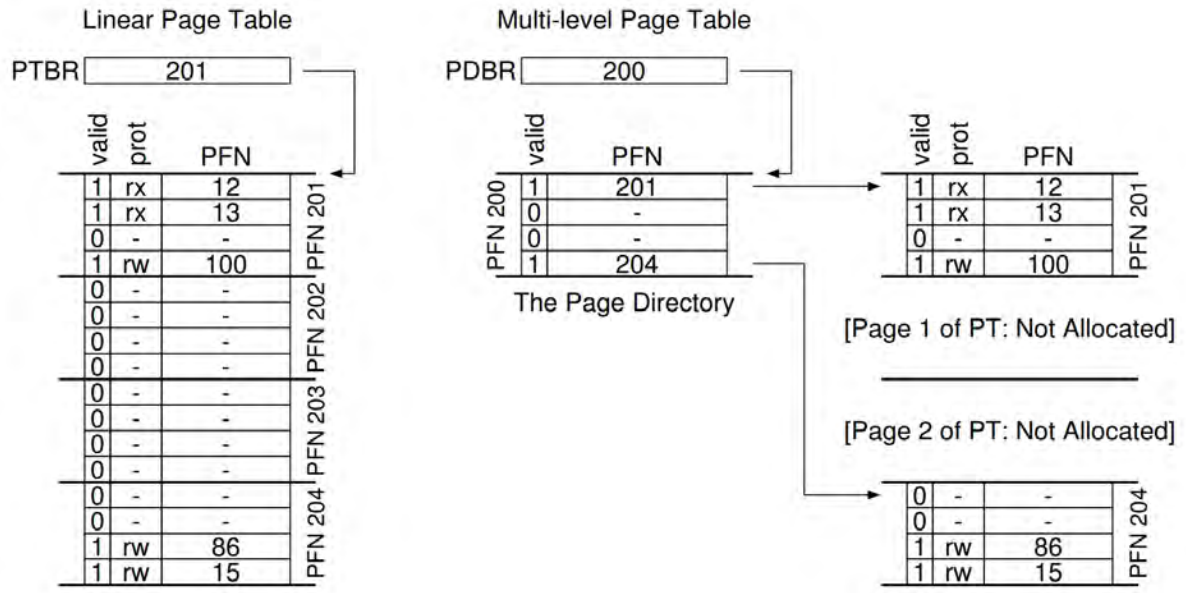


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

在上述两级页表中，页目录由多个**页目录项** (Page Directory Entries, PDE) 组成.
页目录项 PDE 至少拥有**物理页号** (Physical Page Number, PPN) 和**有效位** (valid bit)
这个有效位的含义是:

- 若某一页目录项 PDE 的有效位为 1，
  则意味着该项的物理页号 PPN 对应的页表页中至少有一个页表项 PTE 是有效的.
- 若某一页目录项 PDE 的有效位为 0，
  则意味着该项的物理页号 PPN 对应的页表页中的所有页表项 PTE 都是无效的，
  我们不为这个页表页分配内存空间.

多级页表的具有如下优势:

- 首先，多级页表分配的页表空间，与我们正在使用的地址空间内存量成比例.
  因此它通常很紧凑，并且支持稀疏的地址空间.
- 其次，如果仔细构建，则页表的每个部分都可以整齐地放入一页中，从而更容易管理内存.
  操作系统可以在需要分配或增长页表时简单地获取下一个空闲页，
  因此我们可以将页表页放在物理内存的任何地方.
  而线性页表作为按虚页号 VPN 索引的页表项 PTE 数组，必须连续地存储在物理内存中.

值得注意的是，多级页表也是有代价的:

- 当 TLB 未命中时，需要进行两次访存，才能从页表中获取正确的地址转换信息
  (一次用于查找页目录项 PDE，另一次用于查找页表项 PTE)
  而用线性页表只需要一次访存就可以完成地址转换.
  尽管在常见情况下 (TLB 命中) 性能是相同的，
  但当 TLB 未命中时，则会因较小的表而导致较高的时间成本和操作的复杂性.

下面的代码展示了二级页表的控制流算法:

```
// Step 1: Extract VPN (Virtual Page Number) from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT;

// Step 2: Perform TLB lookup for the VPN
(Success, TlbEntry) = TLB_Lookup(VPN);

if (Success == True)  // TLB Hit
{
    // Step 3: Check access permissions using ProtectBits in the TLB entry
    if (CanAccess(TlbEntry.ProtectBits) == True)
    {
        // Step 4: Extract the offset from the virtual address and form the physical address
```

```
        Offset = VirtualAddress & OFFSET_MASK;
        PhysAddr = (TlbEntry.PPN << SHIFT) | Offset;

        // Step 5: Access the memory at the physical address
        Register = AccessMemory(PhysAddr);
    }
    else
        // Step 6: Raise a protection fault if access is denied
        RaiseException(PROTECTION_FAULT);
}
else  // TLB Miss
{
    // Step 7: Calculate the Page Directory entry index (PDIndex)
    PDIndex = (VPN & PD_MASK) >> PD_SHIFT;

    // Step 8: Form the address of the Page Directory Entry (PDE)
    PDEAddr = PDBR + (PDIndex * sizeof(PDE));

    // Step 9: Fetch the Page Directory Entry (PDE) from memory
    PDE = AccessMemory(PDEAddr);

    // Step 10: Check if the page directory entry is valid
    if (PDE.Valid == False)
        // Step 11: Raise a segmentation fault if PDE is invalid
        RaiseException(SEGMENTATION_FAULT);
    else
    {
        // Step 12: PDE is valid, now calculate the Page Table entry index (PTIndex)
        PTIndex = (VPN & PT_MASK) >> PT_SHIFT;

        // Step 13: Form the address of the Page Table Entry (PTE)
        PTEAddr = (PDE.PPN << SHIFT) + (PTIndex * sizeof(PTE));

        // Step 14: Fetch the Page Table Entry (PTE) from memory
        PTE = AccessMemory(PTEAddr);

        // Step 15: Check if the page table entry is valid
        if (PTE.Valid == False)
            // Step 16: Raise a segmentation fault if PTE is invalid
            RaiseException(SEGMENTATION_FAULT);
        else if (CanAccess(PTE.ProtectBits) == False)
            // Step 17: Raise a protection fault if access is denied
            RaiseException(PROTECTION_FAULT);
        else
        {
            // Step 18: Insert the valid PTE into the TLB for future lookups
            TLB_Insert(VPN, PTE.PPN, PTE.ProtectBits);

            // Step 19: Retry the instruction after TLB update
            RetryInstruction();
        }
    }
}
```

