

9.9 动态内存分配

虽然可以使用低级的 `mmap` 和 `munmap` 函数来创建和删除虚拟内存的区域，但是 C 程序员还是会觉得当运行时需要额外虚拟内存时，用动态内存分配器 (dynamic memory allocator) 更方便，也有更好的可移植性。

动态内存分配器维护着一个进程的虚拟内存区域，称为堆 (heap) (见图 9-33)。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长 (向更高的地址)。对于每个进程，内核维护着一个变量 `brk` (读做 “break”)，它指向堆的顶部。

分配器将堆视为一组不同大小的块 (block) 的集合来维护。每个块就是一个连续的虚拟内存片 (chunk)，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

- 显式分配器 (explicit allocator)，要求应用显式地释放任何已分配的块。例如，C 标准库提供一种叫做 `malloc` 程序包的显式分配器。C 程序通过调用 `malloc` 函数来分配一个块，并通过调用 `free` 函数来释放一个块。`C++` 中的 `new` 和 `delete` 操作符与 C 中的 `malloc` 和 `free` 相当。
- 隐式分配器 (implicit allocator)，另一方面，要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器 (garbage collector)，而自动释放未使用的已分配的块的过程叫做垃圾收集 (garbage collection)。例如，诸如 Lisp、ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

本节剩下的部分讨论的是显式分配器的设计和实现。我们将在 9.10 节中讨论隐式分配器。为了更具体，我们的讨论集中于管理堆内存的分配器。然而，应该明白内存分配是一个普遍的概念，可以出现在各种上下文中。例如，图形处理密集的应用程序就经常使用标准分配器来要求获得一大块虚拟内存，然后使用与应用相关的分配器来管理内存，在该块中创建和销毁图形的节点。

9.9.1 `malloc` 和 `free` 函数

C 标准库提供了一个称为 `malloc` 程序包的显式分配器。程序通过调用 `malloc` 函数来从堆中分配块。

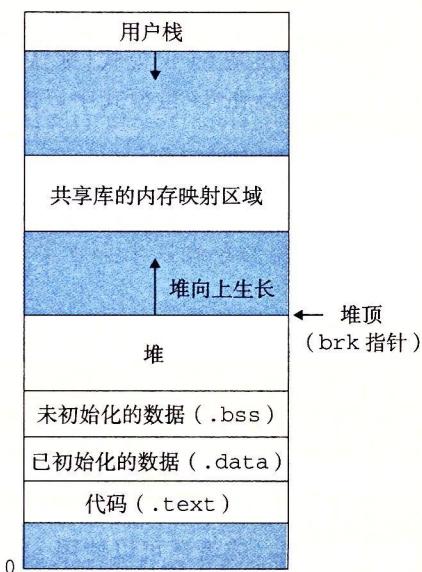


图 9-33 堆

```
#include <stdlib.h>
void *malloc(size_t size);
```

返回：若成功则为已分配块的指针，若出错则为 NULL。

malloc 函数返回一个指针，指向大小为至少 size 字节的内存块，这个块会为可能包含在这个块内的任何数据对象类型做对齐。实际中，对齐依赖于编译代码在 32 位模式（gcc -m32）还是 64 位模式（默认的）中运行。在 32 位模式中，malloc 返回的块的地址总是 8 的倍数。在 64 位模式中，该地址总是 16 的倍数。

旁注 一个字有多大

回想一下在第 3 章中我们对机器代码的讨论，Intel 将 4 字节对象称为双字。然而，在本节中，我们会假设字是 4 字节的对象，而双字是 8 字节的对象，这和传统术语是一致的。

如果 malloc 遇到问题（例如，程序要求的内存块比可用的虚拟内存还要大），那么它就返回 NULL，并设置 errno。malloc 不初始化它返回的内存。那些想要已初始化的动态内存的应用程序可以使用 calloc，calloc 是一个基于 malloc 的瘦包装函数，它将分配的内存初始化为零。想要改变一个以前已分配块的大小，可以使用 realloc 函数。

动态内存分配器，例如 malloc，可以通过使用 mmap 和 munmap 函数，显式地分配和释放堆内存，或者还可以使用 sbrk 函数：

```
#include <unistd.h>
void *sbrk(intptr_t incr);
```

返回：若成功则为旧的 brk 指针，若出错则为 -1。

sbrk 函数通过将内核的 brk 指针增加 incr 来扩展和收缩堆。如果成功，它就返回 brk 的旧值，否则，它就返回 -1，并将 errno 设置为 ENOMEM。如果 incr 为零，那么 sbrk 就返回 brk 的当前值。用一个为负的 incr 来调用 sbrk 是合法的，而且很巧妙，因为返回值（brk 的旧值）指向距新堆顶向上 abs(incr) 字节处。

程序是通过调用 free 函数来释放已分配的堆块。

```
#include <stdlib.h>
void free(void *ptr);
```

返回：无。

ptr 参数必须指向一个从 malloc、calloc 或者 realloc 获得的已分配块的起始位置。如果不是，那么 free 的行为就是未定义的。更糟的是，既然它什么都不返回，free 就不会告诉应用出现了错误。就像我们将在 9.11 节里看到的，这会产生一些令人迷惑的运行时错误。

图 9-34 展示了一个 malloc 和 free 的实现是如何管理一个 C 程序的 16 字的（非常）小的堆的。每个方框代表了一个 4 字节的字。粗线标出的矩形对应于已分配块（有阴影的）和空闲块（无阴影的）。初始时，堆是由一个大小为 16 个字的、双字对齐的、空闲块组成的。（本节中，我们假设分配器返回的块是 8 字节双字边界对齐的。）

- 图 9-34a: 程序请求一个 4 字的块。malloc 的响应是: 从空闲块的前部切出一个 4 字的块, 并返回一个指向这个块的第一字的指针。
- 图 9-34b: 程序请求一个 5 字的块。malloc 的响应是: 从空闲块的前部分配一个 6 字的块。在本例中, malloc 在块里填充了一个额外的字, 是为了保持空闲块是双字边界对齐的。
- 图 9-34c: 程序请求一个 6 字的块, 而 malloc 就从空闲块的前部切出一个 6 字的块。
- 图 9-34d: 程序释放在图 9-34b 中分配的那个 6 字的块。注意, 在调用 free 返回之后, 指针 p2 仍然指向被释放了的块。应用有责任在它被一个新的 malloc 调用重新初始化之前, 不再使用 p2。
- 图 9-34e: 程序请求一个 2 字的块。在这种情况下, malloc 分配在前一步中被释放了的块的一部分, 并返回一个指向这个新块的指针。

9.9.2 为什么要使用动态内存分配

程序使用动态内存分配的最重要的原因是经常直到程序实际运行时, 才知道某些数据结构的大小。例如, 假设要求我们编写一个 C 程序, 它读一个 n 个 ASCII 码整数的链表, 每一行一个整数, 从 `stdin` 到一个 C 数组。输入是由整数 n 和接下来要读和存储到数组中的 n 个整数组成的。最简单的方法就是静态地定义这个数组, 它的最大数组大小是硬编码的:

```

1 #include "csapp.h"
2 #define MAXN 15213
3
4 int array[MAXN];
5
6 int main()
7 {
8     int i, n;
9
10    scanf("%d", &n);
11    if (n > MAXN)
12        app_error("Input file too big");
13    for (i = 0; i < n; i++)
14        scanf("%d", &array[i]);
15    exit(0);
16 }
```

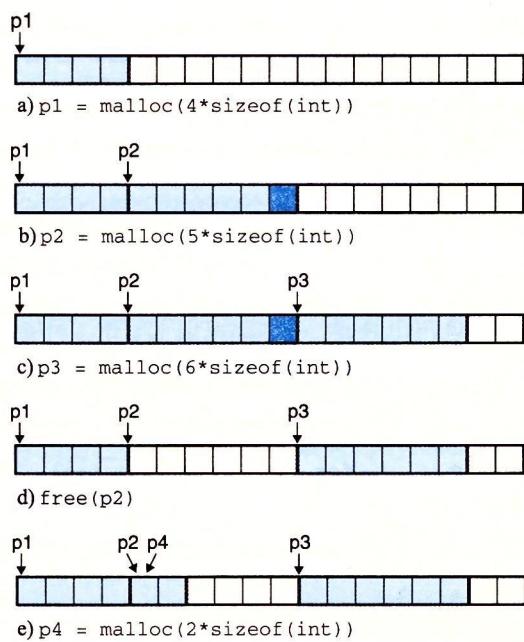


图 9-34 用 malloc 和 free 分配和释放块。每个方框对应于一个字。每个粗线标出的矩形对应于一个块。阴影部分是已分配的块。已分配的块的填充区域是深阴影的。无阴影部分是空闲块。堆地址是从左往右增加的