考虑一个 16KB 的小地址空间，页大小为 64B
因此虚拟地址有 $\log_2(16 \times 2^{10}) = 14$ 位，页内偏移量有 $\log_2(64) = 6$ 位，虚页号 VPN 有 $14 - 6 = 8$ 位.
即使地址空间只有很小一部分在使用，线性页表也会有 $2^8 = 256$ 个项:

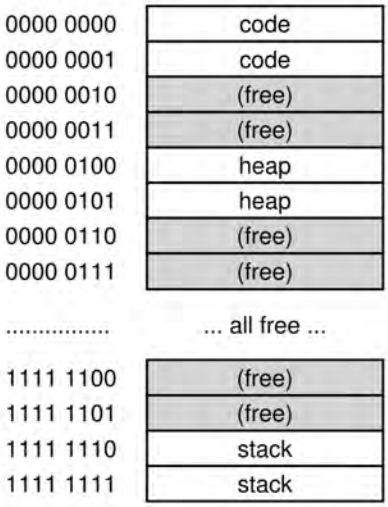
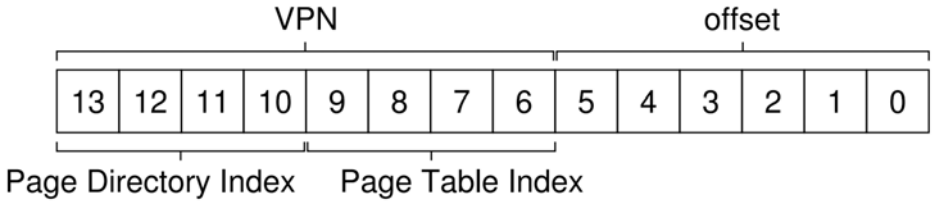


Figure 20.4: **A 16KB Address Space With 64-byte Pages**

要为这个地址空间构建一个两级页表，我们从完整的线性页表开始，将它分解成页大小的单元.
假设线性页表的页表项 PTE 的大小是 4B，则页表大小为 $256 \times 4\mathrm{B} = 1\mathrm{KB}$
鉴于我们的页大小为 64B，故 1KB 的线性页表可以分为 $1\mathrm{KB}/64\mathrm{B} = 16$ 个页表页.
因此页目录具有 16 个页目录项，需要虚拟页号 VPN 的高 $\log_2(16) = 4$ 位作为**页目录索引** PDIndex.
而虚拟页号 VPN 剩下的位为**页表索引** PTIndex，它相当于页表页内的偏移量.



一旦从虚拟页号 VPN 中提取了页目录索引 PDIndex，我们就可以得到页目录项的地址:

```
PDEAddr = PageDirBaseReg + (PDIndex * sizeof(PDE))
```

其中 PageDirBaseReg 是页目录的基地址寄存器的值.
然后我们根据 PDEAddr 来访问该页目录项 PDE:

- 若该页目录项 PDE 被标记为无效，则该访存操作无效.
- 若该页目录项 PDE 被标记为有效，则我们需要前往物理页号 PPN 指定的页表页访问页表项 PTE

  ```
  PTEAddr = (PDE.PPN << SHIFT) + (PTIndex * sizeof(PTE))
  ```

  在本例中 SHIFT 应当取 $4 + 2$，即 PDIndex 的位数 4 加上 PTE 大小造成的字节偏移量 2.
  物理页号 PPN 的位数多少取决于物理内存的大小.
  我们假设物理内存大小为 16KB，则物理页号 PPN 的位数为 $\log\left(16 \times 2^{10}\right) - 6 = 8$

下图给出了一个二级页表的映射状态:
(其中仅有第 0 页表页和第 15 页表页是有效的，分别被映射到第 100 和 101 物理页)

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

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

考虑虚拟地址 $0x3F80 = 0b11\_1111\_1000\_0000$ 的转换:
(当然，这只会在 TLB 未命中的情况下发生)

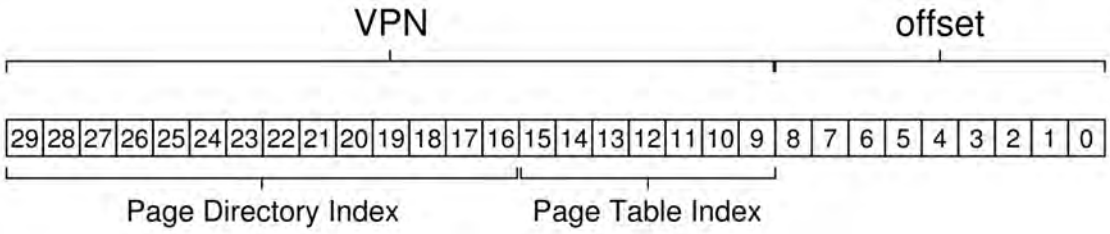
- **① 第一次访存:**
  我们使用 $\mathrm{VPN} = 0b1111\_1110$ 的前四位作为页目录索引: $\mathrm{PDIndex} = 0b1111$
  结合页目录基地址后，指向第 15 页目录项，对应第 15 页表页.
  我们发现它是有效的，且对应的物理页号为 $101 = 0b0110\_0101$
- **② 第二次访存:**
  接下来我们使用 $\mathrm{VPN} = 0b1111\_1110$ 的后四位作为页表索引: $\mathrm{PTIndex} = 0b1110$
  它指向的是第 101 物理页的第 $0b11\_10 = 14$ 个页表项 PTE 的首地址，即第 $0b11\_1000$ 个字节

  > 这个字节的地址计算是这样的:
  >
  > $$\begin{aligned}\mathrm{PTEAddr} &= (\mathrm{PDE.PPN} \ll \mathrm{SHIFT}) + (\mathrm{PTIndex} * \mathrm{sizeof(PTE)}) \quad \left(\text{note that } \begin{cases}\mathrm{SHIFT} = 4 + 2 = 6 \\ \mathrm{sizeof(PTE)} = 4\end{cases}\right) \\ &= (0b0110\_0101 \ll 6) + (0b1110 * 4) \\ &= 0b01\_1001\_0100\_0000 + 0b11\_1000 \\ &= 0b01\_1001\_0111\_1000\end{aligned}$$

  我们发现第 101 物理页的第 14 个页表项是有效的，且对应的物理页号是 $55 = 0b0011\_0111$
- **③ 第三次访存:**
  最后我们连接物理页号 $55 = 0b0011\_0111$ 和从虚拟地址截取的偏移量 $0b00\_0000$
  得到虚拟地址 $0x3F80 = 0b11\_1111\_1000\_0000$ 对应的物理地址 $0b00\_1101\_1100\_0000 = 0x0DC0$

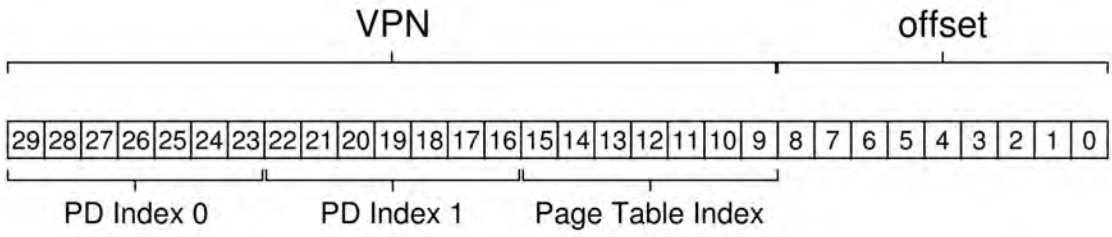
请记住我们构建多级页表的目标: 使页表的每一部分都能放入一个页.
到目前为止，我们只考虑了页表本身. 但如果页目录太大，该怎么办?
因此我们真正的目标是: **让多级页表的每一个部分都放入一页**

考虑 $1\mathrm{GB}$ 的地址空间，设页大小为 $512\mathrm{B}$ (假设按字节寻址)
则虚拟地址有 $\log_2(1 \times 2^{30}) = 30$ 位，页内偏移量有 $\log_2(512) = 9$ 位，虚页号 VPN 有 $30 - 9 = 21$ 位.
设页表项 PTE 的大小为 $4\mathrm{B}$，则每个页表页可以放置 $512\mathrm{B}/4\mathrm{B} = 128$ 个页表项，需要 $\log_2(128) = 7$ 位索引.
因此页目录索引有 $21 - 7 = 14$ 位.



我们发现页目录项有 2^{14} 个
假设每个页目录项的大小也是 $4\mathrm{B}$，则每个页只能存放 $512\mathrm{B}/4\mathrm{B} = 128$ 个页目录项
因此要放置所有的页目录项需要 $2^{14}/128 = 128$ 页.
这表明我们 "让多级页表的每一个部分都放入一页" 的目标失败了.

为了解决这个问题，我们再加一层页目录，按如下方式分割虚拟地址:



当索引一级页目录时，我们使用虚页号 VPN 的最高 7 位 $\mathrm{PDIndex0}$ 结合一级页目录基地址来访问一级页目录项.
如果有效，则通过组合来自一级页目录项的物理页号 $\mathrm{PPN1}$ 和虚页号 VPN 的下一部分 $\mathrm{PDIndex1}$ 来访问二级页目录项.
如果有效，则通过组合来自二级页目录项的物理页号 $\mathrm{PPN2}$ 和虚页号 VPN 的最后部分 $\mathrm{PTIndex}$ 来访问页表项.
如果有效，则通过组合页表项的物理页号 PPN 和来自虚拟地址的偏移量 OFFSET 得到物理地址来访问存储单元.