像这样用硬编码的大小来分配数组通常不是一种好想法。MAXN 的值是任意的，与机器上可用的虚拟内存的实际数量没有关系。而且，如果这个程序的使用者想读取一个比 MAXN 大的文件，唯一的办法就是用一个更大的 MAXN 值来重新编译这个程序。虽然对于这个简单的示例来说这不成问题，但是硬编码数组界限的出现对于拥有百万行代码和大量使用者的大型软件产品而言，会变成一场维护的噩梦。

一种更好的方法是在运行时，在已知了 n 的值之后，动态地分配这个数组。使用这种方法，数组大小的最大值就只由可用的虚拟内存数量来限制了。

```

1 #include "csapp.h"
2
3 int main()
4 {
5     int *array, i, n;
6
7     scanf("%d", &n);
8     array = (int *)Malloc(n * sizeof(int));
9     for (i = 0; i < n; i++)
10         scanf("%d", &array[i]);
11     free(array);
12     exit(0);
13 }
```

动态内存分配是一种有用而重要的编程技术。然而，为了正确而高效地使用分配器，程序员需要对它们是如何工作的有所了解。我们将在 9.11 节中讨论因为不正确地使用分配器所导致的一些可怕的错误。

9.9.3 分配器的要求和目标

显式分配器必须在一些相当严格的约束条件下工作：

- 处理任意请求序列。一个应用可以有任意的分配请求和释放请求序列，只要满足约束条件：每个释放请求必须对应于一个当前已分配块，这个块是由一个以前的分配请求获得的。因此，分配器不可以假设分配和释放请求的顺序。例如，分配器不能假设所有的分配请求都有相匹配的释放请求，或者有相匹配的分配和空闲请求是嵌套的。
- 立即响应请求。分配器必须立即响应分配请求。因此，不允许分配器为了提高性能重新排列或者缓冲请求。
- 只使用堆。为了使分配器是可扩展的，分配器使用的任何非标量数据结构都必须保存在堆里。
- 对齐块(对齐要求)。分配器必须对齐块，使得它们可以保存任何类型的数据对象。
- 不修改已分配的块。分配器只能操作或者改变空闲块。特别是，一旦块被分配了，就不允许修改或者移动它了。因此，诸如压缩已分配块这样的技术是不允许使用的。

在这些限制条件下，分配器的编写者试图实现吞吐率最大化和内存使用率最大化，而这两个性能目标通常是相互冲突的。

- 目标 1：最大化吞吐率。假定 n 个分配和释放请求的某种序列：

$$R_0, R_1, \dots, R_k, \dots, R_{n-1}$$

我们希望一个分配器的吞吐率最大化，吞吐率定义为每个单位时间里完成的请求数。例如，如果一个分配器在1秒内完成500个分配请求和500个释放请求，那么它的吞吐率就是每秒1000次操作。一般而言，我们可以通过使满足分配和释放请求的平均时间最小化来使吞吐率最大化。正如我们会看到的，开发一个具有合理性能的分配器并不困难，所谓合理性能是指一个分配请求的最糟运行时间与空闲块的数量成线性关系，而一个释放请求的运行时间是个常数。

- 目标2：最大化内存利用率。天真的程序员经常不正确地假设虚拟内存是一个无限的资源。实际上，一个系统中被所有进程分配的虚拟内存的全部数量是受磁盘上交换空间的数量限制的。好的程序员知道虚拟内存是一个有限的空间，必须高效地使用。对于可能被要求分配和释放大块内存的动态内存分配器来说，尤其如此。

有很多方式来描述一个分配器使用堆的效率如何。在我们的经验中，最有用的标准是峰值利用率(peak utilization)。像以前一样，我们给定 n 个分配和释放请求的某种顺序

$$R_0, R_1, \dots, R_k, \dots, R_{n-1}$$

如果一个应用程序请求一个 p 字节的块，那么得到的已分配块的有效载荷(payload)是 p 字节。在请求 R_k 完成之后，聚集有效载荷(aggregate payload)表示为 P_k ，为当前已分配的块的有效载荷之和，而 H_k 表示堆的当前的(单调非递减的)大小。

那么，前 $k+1$ 个请求的峰值利用率，表示为 U_k ，可以通过下式得到：

$$U_k = \frac{\max_{i \leq k} P_i}{H_k}$$

那么，分配器的目标就是在整个序列中使峰值利用率 U_{n-1} 最大化。正如我们将要看到的，在最大化吞吐率和最大化利用率之间是互相牵制的。特别是，以堆利用率为代价，很容易编写出吞吐率最大化的分配器。分配器设计中一个有趣的挑战就是在两个目标之间找到一个适当的平衡。

旁注 放宽单调性假设

我们可以通过让 H_k 成为前 $k+1$ 个请求的最高峰，从而使得在我们对 U_k 的定义中放宽单调非递减的假设，并且允许堆增长和降低。

9.9.4 碎片

造成堆利用率很低的主要原因是一种称为碎片(fragmentation)的现象，当虽然有未使用的内存但不能用来满足分配请求时，就发生这种现象。有两种形式的碎片：内部碎片(internal fragmentation)和外部碎片(external fragmentation)。

内部碎片是在一个已分配块比有效载荷大时发生的。很多原因都可能造成这个问题。例如，一个分配器的实现可能对已分配块强加一个最小的大小值，而这个大小要比某个请求的有效载荷大。或者，就如我们在图9-34b中看到的，分配器可能增加块大小以满足对齐约束条件。

内部碎片的量化是简单明了的。它就是已分配块大小和它们的有效载荷大小之差的和。因此，在任意时刻，内部碎片的数量只取决于以前请求的模式和分配器的实现方式。

外部碎片是当空闲内存合计起来足够满足一个分配请求，但是没有一个单独的空闲块足够大可以来处理这个请求时发生的。例如，如果图9-34e中的请求要求6个字，而不是2个字，那么如果不向内核请求额外的虚拟内存就无法满足这个请求，即使在堆中仍然有

6个空闲的字。问题的产生是由于这6个字是分在两个空闲块中的。

外部碎片比内部碎片的量化要困难得多，因为它不仅取决于以前请求的模式和分配器的实现方式，还取决于将来请求的模式。例如，假设在 k 个请求之后，所有空闲块的大小都恰好是4个字。这个堆会有外部碎片吗？答案取决于将来请求的模式。如果将来所有的分配请求都要求小于或者等于4个字的块，那么就不会有外部碎片。另一方面，如果有一个或者多个请求要求比4个字大的块，那么这个堆就会有外部碎片。

因为外部碎片难以量化且不可能预测，所以分配器通常采用启发式策略来试图维持少量的大空闲块，而不是维持大量的小空闲块。

9.9.5 实现问题

可以想象出的最简单的分配器会把堆组织成一个大的字节数组，还有一个指针 p ，初始指向这个数组的第一个字节。为了分配 $size$ 个字节，`malloc`将 p 的当前值保存在栈里，将 p 增加 $size$ ，并将 p 的旧值返回到调用函数。`free`只是简单地返回到调用函数，而不做其他任何事情。

这个简单的分配器是设计中的一种极端情况。因为每个`malloc`和`free`只执行很少量的指令，吞吐率会极好。然而，因为分配器从不重复使用任何块，内存利用率将极差。一个实际的分配器要在吞吐率和利用率之间把握好平衡，就必须考虑以下几个问题：

- 空闲块组织：我们如何记录空闲块？
- 放置：我们如何选择一个合适的空闲块来放置一个新分配的块？
- 分割：在将一个新分配的块放置到某个空闲块之后，我们如何处理这个空闲块中的剩余部分？
- 合并：我们如何处理一个刚刚被释放的块？

本节剩下的部分将更详细地讨论这些问题。因为像放置、分割以及合并这样的基本技术贯穿在许多不同的空闲块组织中，所以我们将在一种叫做隐式空闲链表的简单空闲块组织结构中来介绍它们。

9.9.6 隐式空闲链表

任何实际的分配器都需要一些数据结构，允许它来区别块边界，以及区别已分配块和空闲块。大多数分配器将这些信息嵌入块本身。一个简单的方法如图9-35所示。

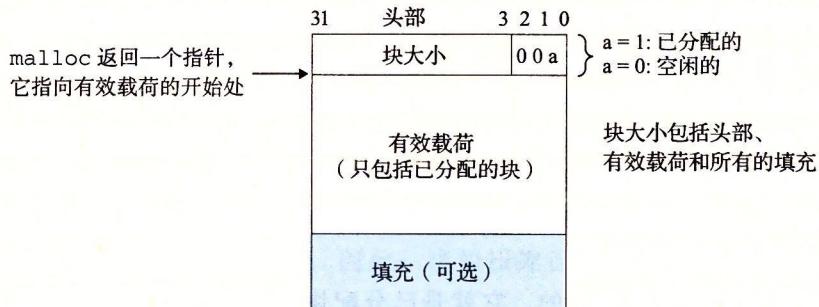


图 9-35 一个简单的堆块的格式

在这种情况下，一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小(包括头部和所有的填充)，以及这个块是已分配的还是空

闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是零。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况下，我们用其中的最低位(已分配位)来指明这个块是已分配的还是空闲的。例如，假设我们有一个已分配的块，大小为 24(0x18)字节。那么它的头部将是

$$0x00000018 \mid 0x1 = 0x00000019$$

类似地，一个块大小为 40(0x28)字节的空闲块有如下的头部：

$$0x00000028 \mid 0x0 = 0x00000028$$

头部后面就是应用调用 `malloc` 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

假设块的格式如图 9-35 所示，我们可以将堆组织为一个连续的已分配块和空闲块的序列，如图 9-36 所示。

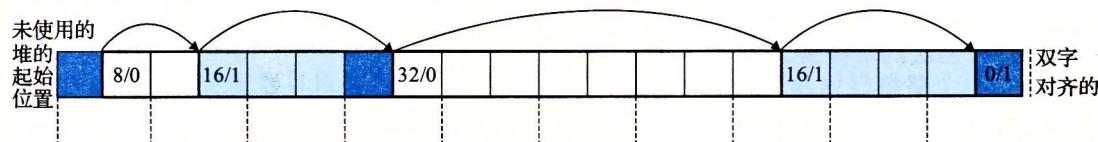


图 9-36 用隐式空闲链表来组织堆。阴影部分是已分配块。没有阴影的部分是空闲块。
头部标记为(大小(字节)/已分配位)

我们称这种结构为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。注意，我们需要某种特殊标记的结束块，在这个示例中，就是一个设置了已分配位而大小为零的终止头部(terminating header)。(就像我们将在 9.9.12 节中看到的，设置已分配位简化了空闲块的合并。)

隐式空闲链表的优点是简单。显著的缺点是任何操作的开销，例如放置分配的块，要求对空闲链表进行搜索，该搜索所需时间与堆中已分配块和空闲块的总数呈线性关系。

很重要的一点就是意识到系统对齐要求和分配器对块格式的选择会对分配器上的最小块大小有强制的要求。没有已分配块或者空闲块可以比这个最小值还小。例如，如果我们假设一个双字的对齐要求，那么每个块的大小都必须是双字(8 字节)的倍数。因此，图 9-35 中的块格式就导致最小的块大小为两个字：一个字作头，另一个字维持对齐要求。即使应用只请求一字节，分配器也仍然需要创建一个两字的块。

 **练习题 9.6** 确定下面 `malloc` 请求序列产生的块大小和头部值。假设：1) 分配器保持双字对齐，并且使用块格式如图 9-35 中所示的隐式空闲链表。2) 块大小向上舍入为最接近的 8 字节的倍数。

请求	块大小(十进制字节)	块头部(十六进制)
<code>malloc(1)</code>		
<code>malloc(5)</code>		
<code>malloc(12)</code>		
<code>malloc(13)</code>		

9.9.7 放置已分配的块

当一个应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大可以放置

所请求块的空闲块。分配器执行这种搜索的方式是由放置策略(placement policy)确定的。一些常见的策略是首次适配(first fit)、下一次适配(next fit)和最佳适配(best fit)。

首次适配从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配和首次适配很相似，只不过不是从链表的起始处开始每次搜索，而是从上一次查询结束的地方开始。最佳适配检查每个空闲块，选择适合所需请求大小的最小空闲块。

首次适配的优点是它趋向于将大的空闲块保留在链表的后面。缺点是它趋向于在靠近链表起始处留下小空闲块的“碎片”，这就增加了对较大块的搜索时间。下一次适配是由Donald Knuth作为首次适配的一种代替品最早提出的，源于这样一个想法：如果我们上一次在某个空闲块里已经发现了一个匹配，那么很可能下一次我们也能在这个剩余块中发现匹配。下一次适配比首次适配运行起来明显要快一些，尤其是当链表的前面布满了许多小的碎片时。然而，一些研究表明，下一次适配的内存利用率要比首次适配低得多。研究还表明最佳适配比首次适配和下一次适配的内存利用率都要高一些。然而，在简单空闲链表组织结构中，比如隐式空闲链表中，使用最佳适配的缺点是它要求对堆进行彻底的搜索。在后面，我们将看到更加精细复杂的分离式空闲链表组织，它接近于最佳适配策略，不需要进行彻底的堆搜索。

9.9.8 分割空闲块

一旦分配器找到一个匹配的空闲块，它就必须做另一个策略决定，那就是分配这个空闲块中多少空间。一个选择是用整个空闲块。虽然这种方式简单而快捷，但是主要的缺点就是它会造成内部碎片。如果放置策略趋向于产生好的匹配，那么额外的内部碎片也是可以接受的。

然而，如果匹配不太好，那么分配器通常会选择将这个空闲块分割为两部分。第一部分变成分配块，而剩下的变成一个新的空闲块。图 9-37 展示了分配器如何分割图 9-36 中 8 个字的空闲块，来满足一个应用的对堆内存 3 个字的请求。

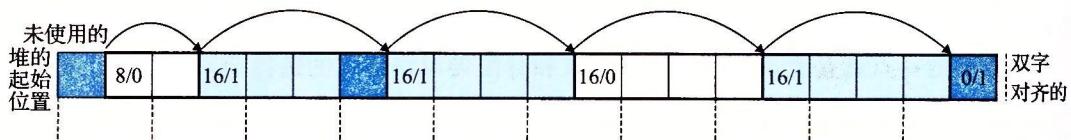


图 9-37 分割一个空闲块，以满足一个 3 个字的分配请求。阴影部分是已分配块。

没有阴影的部分是空闲块。头部标记为(大小(字节)/已分配位)

9.9.9 获取额外的堆内存

如果分配器不能为请求块找到合适的空闲块将发生什么呢？一个选择是通过合并那些在内存中物理上相邻的空闲块来创建一些更大的空闲块(在下一节中描述)。然而，如果这样还是不能生成一个足够大的块，或者如果空闲块已经最大程度地合并了，那么分配器就会通过调用 sbrk 函数，向内核请求额外的堆内存。分配器将额外的内存转化成一个大的空闲块，将这个块插入到空闲链表中，然后将被请求的块放置在这个新的空闲块中。

9.9.10 合并空闲块

当分配器释放一个已分配块时，可能有其他空闲块与这个新释放的空闲块相邻。这些邻接的空闲块可能引起一种现象，叫做假碎片(fault fragmentation)，就是有许多可用的

空闲块被切割成为小的、无法使用的空闲块。比如，图 9-38 展示了释放图 9-37 中分配的块后得到的结果。结果是两个相邻的空闲块，每一个的有效载荷都为 3 个字。因此，接下来一个对 4 字有效载荷的请求就会失败，即使两个空闲块的合计大小足够大，可以满足这个请求。

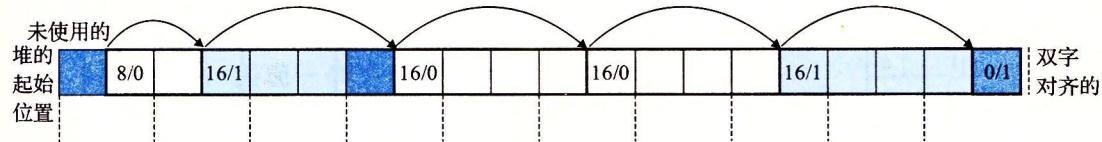


图 9-38 假碎片的示例。阴影部分是已分配块。没有阴影的部分是空闲块。
头部标记为(大小(字节)/已分配位)

为了解决假碎片问题，任何实际的分配器都必须合并相邻的空闲块，这个过程称为合并(coalescing)。这就出现了一个重要的策略决定，那就是何时执行合并。分配器可以选择立即合并(immediate coalescing)，也就是在每次一个块被释放时，就合并所有的相邻块。或者它也可以选择推迟合并(deferred coalescing)，也就是等到某个稍晚的时候再合并空闲块。例如，分配器可以推迟合并，直到某个分配请求失败，然后扫描整个堆，合并所有的空闲块。

立即合并很简单明了，可以在常数时间内执行完成，但是对于某些请求模式，这种方式会产生一种形式的抖动，块会反复地合并，然后马上分割。例如，在图 9-38 中，反复地分配和释放一个 3 个字的块将产生大量不必要的分割和合并。在对分配器的讨论中，我们会假设使用立即合并，但是你应该了解，快速的分配器通常会选择某种形式的推迟合并。

9.9.11 带边界标记的合并

分配器是如何实现合并的？让我们称想要释放的块为当前块。那么，合并(内存中的)下一个空闲块很简单而且高效。当前块的头部指向下一个块的头部，可以检查这个指针以判断下一个块是否是空闲的。如果是，就将它的大小简单地加到当前块头部的大小上，这两个块在常数时间内被合并。

但是我们该如何合并前面的块呢？给定一个带头部的隐式空闲链表，唯一的选择将是搜索整个链表，记住前面块的位置，直到我们到达当前块。使用隐式空闲链表，这意味着每次调用 free 需要的时间都与堆的大小成线性关系。即使使用更复杂精细的空闲链表组织，搜索时间也不会是常数。

Knuth 提出了一种聪明而通用的技术，叫做边界标记(boundary tag)，允许在常数时间内进行对前面块的合并。这种思想，如图 9-39 所示，是在每个块的结尾处添加一个脚部(footer, 边界标记)，其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前一个块的起始位置和状态，这个脚部总是在距当前块开始位置一个字的距离。

考虑当分配器释放当前块时所有可能存在的情况：

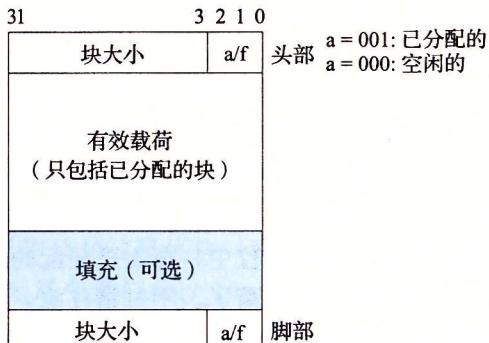


图 9-39 使用边界标记的堆块的格式

- 1) 前面的块和后面的块都是已分配的。
- 2) 前面的块是已分配的，后面的块是空闲的。
- 3) 前面的块是空闲的，而后面的块是已分配的。
- 4) 前面的和后面的块都是空闲的。

图 9-40 展示了我们如何对这四种情况进行合并。

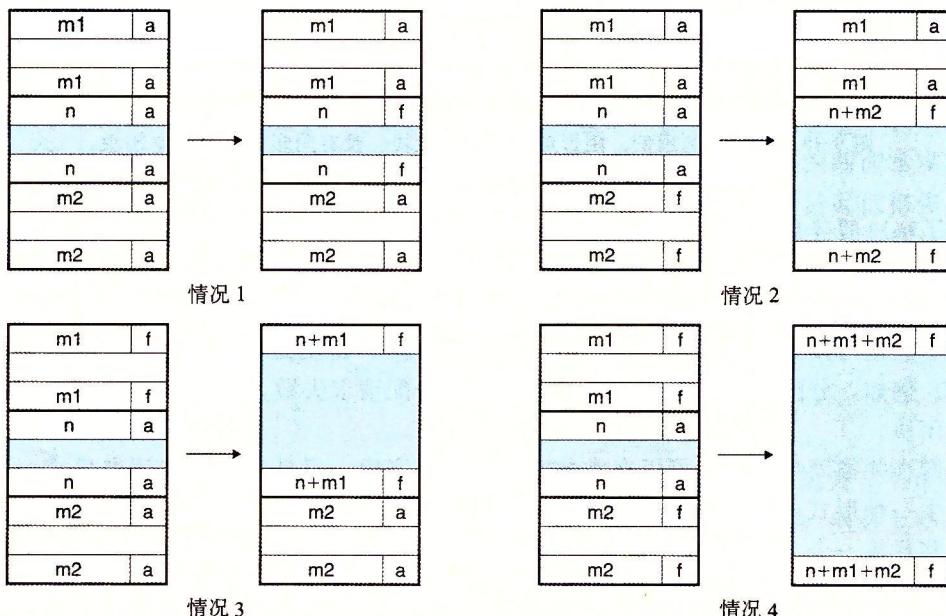


图 9-40 使用边界标记的合并(情况 1: 前面的和后面块都已分配。情况 2: 前面块已分配, 后面块空闲。情况 3: 前面块空闲, 后面块已分配。情况 4: 后面块和前面块都空闲)

在情况 1 中, 两个邻接的块都是已分配的, 因此不可能进行合并。所以当前块的状态只是简单地从已分配变成空闲。在情况 2 中, 当前块与后面的块合并。用当前块和后面块的大小的和来更新当前块的头部和后面块的脚部。在情况 3 中, 前面的块和当前块合并。用两个块大小的和来更新前面块的头部和当前块的脚部。在情况 4 中, 要合并所有的三个块形成一个单独的空闲块, 用三个块大小的和来更新前面块的头部和后面块的脚部。在每种情况下, 合并都是在常数时间内完成的。

边界标记的概念是简单优雅的, 它对许多不同类型的分配器和空闲链表组织都是通用的。然而, 它也存在一个潜在的缺陷。它要求每个块都保持一个头部和一个脚部, 在应用程序操作许多个小块时, 会产生显著的内存开销。例如, 如果一个图形应用通过反复调用 `malloc` 和 `free` 来动态地创建和销毁图形节点, 并且每个图形节点都只要求两个内存字, 那么头部和脚部将占用每个已分配块的一半的空间。

幸运的是, 有一种非常聪明的边界标记的优化方法, 能够使得在已分配块中不再需要脚部。回想一下, 当我们试图在内存中合并当前块以及前面的块和后面的块时, 只有在前面的块是空闲时, 才会需要用到它的脚部。如果我们把前面块的已分配/空闲位存放在当前块中多出来的低位中, 那么已分配的块就不需要脚部了, 这样我们就可以将这个多出来的空间用作有效载荷了。不过请注意, 空闲块仍然需要脚部。

 **练习题 9.7** 确定下面每种对齐要求和块格式的组合的最小的块大小。假设: 隐式空闲链表, 不允许有效载荷为零, 头部和脚部存放在 4 字节的字中。

对齐要求	已分配的块	空闲块	最小块大小(字节)
单字	头部和脚部	头部和脚部	
单字	头部,但是无脚部	头部和脚部	
双字	头部和脚部	头部和脚部	
双字	头部,但是没有脚部	头部和脚部	

9.9.12 综合：实现一个简单的分配器

构造一个分配器是一件富有挑战性的任务。设计空间很大，有多种块格式、空闲链表格式，以及放置、分割和合并策略可供选择。另一个挑战就是你经常被迫在类型系统的安全和熟悉的限定之外编程，依赖于容易出错的指针强制类型转换和指针运算，这些操作都属于典型的低层系统编程。

虽然分配器不需要大量的代码，但是它们也还是细微而不可忽视的。熟悉诸如 C++ 或者 Java 之类高级语言的学生通常在他们第一次遇到这种类型的编程时，会遭遇一个概念上的障碍。为了帮助你清除这个障碍，我们将基于隐式空闲链表，使用立即边界标记合并方式，从头至尾地讲述一个简单分配器的实现。最大的块大小为 $2^{32} = 4\text{GB}$ 。代码是 64 位干净的，即代码能不加修改地运行在 32 位(gcc -m32)或 64 位(gcc -m64)的进程中。

1. 通用分配器设计

我们的分配器使用如图 9-41 所示的 memlib.c 包所提供的一个内存系统模型。模型的目的在于允许我们在不干涉已存在的系统层 malloc 包的情况下，运行分配器。

mem_init 函数将对于堆来说可用的虚拟内存模型化为一个大的、双字对齐的字节数组。在 mem_heap 和 mem_brk 之间的字节表示已分配的虚拟内存。mem_brk 之后的字节表示未分配的虚拟内存。分配器通过调用 mem_sbrk 函数来请求额外的堆内存，这个函数和系统的 sbrk 函数的接口相同，而且语义也相同，除了它会拒绝收缩堆的请求。

- 分配器包含在一个源文件中(mm.c)，用户可以编译和链接这个源文件到他们的应用之中。分配器输出三个函数到应用程序：

```

1  extern int mm_init(void);
2  extern void *mm_malloc (size_t size);
3  extern void mm_free (void *ptr);

```

mm_init 函数初始化分配器，如果成功就返回 0，否则就返回 -1。mm_malloc 和 mm_free 函数与它们对应的系统函数有相同的接口和语义。分配器使用如图 9-39 所示的块格式。最小块的大小为 16 字节。空闲链表组织成为一个隐式空闲链表，具有如图 9-42 所示的恒定形式。