周喆老师提供了一个生动的例子:

- 线性页表 big array:

```
int a[1024][1024]; // continuous 4MB space (which may not be allocated successfully)
```

- 二级页表:

```
// note that 4KB space (the size of one page) is guaranteed to be allocated
successfully
int ** a = malloc(4 * 1024); // continuous 4KB space (stands for Page Directory)
for(i=0; i<1024; i++)
    a[i] = malloc(4 * 1024); // continuous 4KB space (stands for the i-th "Page" of
Page Table)
```

操作系统有一个原则: 申请小于等于一个页大小的连续内存一定要保证成功.
因此多级页表的每一级最好都用满一个页大小 (最高一级可以不用满一页，这样页内空间浪费得最少)

(周喆老师的补充)
64 位系统 (假设页大小为 $4\mathrm{KB}$，因此页内偏移量为 12 位)
假设每个页表项 $8\mathrm{B}$，那么我们可以构造 6 级页表: $64 - 12 = 52 = 7 + 5 \times 9$
但是这会造成地址转换变慢.
对于 6 级页表来说，每次虚拟地址到物理地址的转换，最差情况下需要 6 次内存访问，外加一次访问实际的物理内存.
因此总共需要 7 次内存访问 (包括访问最终的物理内存)

实际上 64 位系统没有一个好的方法可以把级联页表的级数降下来.
但我们又不能返回 32 位，这样我们只能使用 $4\mathrm{GB}$ 的物理内存.
(不过 PAE (Physical Address Extension) 可以实现让 32 位系统使用 32 位指针访问超过 $4\mathrm{GB}$ 的物理内存)

- **Page Walk Caches** 是一种专门用于缓存页表层次中的部分或全部页表项的缓存
  它仍然可以显著减少每次完整的多级页表遍历所需的时间.

- 在 Intel 的一些处理器中，有一种名为 **Page Miss Handler (PMH)** 的硬件模块，专门负责在 TLB miss 时加速多级页表的遍历.
  PMH 可以在硬件级别并行化页表查找，从而减少每一级页表查找的时间.
  这种方式使得地址转换的开销更小，从而提升系统性能.
- 实际上，大多数 64 位处理器只使用 48 位.
  x86-64 架构在实际实现中，只支持 48 位虚拟地址空间，对应 $256\mathrm{TB}$ 的虚拟地址空间.
  并采用 5 级页表 $48 = 9 + 9 + 9 + 9 + 12$ (因为各级页表项的大小从 $4\mathrm{B}$ 变为 $8\mathrm{B}$)
  而且 64 位地址是 48 位 CPU 通过**符号扩展**脑补出来的，高 16 位要么全零 $(0\mathrm{x}0000)$，要么全一 $(0\mathrm{xFFFF})$
  架构师提出从 $32 \to 39 \to 48 \to 56 \to 64$ 逐步奔向真正的 64 位系统.
  (现在服务器用的大多是实际 48 位 CPU)
- 有时较大的数组 (例如 $4\mathrm{MB}$) 进行随机访问，那么使用 $4\mathrm{KB}$ 的页会造成 TLB 未命中率增加.
  这时我们可以使用 **Huge Page**
  例如将一级页目录当作底层页表，则页大小增大 2^9 倍 (64 位系统中)，得到 $2\mathrm{MB}$ 的页
  将二级页目录当作底层页表，则页大小增大 2^{18} 倍，得到 $1\mathrm{GB}$ 的页
  这在虚拟机实现中非常好用:
  假设在 64 位系统中运行 64 位虚拟机，假设它们的可用地址都只有 48 位，则它们各有一个 5 级页表.
  那么一次 TLB 未命中就需要 $5 + 5 = 10$ 次访存才能得到真正的物理地址，这会使地址转换非常慢.
  此时我们可以使用 $1\mathrm{GB}$ 的页，于是虚拟机可以只对应 TLB 中的寥寥数个表项，命中率大大提高.

2.7 页交换

2.7.1 页的交换

到目前为止，我们一直假定地址空间非常小，能放入物理内存.
现在我们放松这些假设，即我们需要支持许多进程同时运行的巨大地址空间.

> 为什么我们要为进程支持巨大的地址空间?
> 答案还是易用性.
> 有了巨大的地址空间，我们就不必担心程序的数据结构是否有足够空间存储，
> 只需自然地编写程序，根据需要分配内存.
> 这是操作系统提供的一个强大的假象，使我们的生活简单很多.

我们要做的第一件事情就是在硬盘上开辟一部分空间用于物理页的移入和移出.
在操作系统中，我们称这样空间称为**交换空间** (swap space)
因为我们将内存中多出的页存储到其中，并在需要的时候又取回去.
假设操作系统能够以页为单元读写交换空间.
为了达到这个目的，操作系统需要记住给定页的**硬盘地址** (disk adderss)
交换空间的大小是非常重要的，它决定了系统在某一时刻能够使用的最大内存页数.
简单起见，我们假设它非常大，这能让操作系统假装内存远大于实际物理内存.

一个简单的例子如图所示:
(其中进程 $0, 1, 2$ 共享物理内存，但它们都有一些有效页在交换空间中，而进程 3 此时没有运行)

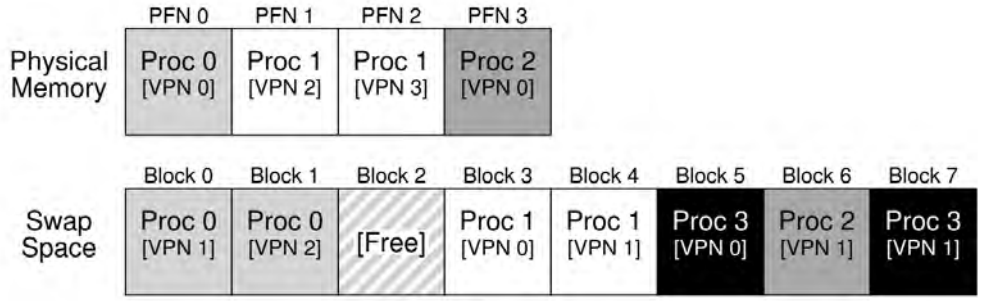


Figure 21.1: **Physical Memory and Swap Space**

现在需要在系统中增加一些更高级的机制，来支持内存与硬盘的页交换.
简单起见，假设有一个硬件管理 TLB 的系统.