第一个字是一个双字边界对齐的不使用的填充字。填充后面紧跟着一个特殊的序言块(prologue block)，这是一个 8 字节的已分配块，只由一个头部和一个脚部组成。序言块是在初始化时创建的，并且永不释放。在序言块后紧跟的是零个或者多个由 malloc 或者 free 调用创建的普通块。堆总是以一个特殊的结尾块(epilogue block)来结束，这个块是一个大小为零的已分配块，只由一个头部组成。序言块和结尾块是一种消除合并时边界条件的技巧。分配器使用一个单独的私有(static)全局变量(heap_listp)，它总是指向序言块。(作为一个小优化，我们可以让它指向下一个块，而不是这个序言块。)

code/vm/malloc/memlib.c

```

1  /* Private global variables */
2  static char *mem_heap;      /* Points to first byte of heap */
3  static char *mem_brk;       /* Points to last byte of heap plus 1 */
4  static char *mem_max_addr;  /* Max legal heap addr plus 1*/
5
6  /*
7   * mem_init - Initialize the memory system model
8   */
9  void mem_init(void)
10 {
11     mem_heap = (char *)Malloc(MAX_HEAP);
12     mem_brk = (char *)mem_heap;
13     mem_max_addr = (char *)(mem_heap + MAX_HEAP);
14 }
15
16 /*
17  * mem_sbrk - Simple model of the sbrk function. Extends the heap
18  * by incr bytes and returns the start address of the new area. In
19  * this model, the heap cannot be shrunk.
20  */
21 void *mem_sbrk(int incr)
22 {
23     char *old_brk = mem_brk;
24
25     if ((incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
26         errno = ENOMEM;
27         fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
28         return (void *)-1;
29     }
30     mem_brk += incr;
31     return (void *)old_brk;
32 }
```

code/vm/malloc/memlib.c

图 9-41 memlib.c: 内存系统模型

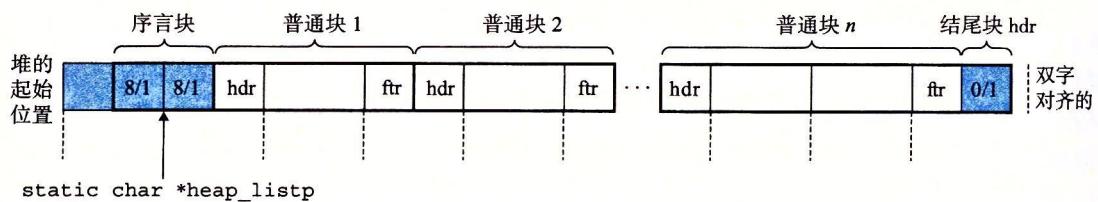


图 9-42 隐式空闲链表的恒定形式

2. 操作空闲链表的基本常数和宏

图 9-43 展示了一些我们在分配器编码中将要使用的基本常数和宏。第 2~4 行定义了一些基本的大小常数：字的大小(WSIZE)和双字的大小(DSIZE)，初始空闲块的大小和扩展堆时的默认大小(CHUNKSIZE)。

在空闲链表中操作头部和脚部可能是很麻烦的，因为它要求大量使用强制类型转换和指针运算。因此，我们发现定义一小组宏来访问和遍历空闲链表是很有帮助的(第 9~25 行)。PACK

宏(第9行)将大小和已分配位结合起来并返回一个值，可以把它存放在头部或者脚部中。

code/vm/malloc/mm.c

```

1  /* Basic constants and macros */
2  #define WSIZE      4      /* Word and header/footer size (bytes) */
3  #define DSIZE      8      /* Double word size (bytes) */
4  #define CHUNKSIZE  (1<<12) /* Extend heap by this amount (bytes) */
5
6  #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8  /* Pack a size and allocated bit into a word */
9  #define PACK(size, alloc) ((size) | (alloc))
10
11 /* Read and write a word at address p */
12 #define GET(p)      (*(unsigned int *) (p))
13 #define PUT(p, val) (*(unsigned int *) (p) = (val))
14
15 /* Read the size and allocated fields from address p */
16 #define GET_SIZE(p) (GET(p) & ~0x7)
17 #define GET_ALLOC(p) (GET(p) & 0x1)
18
19 /* Given block ptr bp, compute address of its header and footer */
20 #define HDRP(bp)    ((char *) (bp) - WSIZE)
21 #define FTRP(bp)    ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23 /* Given block ptr bp, compute address of next and previous blocks */
24 #define NEXT_BLKP(bp) ((char *) (bp) + GET_SIZE(((char *) (bp) - WSIZE)))
25 #define PREV_BLKP(bp) ((char *) (bp) - GET_SIZE(((char *) (bp) - DSIZE)))

```

code/vm/malloc/mm.c

图 9-43 操作空闲链表的基本常数和宏

GET 宏(第 12 行)读取和返回参数 p 引用的字。这里强制类型转换是至关重要的。参数 p 典型地是一个(viod*)指针，不可以直接进行间接引用。类似地，PUT 宏(第 13 行)将 val 存放在参数 p 指向的字中。

GET_SIZE 和 GET_ALLOC 宏(第 16~17 行)从地址 p 处的头部或者脚部分别返回大小和已分配位。剩下的宏是对块指针(block pointer, 用 bp 表示)的操作，块指针指向第一个有效载荷字节。给定一个块指针 bp，HDRP 和 FTRP 宏(第 20~21 行)分别返回指向这个块的头部和脚部的指针。NEXT_BLKP 和 PREV_BLKP 宏(第 24~25 行)分别返回指向后面的块和前面的块的块指针。

可以用多种方式来编辑宏，以操作空闲链表。比如，给定一个指向当前块的指针 bp，我们可以使用下面的代码行来确定内存中后面的块的大小：

```
size_t size = GET_SIZE(HDRP(NEXT_BLKP(bp)));
```

3. 创建初始空闲链表

在调用 mm_malloc 或者 mm_free 之前，应用必须通过调用 mm_init 函数来初始化堆(见图 9-44)。

mm_init 函数从内存系统得到 4 个字，并将它们初始化，创建一个空的空闲链表(第 4~10 行)。然后它调用 extend_heap 函数(图 9-45)，这个函数将堆扩展 CHUNKSIZE 字

节，并且创建初始的空闲块。此刻，分配器已初始化了，并且准备好接受来自应用的分配和释放请求。

```
1 int mm_init(void)
2 {
3     /* Create the initial empty heap */
4     if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
5         return -1;
6     PUT(heap_listp, 0);                                /* Alignment padding */
7     PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
8     PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
9     PUT(heap_listp + (3*WSIZE), PACK(0, 1));      /* Epilogue header */
10    heap_listp += (2*WSIZE);
11
12    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14        return -1;
15    return 0;
16 }
```

code/vm/malloc/mm.c

图 9-44 mm_init: 创建带一个初始空闲块的堆

```
1 static void *extend_heap(size_t words)
2 {
3     char *bp;
4     size_t size;
5
6     /* Allocate an even number of words to maintain alignment */
7     size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8     if ((long)(bp = mem_sbrk(size)) == -1)
9         return NULL;
10
11    /* Initialize free block header/footer and the epilogue header */
12    PUT(HDRP(bp), PACK(size, 0));           /* Free block header */
13    PUT(FTRP(bp), PACK(size, 0));           /* Free block footer */
14    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
15
16    /* Coalesce if the previous block was free */
17    return coalesce(bp);
18 }
```

code/vm/malloc/mm.c

图 9-45 extend_heap: 用一个新的空闲块扩展堆

`extend_heap` 函数会在两种不同的环境中被调用：1)当堆被初始化时；2)当 `mm_malloc` 不能找到一个合适的匹配块时。为了保持对齐，`extend_heap` 将请求大小向上舍入为最接近的 2 字(8 字节)的倍数，然后向内存系统请求额外的堆空间(第 7~9 行)。

`extend_heap` 函数的剩余部分(第 12~17 行)有点儿微妙。堆开始于一个双字对齐的边界，并且每次对 `extend_heap` 的调用都返回一个块，该块的大小是双字的整数倍。因此，对 `mem_sbrk` 的每次调用都返回一个双字对齐的内存片，紧跟在结尾块的头部后面。这个头部变成了新的空闲块的头部(第 12 行)，并且这个片的最后一个字变成了新的结尾

块的头部(第 14 行)。最后，在很可能出现的前一个堆以一个空闲块结束的情况下，我们调用 coalesce 函数来合并两个空闲块，并返回指向合并后的块的块指针(第 17 行)。

4. 释放和合并块

应用通过调用 mm_free 函数(图 9-46)，来释放一个以前分配的块，这个函数释放所请求的块(bp)，然后使用 9.9.11 节中描述的边界标记合并技术将之与邻接的空闲块合并起来。

code/vm/malloc/mm.c

```

1 void mm_free(void *bp)
2 {
3     size_t size = GET_SIZE(HDRP(bp));
4
5     PUT(HDRP(bp), PACK(size, 0));
6     PUT(FTRP(bp), PACK(size, 0));
7     coalesce(bp);
8 }
9
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15
16     if (prev_alloc && next_alloc) {           /* Case 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) {      /* Case 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22         PUT(HDRP(bp), PACK(size, 0));
23         PUT(FTRP(bp), PACK(size, 0));
24     }
25
26     else if (!prev_alloc && next_alloc) {      /* Case 3 */
27         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
28         PUT(FTRP(bp), PACK(size, 0));
29         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
30         bp = PREV_BLKP(bp);
31     }
32
33     else {                                     /* Case 4 */
34         size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
35             GET_SIZE(FTRP(NEXT_BLKP(bp)));
36         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
37         PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
38         bp = PREV_BLKP(bp);
39     }
40
41 }

```

code/vm/malloc/mm.c

图 9-46 mm_free：释放一个块，并使用边界标记合并将之与所有的邻接空闲块在常数时间内合并

coalesce 函数中的代码是图 9-40 中勾画的四种情况的一种简单直接的实现。这里也有一个微妙的方面。我们选择的空闲链表格式(它的序言块和结尾块总是标记为已分配)允许我们忽略潜在的麻烦边界情况，也就是，请求块 bp 在堆的起始处或者是在堆的结尾处。如果没有这些特殊块，代码将混乱得多，更加容易出错，并且更慢，因为我们将不得不在每次释放请求时，都去检查这些并不常见的边界情况。

5. 分配块

一个应用通过调用 mm_malloc 函数(见图 9-47)来向内存请求大小为 size 字节的块。在检查完请求的真假之后，分配器必须调整请求块的大小，从而为头部和脚部留有空间，并满足双字对齐的要求。第 12~13 行强制了最小块大小是 16 字节：8 字节用来满足对齐要求，而另外 8 个用来放头部和脚部。对于超过 8 字节的请求(第 15 行)，一般的规则是加上开销字节，然后向上舍入到最接近的 8 的整数倍。

code/vm/malloc/mm.c

```

1 void *mm_malloc(size_t size)
2 {
3     size_t asize;      /* Adjusted block size */
4     size_t extendsize; /* Amount to extend heap if no fit */
5     char *bp;
6
7     /* Ignore spurious requests */
8     if (size == 0)
9         return NULL;
10
11    /* Adjust block size to include overhead and alignment reqs. */
12    if (size <= DSIZE)
13        asize = 2*DSIZE;
14    else
15        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
16
17    /* Search the free list for a fit */
18    if ((bp = find_fit(asize)) != NULL) {
19        place(bp, asize);
20        return bp;
21    }
22
23    /* No fit found. Get more memory and place the block */
24    extendsize = MAX(asize,CHUNKSIZE);
25    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26        return NULL;
27    place(bp, asize);
28    return bp;
29 }
```

code/vm/malloc/mm.c

图 9-47 mm_malloc: 从空闲链表分配一个块

一旦分配器调整了请求的大小，它就会搜索空闲链表，寻找一个合适的空闲块(第 18 行)。如果有合适的，那么分配器就放置这个请求块，并可选地分割出多余的部分(第 19 行)，然后返回新分配块的地址。

如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆(第 24~26 行)，把请求块放置在这个新的空闲块里，可选地分割这个块(第 27 行)，然后返回一个指针，指向这个新分配的块。

 练习题 9.8 为 9.9.12 节中描述的简单分配器实现一个 `find_fit` 函数。

```
static void *find_fit(size_t asize)
```

你的解答应该对隐式空闲链表执行首次适配搜索。

 练习题 9.9 为示例的分配器编写一个 `place` 函数。

```
static void place(void *bp, size_t asize)
```

你的解答应该将请求块放置在空闲块的起始位置，只有当剩余部分的大小等于或者超出最小块的大小时，才进行分割。

9.9.13 显式空闲链表

隐式空闲链表为我们提供了一种介绍一些基本分配器概念的简单方法。然而，因为块分配与堆块的总数呈线性关系，所以对于通用的分配器，隐式空闲链表是不适合的(尽管对于堆块数量预先就知道是很小的特殊的分配器来说它是可以的)。

一种更好的方法是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 `pred`(前驱)和 `succ`(后继)指针，如图 9-48 所示。

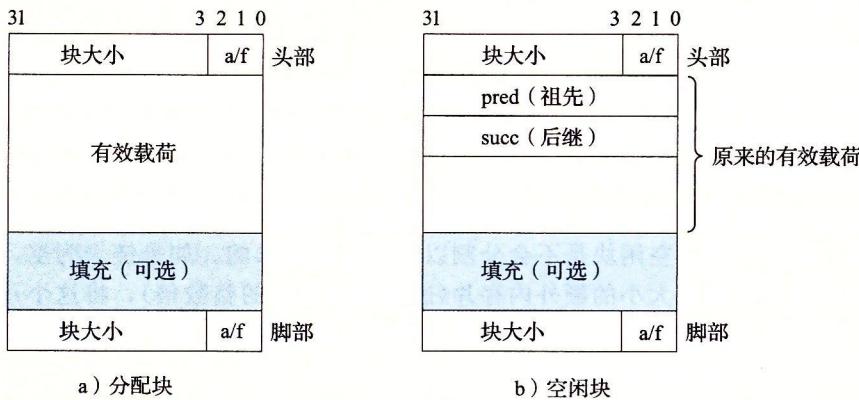


图 9-48 使用双向空闲链表的堆块的格式

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以是线性的，也可能是个常数，这取决于我们所选择的空闲链表中块的排序策略。

一种方法是用后进先出(LIFO)的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按

照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

9.9.14 分离的空闲链表

就像我们已经看到的，一个使用单向空闲块链表的分配器需要与空闲块数量呈线性关系的时间来分配块。一种流行的减少分配时间的方法，通常称为分离存储 (segregated storage)，就是维护多个空闲链表，其中每个链表中的块有大致相等的大小。一般的思路是将所有可能的块大小分成一些等价类，也叫做大小类 (size class)。有很多种方式来定义大小类。例如，我们可以根据 2 的幂来划分块大小：

$\{1\}, \{2\}, \{3, 4\}, \{5 \sim 8\}, \dots, \{1025 \sim 2048\}, \{2049 \sim 4096\}, \{4097 \sim \infty\}$

或者我们可以将小的块分派到它们自己的大小类里，而将大块按照 2 的幂分类：

$\{1\}, \{2\}, \{3\}, \dots, \{1023\}, \{1024\}, \{1025 \sim 2048\}, \{2049 \sim 4096\}, \{4097 \sim \infty\}$

分配器维护着一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序排列。当分配器需要一个大小为 n 的块时，它就搜索相应的空闲链表。如果不能找到合适的块与之匹配，它就搜索下一个链表，以此类推。

有关动态内存分配的文献描述了几十种分离存储方法，主要的区别在于它们如何定义大小类，何时进行合并，何时向操作系统请求额外的堆内存，是否允许分割，等等。为了使你大致了解有哪些可能性，我们会描述两种基本的方法：简单分离存储 (simple segregated storage) 和分离适配 (segregated fit)。

1. 简单分离存储

使用简单分离存储，每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小。例如，如果某个大小类定义为 $\{17 \sim 32\}$ ，那么这个类的空闲链表全由大小为 32 的块组成。

为了分配一个给定大小的块，我们检查相应的空闲链表。如果链表非空，我们简单地分配其中第一块的全部。空闲块是不会分割以满足分配请求的。如果链表为空，分配器就向操作系统请求一个固定大小的额外内存片（通常是页大小的整数倍），将这个片分成大小相等的块，并将这些块链接起来形成新的空闲链表。要释放一个块，分配器只要简单地将这个块插入到相应的空闲链表的前部。

这种简单的方法有许多优点。分配和释放块都是很快的常数时间操作。而且，每个片中都是大小相等的块，不分割，不合并，这意味着每个块只有很少的内存开销。由于每个片只有大小相同的块，那么一个已分配块的大小就可以从它的地址中推断出来。因为没有合并，所以已分配块的头部就不需要一个已分配/空闲标记。因此已分配块不需要头部，同时因为没有合并，它们也不需要脚部。因为分配和释放操作都是在空闲链表的起始处操作，所以链表只需要是单向的，而不用是双向的。关键点在于，在任何块中都需要的唯一字段是每个空闲块中的一个字的 succ 指针，因此最小块大小就是一个字。