先回想一下内存引用发生了什么:
正在运行的进程生成虚拟内存引用 (用于获取指令或访问数据)，硬件将其转换为物理地址，再从内存中获取所需数据.

硬件首先从虚拟地址获得虚页号 VPN，检查 TLB 是否命中.

- 如果 TLB 命中，则获得物理页号 PPN，产生最终的物理地址用于访存，这会很快 (因为不需要额外的内存访问)
- 如果在 TLB 中找不到 VPN (即 TLB 未命中),
  则硬件在内存中查找页表 (使用页表基址寄存器 PTBR 中的基址)，并使用 VPN 作为索引查找该页的页表项 PTE.
  - 如果页无效 (有效位为 0)，则产生**段错误** (segmentation fault)
    (这是访问一个未被分配的地址时会发生的情况)
  - 如果页有效 (有效位为 1)，则该地址是合法的.
    - 如果该有效页存在于物理内存中 (存在位为 1),
      则硬件从 PTE 中获得物理页号 PPN，将其插入 TLB，并重试该指令，这次产生 TLB 命中.
    - 如果该有效页不在物理内存中 (存在位为 0)
      则会发生**页错误** (page fault)，此时操作系统才介入 (跳转到预先约定的内核处理程序进行处理)
      (页错误又称**缺页中断**, 事实上它应该称为 "页未命中", page miss, 因为它是完全合法的访问)

      操作系统接收到页错误后,
      首先会检查触发页错误的虚拟地址是否对应于一个共享页 (例如通过维护一个共享页的元数据结构)
      共享页是被多个进程共享的内存页，比如共享库或相同进程间的共享数据.
      Linux 有一个思想叫 "一切皆文件"，即几乎所有资源 (包括内存) 都可以被视为文件.
      如果页面是从文件映射而来的，操作系统会检查该文件是否在物理内存中
      - 如果页面已经存在于物理内存中 (很可能是共享页),
        那么操作系统简单地为当前进程的页表建立映射，更新页表中的 PTE (并将存在位设置为 1)

        > 若两个进程都要使用库 LibC (只读)，则操作系统只拷贝一份 LibC 进入物理内存.
        > 这就造成了**共享页错误**的问题:
        > 当其中一个进程刚刚被创建时，LibC 可能已经在物理内存中了，但该进程的页表中没有建立到 LibC 的映射
        > 此时我们只需简单地建立映射即可解决该页错误.

      - 否则意味着我们正在访问的内存页被操作系统交换到了硬盘上,
        操作系统将把该页从磁盘加载回物理内存，并更新页表中的 PTE (将存在位设置为 1)

        预页面调入 (prepaging) 可以减少此类页错误的发生.
        它通过提前加载可能会被访问的页面到内存中，从而减少由于页面不在物理内存中而引发的页错误.
        为了改善预页面调入的效果，可以结合用户提供的提示信息.
        以下是一些用户可能提供的提示:
        ① **MADV_WILLNEED:** 用户告知操作系统某个页面将会在未来被访问，操作系统可以据此提前加载该页面.
        ② **MADV_DONTNEED:** 用户表明某个页面不再需要，操作系统可以释放该页面的内存.
        ③ **MADV_SEQUENTIAL:** 用户提示操作系统页面访问将是顺序的，操作系统可以采用预取策略来加载后续页面.
        ④ **MADV_RANDOM:** 用户声明页面访问将是随机的，操作系统可以调整预加载策略以适应这种模式.

      页错误处理结束后，硬件会重新执行触发页未命中的指令，并产生 TLB 命中.

回想一下，在 TLB 未命中的情况下，我们有两种类型的系统:
硬件管理的 TLB (硬件在页表中找到需要的转换映射) 和软件管理的 TLB (操作系统执行查找过程)
但几乎所有的系统都在软件中处理页错误.
尽管硬件设计者不愿意信任操作系统做所有事情，但他们依然选择让操作系统来处理页错误，主要原因如下:

- ① 性能因素:
  页错误导致的硬盘操作很慢.
  即使操作系统需要很长时间来处理故障，但相比于硬盘操作，这些额外开销是很小的.

- ② 设计复杂度因素:
  为了能够处理页错误，硬件必须了解交换空间、如何向硬盘发起 $\mathrm{I/O}$ 操作，以及很多其他细节.
  这实在是太复杂了.

如果正在访问的有效页已经被交换到硬盘上了，那么操作系统需要将该页交换到内存中.
为知道所需页在哪里，操作系统使用页表项 PTE 的某些位来存储硬盘地址.
当操作系统接收到页错误时，它会在 PTE 中查找地址，并将请求发送到硬盘，将页读取到内存中.

在硬盘 $\mathrm{I/O}$ 进行过程中，进程将处于阻塞 (blocked) 状态.
当硬盘 $\mathrm{I/O}$ 完成时，操作系统会更新页表，将此页的页表项 PTE 的存在位设为 1，并记录物理页号 PPN 字段，最后重试导致页错误的指令.
这仍会造成一次 TLB 未命中，因为引发页错误后 TLB 中对应的表项会被 flush 掉 (因为它现在是无效的了)，在页错误处理程序对页表项进行修改，但这对 TLB 来说是不可见的.
因此只有重试导致页错误的指令，再触发一次 TLB 未命中之后，TLB 中才会有正确的表项.