一个显著的缺点是，简单分离存储很容易造成内部和外部碎片。因为空闲块是不会被分割的，所以可能会造成内部碎片。更糟的是，因为不会合并空闲块，所以某些引用模式会引起极多的外部碎片（见练习题 9.10）。

 练习题 9.10 描述一个在基于简单分离存储的分配器中会导致严重外部碎片的引用模式。

2. 分离适配

使用这种方法，分配器维护着一个空闲链表的数组。每个空闲链表是和一个大小类相关的，并且被组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块，这些块的大小是大小类的成员。有许多种不同的分离适配分配器。这里，我们描述了一种简单的版本。

为了分配一个块，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果找到了一个，那么就(可选地)分割它，并将剩余的部分插入到适当的空闲链表中。如果找不到合适的块，那么就搜索下一个更大的大小类的空闲链表。如此重复，直到找到一个合适的块。如果空闲链表中没有合适的块，那么就向操作系统请求额外的堆内存，从这个新的堆内存中分配出一个块，将剩余部分放置在适当的大小类中。要释放一个块，我们执行合并，并将结果放置到相应的空闲链表中。

分离适配方法是一种常见的选择，C 标准库中提供的 GNU malloc 包就是采用的这种方法，因为这种方法既快速，对内存的使用也很有效率。搜索时间减少了，因为搜索被限制在堆的某个部分，而不是整个堆。内存利用率得到了改善，因为有一个有趣的事：对分离空闲链表的简单的首次适配搜索，其内存利用率近似于对整个堆的最佳适配搜索的内存利用率。

3. 伙伴系统

伙伴系统(buddy system)是分离适配的一种特例，其中每个大小类都是 2 的幂。基本的思路是假设一个堆的大小为 2^m 个字，我们为每个块大小 2^k 维护一个分离空闲链表，其中 $0 \leq k \leq m$ 。请求块大小向上舍入到最接近的 2 的幂。最开始时，只有一个大小为 2^m 个字的空闲块。

为了分配一个大小为 2^k 的块，我们找到第一个可用的、大小为 2^j 的块，其中 $k \leq j \leq m$ 。如果 $j = k$ ，那么我们就完成了。否则，我们递归地二分割这个块，直到 $j = k$ 。当我们进行这样的分割时，每个剩下的半块(也叫做伙伴)被放置在相应的空闲链表中。要释放一个大小为 2^k 的块，我们继续合并空闲的伙伴。当遇到一个已分配的伙伴时，我们就停止合并。

关于伙伴系统的一个关键事实是，给定地址和块的大小，很容易计算出它的伙伴的地址。例如，一个块，大小为 32 字节，地址为：

$xxx \cdots x00000$

它的伙伴的地址为

$xxx \cdots x10000$

换句话说，一个块的地址和它的伙伴的地址只有一位不相同。

伙伴系统分配器的主要优点是它的快速搜索和快速合并。主要缺点是要求块大小为 2 的幂可能导致显著的内部碎片。因此，伙伴系统分配器不适合通用目的的工作负载。然而，对于某些特定应用的工作负载，其中块大小预先知道是 2 的幂，伙伴系统分配器就很吸引人了。

9.10 垃圾收集

在诸如 C malloc 包这样的显式分配器中，应用通过调用 malloc 和 free 来分配和释放堆块。应用要负责释放所有不再需要的已分配块。

未能释放已分配的块是一种常见的编程错误。例如，考虑下面的 C 函数，作为处理的一部分，它分配一块临时存储：

```

1 void garbage()
2 {
3     int *p = (int *)Malloc(15213);
4
5     return; /* Array p is garbage at this point */
6 }

```

因为程序不再需要 `p`，所以在 `garbage` 返回前应该释放 `p`。不幸的是，程序员忘了释放这个块。它在程序的生命周期内都保持为已分配状态，毫无必要地占用着本来可以用来满足后面分配请求的堆空间。

垃圾收集器(garbage collector)是一种动态内存分配器，它自动释放程序不再需要的已分配块。这些块被称为垃圾(garbage)(因此术语就称之为垃圾收集器)。自动回收堆存储的过程叫做垃圾收集(garbage collection)。在一个支持垃圾收集的系统中，应用显式分配堆块，但是从不显示地释放它们。在 C 程序的上下文中，应用调用 `malloc`，但是从不调用 `free`。反之，垃圾收集器定期识别垃圾块，并相应地调用 `free`，将这些块放回到空闲链表中。

垃圾收集可以追溯到 John McCarthy 在 20 世纪 60 年代早期在 MIT 开发的 Lisp 系统。它是诸如 Java、ML、Perl 和 Mathematica 等现代语言系统的一个重要部分，而且它仍然是一个重要而活跃的研究领域。有关文献描述了大量的垃圾收集方法，其数量令人吃惊。我们的讨论局限于 McCarthy 独创的 Mark&Sweep(标记 & 清除)算法，这个算法很有趣，因为它可以建立在已存在的 `malloc` 包的基础之上，为 C 和 C++ 程序提供垃圾收集。

9.10.1 垃圾收集器的基本知识

垃圾收集器将内存视为一张有向可达图(reachability graph)，其形式如图 9-49 所示。该图的节点被分成一组根节点(root node)和一组堆节点(heap node)。每个堆节点对应于堆中的一个已分配块。有向边 $p \rightarrow q$ 意味着块 p 中的某个位置指向块 q 中的某个位置。根节点对应于这样一种不在堆中的位置，它们中包含指向堆中的指针。这些位置可以是寄存器、栈里的变量，或者是虚拟内存中读写数据区域内的全局变量。

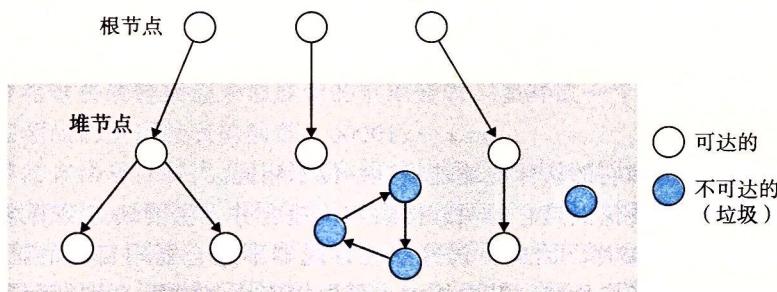


图 9-49 垃圾收集器将内存视为一张有向图

当存在一条从任意根节点出发并到达 p 的有向路径时，我们说节点 p 是可达的(reachable)。在任何时刻，不可达节点对应于垃圾，是不能被应用再次使用的。垃圾收集器的角色是维护可达图的某种表示，并通过释放不可达节点且将它们返回给空闲链表，来定期地回收它们。

像 ML 和 Java 这样的语言的垃圾收集器，对应用如何创建和使用指针有很严格的控制，能够维护可达图的一种精确的表示，因此也就能够回收所有垃圾。然而，诸如 C 和

C++ 这样的语言的收集器通常不能维持可达图的精确表示。这样的收集器也叫做保守的垃圾收集器(conservative garbage collector)。从某种意义上来说它们是保守的，即每个可达块都被正确地标记为可达了，而一些不可达节点却可能被错误地标记为可达。

收集器可以按需提供它们的服务，或者它们可以作为一个和应用并行的独立线程，不断地更新可达图和回收垃圾。例如，考虑如何将一个 C 程序的保守的收集器加入到已存在的 malloc 包中，如图 9-50 所示。

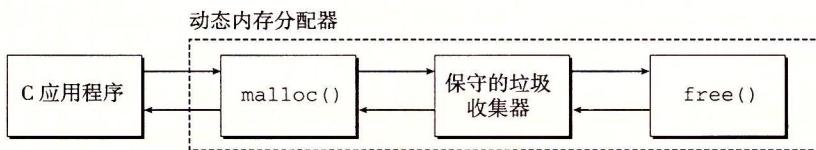


图 9-50 将一个保守的垃圾收集器加入到 C 的 malloc 包中

无论何时需要堆空间时，应用都会用通常的方式调用 malloc。如果 malloc 找不到一个合适的空闲块，那么它就调用垃圾收集器，希望能够回收一些垃圾到空闲链表。收集器识别出垃圾块，并通过调用 free 函数将它们返回给堆。关键的思想是收集器代替应用去调用 free。当对收集器的调用返回时，malloc 重试，试图发现一个合适的空闲块。如果还是失败了，那么它就会向操作系统要求额外的内存。最后，malloc 返回一个指向请求块的指针(如果成功)或者返回一个空指针(如果不成功)。

9.10.2 Mark & Sweep 垃圾收集器

Mark&Sweep 垃圾收集器由标记(mark)阶段和清除(sweep)阶段组成，标记阶段标记出根节点的所有可达的和已分配的后继，而后面的清除阶段释放每个未被标记的已分配块。块头部中空闲的低位中的一位通常用来表示这个块是否被标记了。

- 我们对 Mark&Sweep 的描述将假设使用下列函数，其中 ptr 定义为 `typedef void *ptr;`
- `ptr isPtr(ptr p)`。如果 p 指向一个已分配块中的某个字，那么就返回一个指向这个块的起始位置的指针 b。否则返回 NULL。
 - `int blockMarked(ptr b)`。如果块 b 是已标记的，那么就返回 true。
 - `int blockAllocated(ptr b)`。如果块 b 是已分配的，那么就返回 true。
 - `void markBlock(ptr b)`。标记块 b。
 - `int length(b)`。返回块 b 的以字为单位的长度(不包括头部)。
 - `void unmarkBlock(ptr b)`。将块 b 的状态由已标记的改为未标记的。
 - `ptr nextBlock(ptr b)`。返回堆中块 b 的后继。

标记阶段为每个根节点调用一次图 9-51a 所示的 mark 函数。如果 p 不指向一个已分配并且未标记的堆块，mark 函数就立即返回。否则，它就标记这个块，并对块中的每个字递归地调用它自己。每次对 mark 函数的调用都标记某个根节点的所有未标记并且可达的后继节点。在标记阶段的末尾，任何未标记的已分配块都被认定为是不可达的，是垃圾，可以在清除阶段回收。

清除阶段是对图 9-51b 所示的 sweep 函数的一次调用。sweep 函数在堆中每个块上反复循环，释放它所遇到的所有未标记的已分配块(也就是垃圾)。

图 9-52 展示了一个小堆的 Mark&Sweep 的图形化解释。块边界用粗线条表示。每个方块对应于内存中的一个字。每个块有一个字的头部，要么是已标记的，要么是未标记的。

```

void mark(ptr p) {
    if ((b = isPtr(p)) == NULL)
        return;
    if (blockMarked(b))
        return;
    markBlock(b);
    len = length(b);
    for (i=0; i < len; i++)
        mark(b[i]);
    return;
}

```

a) mark 函数

```

void sweep(ptr b, ptr end) {
    while (b < end) {
        if (blockMarked(b))
            unmarkBlock(b);
        else if (blockAllocated(b))
            free(b);
        b = nextBlock(b);
    }
    return;
}

```

b) sweep 函数

图 9-51 mark 和 sweep 函数的伪代码

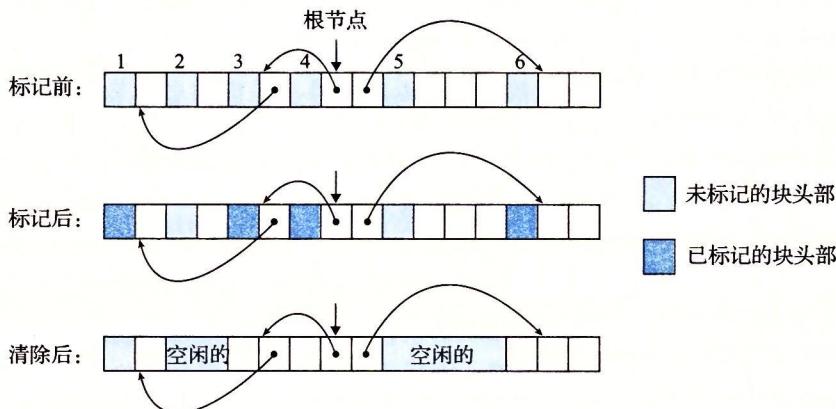


图 9-52 Mark & Sweep 示例。注意这个示例中的箭头表示内存引用，而不是空闲链表指针

初始情况下，图 9-52 中的堆由六个已分配块组成，其中每个块都是未分配的。第 3 块包含一个指向第 1 块的指针。第 4 块包含指向第 3 块和第 6 块的指针。根指向第 4 块。在标记阶段之后，第 1 块、第 3 块、第 4 块和第 6 块被做了标记，因为它们是从根节点可达的。第 2 块和第 5 块是未标记的，因为它们是不可达的。在清除阶段之后，这两个不可达块被回收到空闲链表。

9.10.3 C 程序的保守 Mark & Sweep

Mark & Sweep 对 C 程序的垃圾收集是一种合适的方法，因为它可以就地工作，而不需要移动任何块。然而，C 语言为 isPtr 函数的实现造成了一些有趣的挑战。

第一，C 不会用任何类型信息来标记内存位置。因此，对 isPtr 没有一种明显的方式来判断它的输入参数 p 是不是一个指针。第二，即使我们知道 p 是一个指针，对 isPtr 也没有明显的方式来判断 p 是否指向一个已分配块的有效载荷中的某个位置。

对后一问题的解决方法是将已分配块集合维护成一棵平衡二叉树，这棵树保持着这样一个属性：左子树中的所有块都放在较小的地址处，而右子树中的所有块都放在较大的地址处。如图 9-53 所示，这就要求每个已分配块的头部里有两个附加字段(left 和 right)。每个字段指向某个已分配块的头部。isPtr(ptr p) 函数用树来执行对已分配块的二分查找。在每一步中，它依赖于块头部中的大小字段来判断 p 是否落在这个块的范围之内。

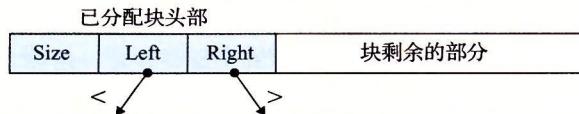


图 9-53 一棵已分配块的平衡树中的左右指针

平衡树方法保证会标记所有从根节点可达的节点，从这个意义上来说它是正确的。这是一个必要的保证，因为应用程序的用户当然不会喜欢把他们的已分配块过早地返回给空闲链表。然而，这种方法从某种意义上而言又是保守的，因为它可能不正确地标记实际上不可达的块，因此它可能不会释放某些垃圾。虽然这并不影响应用程序的正确性，但是这可能导致不必要的外部碎片。

C 程序的 Mark & Sweep 收集器必须是保守的，其根本原因是 C 语言不会用类型信息来标记内存位置。因此，像 int 或者 float 这样的标量可以伪装成指针。例如，假设某个可达的已分配块在它的有效载荷中包含一个 int，其值碰巧对应于某个其他已分配块 b 的有效载荷中的一个地址。对收集器而言，是没有办法推断出这个数据实际上是 int 而不是指针。因此，分配器必须保守地将块 b 标记为可达，尽管事实上它可能是不可达的。

9.11 C 程序中常见的与内存有关的错误

对 C 程序员来说，管理和使用虚拟内存可能是个困难的、容易出错的任务。与内存有关的错误属于那些最令人惊恐的错误，因为它们在时间和空间上，经常在距错误源一段距离之后才表现出来。将错误的数据写到错误的位置，你的程序可能在最终失败之前运行了好几个小时，且使程序中止的位置距离错误的位置已经很远了。我们用一些常见的与内存有关错误的讨论，来结束对虚拟内存的讨论。

9.11.1 间接引用坏指针

正如我们在 9.7.2 节中学到的，在进程的虚拟地址空间中有较大的洞，没有映射到任何有意义的数据。如果我们试图间接引用一个指向这些洞的指针，那么操作系统就会以段异常中止程序。而且，虚拟内存的某些区域是只读的。试图写这些区域将会以保护异常中止这个程序。

间接引用坏指针的一个常见示例是经典的 scanf 错误。假设我们想要使用 scanf 从 stdin 读一个整数到一个变量。正确的方法是传递给 scanf 一个格式串和变量的地址：

```
scanf("%d", &val)
```

然而，对于 C 程序员初学者而言（对有经验者也是如此！），很容易传递 val 的内容，而不是它的地址：

```
scanf("%d", val)
```

在这种情况下，scanf 将把 val 的内容解释为一个地址，并试图将一个字写到这个位置。在最好的情况下，程序立即以异常终止。在最糟糕的情况下，val 的内容对应于虚拟内存的某个合法的读/写区域，于是我们就覆盖了这块内存，这通常会在相当长的一段时间以后造成灾难性的、令人困惑的后果。

9.11.2 读未初始化的内存

虽然 bss 内存位置（诸如未初始化的全局 C 变量）总是被加载器初始化为零，但是对于堆内存却并不是这样的。一个常见的错误就是假设堆内存被初始化为零：