下面的代码展示了硬件在地址转换过程中所做的工作:

```
// Extract the Virtual Page Number (VPN) from the Virtual Address.
// The VPN is calculated by masking the relevant bits of the VirtualAddress
// (using VPN_MASK) and then shifting it right by the SHIFT value.
VPN = (VirtualAddress & VPN_MASK) >> SHIFT;

// Perform a TLB (Translation Lookaside Buffer) lookup for the VPN.
// This function returns a tuple: Success (indicating if the lookup was successful)
// and TlbEntry (the actual TLB entry if it was a hit).
(Success, TlbEntry) = TLB_Lookup(VPN);

// Check if the TLB lookup was successful (i.e., a TLB hit).
if (Success == True) // TLB Hit
{
    // Check if the access to the page is allowed based on the protection bits
    // stored in the TLB entry (i.e., protection check).
    if (CanAccess(TlbEntry.ProtectBits) == True)
    {
        // Extract the page offset from the Virtual Address by applying the OFFSET_MASK.
        Offset = VirtualAddress & OFFSET_MASK;

        // Compute the physical address by combining the TLB entry's Physical Page Number
(PPN),
        // shifted by SHIFT to account for page size, and the offset.
        PhysAddr = (TlbEntry.PPN << SHIFT) | Offset;

        // Access memory at the calculated physical address and load the value into a
register.
        Register = AccessMemory(PhysAddr);
    }
    else
    {
        // If the access is not allowed (based on protection bits), raise a protection
fault.
        RaiseException(PROTECTION_FAULT);
    }
}
else // TLB Miss
{
    // If the TLB miss occurs, the system must fall back to the page table.
    // Compute the Page Table Entry (PTE) address. This is done by adding the VPN offset,
    // multiplied by the size of a PTE, to the base of the page table (PTBR).
    PTEAddr = PTBR + (VPN * sizeof(PTE));

    // Access memory to fetch the PTE from the page table.
    PTE = AccessMemory(PTEAddr);
```

```
    // Check if the PTE is marked as valid.
    // If the PTE is invalid, it means the virtual address is not mapped to any page.
    if (PTE.Valid == False)
    {
        // Raise a segmentation fault if the PTE is invalid.
        RaiseException(SEGMENTATION_FAULT);
    }
    else
    {
        // Check if the protection bits in the PTE allow access.
        if (CanAccess(PTE.ProtectBits) == False)
        {
            // Raise a protection fault if the access is not allowed.
            RaiseException(PROTECTION_FAULT);
        }
        // Check if the page is present in physical memory (i.e., not swapped to disk).
        else if (PTE.Present == True)
        {
            // For hardware-managed TLBs, insert the new translation into the TLB.
            TLB_Insert(VPN, PTE.PPN, PTE.ProtectBits);

            // Retry the instruction now that the TLB has the valid mapping.
            RetryInstruction();
        }
        else if (PTE.Present == False)
        {
            // If the page is not present in memory (swapped out or unallocated),
            // raise a page fault to trigger the OS to load the page.
            RaiseException(PAGE_FAULT);
        }
    }
}
```

下面是操作系统的页错误处理程序 (page-fault handler):

```
// Step 1: Find a free physical page.
// The system attempts to allocate a free physical page frame.
// If a free page is found, its Physical Page Number (PPN) is returned.
PPN = FindFreePhysicalPage();

// Step 2: Check if a free physical page was found.
// If the function `FindFreePhysicalPage()` returns -1, it indicates that
// no free physical pages are available.
if (PPN == -1) // No free page found
{
    // Step 3: Evict an existing page from memory to make room.
    // When there are no free pages, a page replacement algorithm is invoked
    // (such as Least Recently Used or FIFO) to select a page to evict.
    // The evicted page's PPN will be returned and reused.
    PPN = EvictPage(); // Run replacement algorithm
}

// Step 4: Read the page from disk into memory.
// Once a free or evicted physical page is identified, the data for the page
// that caused the page fault must be read from disk (swap space or a page file).
// `DiskRead` transfers the page from the disk (using the page table entry's DiskAddr)
// into the selected physical page (represented by PPN).
// This operation will put the process to sleep as it waits for the I/O to complete.
DiskRead(PTE.DiskAddr, PPN); // Sleep (waiting for I/O)

// Step 5: Update the Page Table Entry (PTE).
// Mark the page as "present" in the page table. This informs the system that
```

```
// the page is now loaded into memory and accessible.
PTE.present = True; // Update page table to mark page as present

// Step 6: Update the PPN in the Page Table Entry.
// Set the PPN (Physical Page Number) field of the page table entry to the
// physical page frame where the page has been loaded.
PTE.PPN = PPN; // Set PPN in PTE to indicate the new location in memory

// Step 7: Update the TLB with the new mapping.
// Insert the new virtual-to-physical address mapping (VPN -> PPN) into the TLB.
// The VPN (Virtual Page Number) is assumed to be accessible at this point,
// and the protection bits from the PTE are also added for access control.
TLB_Insert(VPN, PPN, PTE.ProtectBits); // Insert new mapping into TLB

// Step 8: Retry the instruction that caused the page fault.
// Now that the page is loaded into memory, the TLB is updated, and the
// page table is also updated, the system can retry the instruction that
// originally caused the page fault.
RetryInstruction(); // Retry instruction
```

在使用机械硬盘的情况下，
为优化页交换的性能，操作系统会将多个要换入硬盘的物理页**聚集** (cluster) 或**分组** (group)，
然后同时写到交换空间，这能减少硬盘的寻道和旋转开销，从而提高硬盘的效率.

把一些工作放在后台运行是一个好主意，这可以让这些操作合并执行.
例如要写入硬盘的数据可以先存放在内存缓冲区中，这有很多好处:

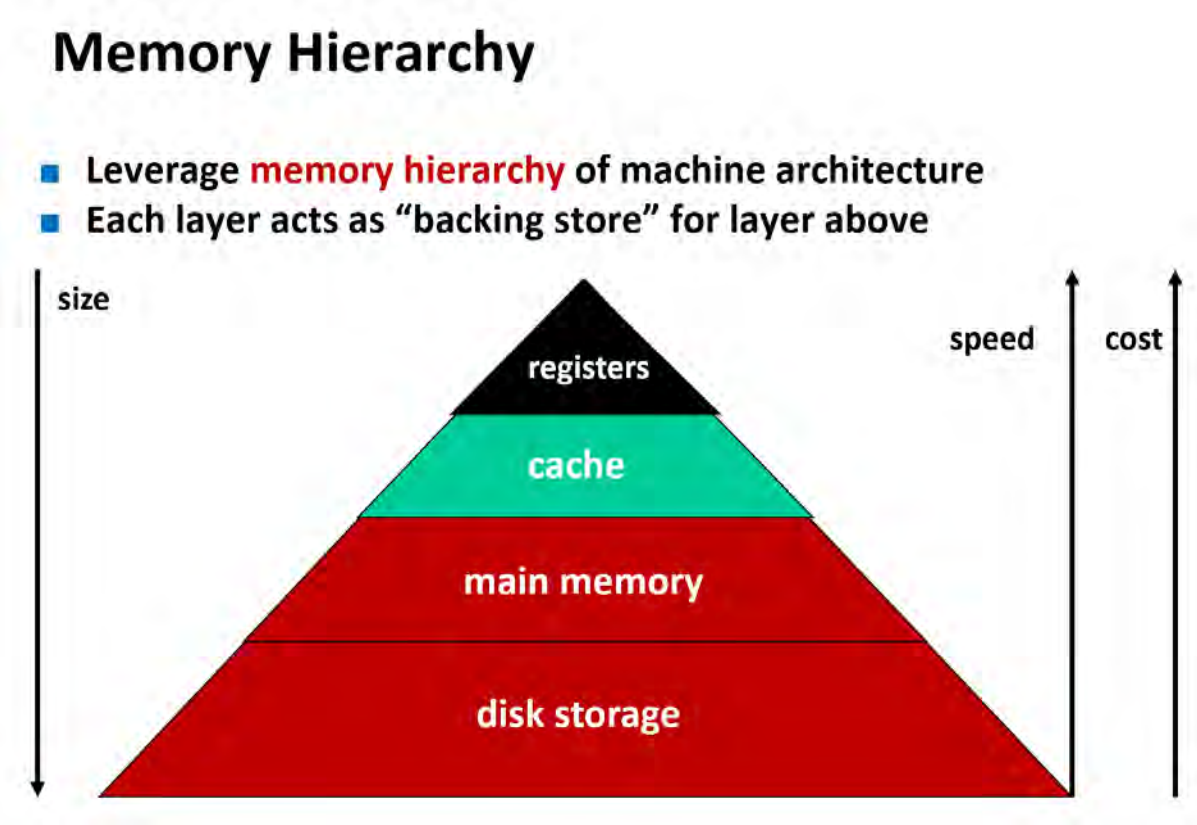
- 能够优化写入延迟 (因为数据写入到内存就可以返回)
- 提高硬盘效率 (因为硬盘可以将数据一次性写入)
- 可以减少无用操作 (例如文件很快又被修改或删除，那么它写入硬盘的操作就是无用的)
- 可以更好地利用系统空闲时间 (idle time)，从而更好地利用硬件资源

2.7.2 页交换策略

在 2.6.1 节描述的过程中，我们假设总是有足够的空闲内存来从存储交换空间换入的页.
当然实际情况可能并非如此，内存可能已经满了.
因此操作系统可能希望先换出若干页，以便为操作系统即将换入的新页腾留空间.
选择哪些页被换出或替换的过程，被称为**页交换策略** (page-replacement policy)
(换出的页需要将其页表项的存在位设为 0)

实际上，操作系统更倾向于主动地预留一小部分空闲内存.
为决定何时从内存中换出页，
操作系统会设置**低水位线** (Low Watermark, LW) 和**高水位线** (High Watermark, HW)
当少于 LW 个物理页可用时，后台负责释放内存的线程就会开始运行，直到有 HW 个可用的物理页.
这个后台线程称为**交换守护进程** (swap daemon)

(物理内存是一个巨大的缓存)
由于内存只包含系统中所有虚拟页的子集，因此可以将其视为虚拟内存的缓存 (cache)
因此在为这个缓存选择替换策略时，我们的目标是让缓存命中率尽可能高，即在内存中找到待访问页的概率最高.
(但是有些服务器的主存可能比硬盘还大，这是因为它们不需要持久化用户态数据，只要存在主存中就行了，例如实时数据分析)



知道了缓存命中和未命中的次数，就可以计算程序的**平均内存访问时间** (Average Memory Access Time, AMAT)

$$\mathrm{AMAT} := \mathrm{P}_{\mathrm{hit}} \cdot t_{\mathrm{hit}} + \mathrm{P}_{\mathrm{miss}} \cdot t_{\mathrm{miss}} = \frac{n_{\mathrm{hit}} \cdot t_{\mathrm{hit}} + n_{\mathrm{miss}} \cdot t_{\mathrm{miss}}}{n_{\mathrm{hit}} + n_{\mathrm{miss}}}$$

其中 n_{hit} 和 n_{miss} 分别为命中和未命中的次数，t_{hit} 和 t_{miss} 分别为命中和未命中的开销.

在计算机体系结构中，我们将未命中分为 3 类:

- **① 强制未命中 (compulsory miss):**
  这是因为缓存最开始是空的，对任何项目的第一次引用都将造成未命中.
- **② 容量未命中 (capacity miss):**
  这是由于缓存空间不足而不得已驱逐一个项目，以将新项目引入缓存而造成的未命中.
- **③ 冲突未命中 (conflict miss):**
  这是由于集合关联性 (set-associativity) (即硬件缓存中对项的放置位置有限制) 引发的替换所造成的未命中.
  但物理内存是完全关联的 (fully-associative) 的，即对页面可以存放的内存位置没有限制
  因此其冲突未命中就是容量未命中.

(1) 最优策略

Belady 提供了一个最优策略: 替换内存中在最远将来才会被访问到的页.
这个策略是无法真正实现的 (即使实现了也无法做到通用)，因为未来的访问我们是无法提前预知的.
尽管如此，最优策略在与其他策略的仿真比较中还是非常有用的.
通过与最优策略的比较，我们可以知道当前策略有多大的改进空间.
当我们的策略已经非常接近最优策略的表现时，我们便可停止做无谓的优化.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**

(2) FIFO

先进先出策略 (First In First Out, FIFO) 的实现非常简单:
页在进入物理内存时，简单地放入一个队列，当发生替换时，最早进入的页 (位于队列尾部) 先被踢出.

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2

Figure 22.2: **Tracing The FIFO Policy**

但 FIFO 策略无法确定页的重要性，即使页 0 的访问次数较多，FIFO 仍会将其踢出，因为它是第一个进入内存的.

Belady 发现了一个有趣的引用序列 $1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
可以证明在 FIFO 策略下，当缓存大小从 3 变到 4 时，缓存命中率反而降低了.
这种奇怪的现象称为 Belady 异常.
实际上，FIFO 策略 (以及随机策略, 它取决于运气) 不具有**栈特性** (stack property)
具有栈特性的算法，大小为 $N+1$ 的缓存自然能够完成大小为 N 的缓存的任务
因此当增大缓存时，缓存命中率至少保证不变，有可能提高.
但不具有栈特性的算法容易出现异常行为.

(3) LRU

任何像 FIFO 或随机驱逐这样简单的策略都可能会有一个共同的问题:
它可能会踢出一个重要的页，而这个页马上要被引用.
因此这样的策略不太可能达到最优，我们需要更智能的策略.

正如在调度策略所做的那样，为了提高后续的命中率，我们再次以历史的访问情况作为参考.
我们假设，如果某个程序在过去访问过某个页，则很有可能在不久的将来会再次访问该页.

- 越近被访问过的页，其再次被访问的可能性也就越大.
  这引导我们设计出**最近最少使用策略** (Least-Recently-Used, LRU)
- 一个页被访问的次数越频繁，其再次被访问的可能性也就越大.
  这引导我们设计出**最不经常使用策略** (Least-Frequently-Used, LFU)

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

硬件可以在每个页访问时更新内存中的时间字段
(时间字段可以在每个进程的页表中，或者在内存的某个单独的数组中，每个物理页有一个)
因此当页被访问时，时间字段将被硬件设置为当前时间.
在需要替换页时，操作系统可以扫描系统中所有页的时间字段以找到最近最少使用的页.
遗憾的是，随着系统中页数量的增长，为找到精确最少使用的页而扫描所有页的时间字段的代价太昂贵了.

从计算开销的角度来看，近似 LRU 更为可行 (又称为**时钟算法**, Clock Algorithm)，实际上这也是许多现代系统的做法.
这个策略需要为每个页增加一个**使用位** (use bit) (x86 中称为 $\mathrm{accessed}$ 位)
这些使用位可以存储在进程的页表中，或存储在某个数组中.
当页被引用时，硬件将使用位设置为 1，操作系统负责周期性地将使用位设置为 0
当必须进行页替换时，操作系统检查当前指向的页 P 的使用位是 1 还是 0
如果是 1，则意味着页面 P 最近被使用过，不适合被替换
然后将 P 的使用位设置为 0，时钟指针递增到下一页 $P+1$，重复上述过程直至找到一个使用位为 0 的页.

任何周期性地清除使用位，然后通过区分使用位是 1 和 0 来判定该替换哪个页的方法都是可行的近似 LRU 策略.
我们也可以在需要进行页替换时随机扫描各页
如果遇到一个页的引用位为 1，就清除该位 (即将它设置为 0)
直到找到一个使用位为 0 的页，将这个页替换掉.

在不要替换时，对于经过上一周期使用位仍为 0 的页，我们可以将其添加到 victim list 中，以供后续替换使用.
为给介于常用和不常用之间的页面一定机会，我们可以设置一个阈值，
只有 "使用位为 0" 的次数超过阈值的页才放入 victim list 中.

(4) 考虑脏页

如果页已被修改，即其修改位 (又称脏位) 为 1，则踢出它就必须将它写回磁盘，这很昂贵.
如果它没有被修改，踢出就没有成本: 该物理页可以简单地重用于其他目的而无须额外的 $\mathrm{I/O}$
因此有些操作系统更倾向于踢出干净页，而不是脏页.
在近似 LRU 策略的实现中，我们可以优先踢出未被修改过且近期未使用的页.
无法找到这种页时，再查找被修改过但近期未使用的页.

The End