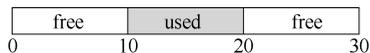


第 17 章 空闲空间管理

本章暂且将对虚拟内存的讨论放在一边，来讨论所有内存管理系统的一个基本方面，无论是 `malloc` 库（管理进程中堆的页），还是操作系统本身（管理进程的地址空间）。具体来说，我们会讨论空闲空间管理（`free-space management`）的一些问题。

让问题更明确一点。管理空闲空间当然可以很容易，我们会在讨论分页概念时看到。如果需要管理的空间被划分为固定大小的单元，就很容易。在这种情况下，只需要维护这些大小固定的单元的列表，如果有请求，就返回列表中的第一项。

如果要管理的空闲空间由大小不同的单元构成，管理就变得困难（而且有趣）。这种情况出现在用户级的内存分配库（如 `malloc()` 和 `free()`），或者操作系统用分段（`segmentation`）的方式实现虚拟内存。在这两种情况下，出现了外部碎片（`external fragmentation`）的问题：空闲空间被分割成不同大小的小块，成为碎片，后续的请求可能失败，因为没有一块足够大的连续空闲空间，即使这时总的空闲空间超出了请求的大小。



上面展示了该问题的一个例子。在这个例子中，全部可用空闲空间是 20 字节，但被切成两个 10 字节大小的碎片，导致一个 15 字节的分配请求失败。所以本章需要解决的问题是：

关键问题：如何管理空闲空间

要满足变长的分配请求，应该如何管理空闲空间？什么策略可以让碎片最小化？不同方法的时间和空间开销如何？

17.1 假设

本章的大多数讨论，将聚焦于用户级内存分配库中分配程序的辉煌历史。我们引用了 Wilson 的出色调查 [W+95]，有兴趣的读者可以从原文了解更多细节^①。

我们假定基本的接口就像 `malloc()` 和 `free()` 提供的那样。具体来说，`void * malloc(size t size)` 需要一个参数 `size`，它是应用程序请求的字节数。函数返回一个指针（没有具体的类型，在 C 语言的术语中是 `void` 类型），指向这样大小（或较大一点）的一块空间。对应的函数 `void free(void *ptr)` 函数接受一个指针，释放对应的内存块。请注意该接口的隐含意义，在释放空间时，用户不需告知库这块空间的大小。因此，在只传入一个指针的情况下，库必须能够弄清楚这块内存的大小。我们将在稍后介绍是如何得知的。

^① 它有近 80 页长。因此，你必须真的对它感兴趣！

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）。该结构包含了管理内存区域中所有空闲块的引用。当然，该数据结构不一定真的是列表，而只是某种可以追踪空闲空间的数据结构。

进一步假设，我们主要关心的是外部碎片（external fragmentation），如上所述。当然，分配程序也可能有内部碎片（internal fragmentation）的问题。如果分配程序给出的内存块超出请求的大小，在这种块中超出请求的空间（因此而未使用）就被认为是内部碎片（因为浪费发生在已分配单元的内部），这是另一种形式的空间浪费。但是，简单起见，同时也因为它更有趣，这里主要讨论外部碎片。

我们还假设，内存一旦被分配给客户，就不可以被重定位到其他位置。例如，一个程序调用 `malloc()`，并获得一个指向堆中一块空间的指针，这块区域就“属于”这个程序了，库不再能够移动，直到程序调用相应的 `free()` 函数将它归还。因此，不可能进行紧凑（compaction）空闲空间的操作，从而减少碎片^①。但是，操作系统层在实现分段（segmentation）时，却可以通过紧凑来减少碎片（正如第 16 章讨论的那样）。

最后我们假设，分配程序所管理的是连续的一块字节区域。在一些情况下，分配程序可以要求这块区域增长。例如，一个用户级的内存分配库在空间快用完时，可以向内核申请增加堆空间（通过 `sbrk` 这样的系统调用），但是，简单起见，我们假设这块区域在整个生命周期内大小固定。

17.2 底层机制

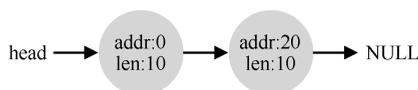
在深入策略细节之前，我们先来介绍大多数分配程序采用的通用机制。首先，探讨空间分割与合并的基本知识。其次，看看如何快速并相对轻松地追踪已分配的空间。最后，讨论如何利用空闲区域的内部空间维护一个简单的列表，来追踪空闲和已分配的空间。

分割与合并

空闲列表包含一组元素，记录了堆中的哪些空间还没有分配。假设有下面的 30 字节的堆：



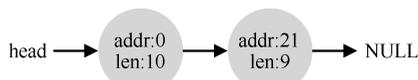
这个堆对应的空闲列表会有两个元素，一个描述第一个 10 字节的空闲区域（字节 0~9），一个描述另一个空闲区域（字节 20~29）：



^① 一旦将指向内存块的一个指针交给 C 程序，通常很难确定所有对该区域的引用（指针），这些引用（指针）可能存储在其他变量中，或者甚至在执行的某个时刻存储在寄存器中。在更强类型的、带垃圾收集的语言中，情况可能并非如此，因此可以用紧凑技术来减少碎片。

通过上面的介绍可以看出，任何大于 10 字节的分配请求都会失败（返回 NULL），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。但是，如果申请小于 10 字节空间，会发生什么？

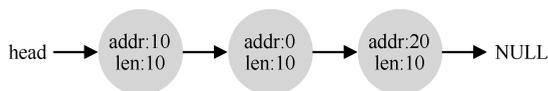
假设我们只申请一个字节内存。这种情况下，分配程序会执行所谓的分割（splitting）动作：它找到一块可以满足请求的空闲空间，将其分割，第一块返回给用户，第二块留在空闲列表中。在我们的例子中，假设这时遇到申请一个字节请求，分配程序选择使用第二块空闲空间，对 malloc() 的调用会返回 20（1 字节分配区域的地址），空闲列表会变成这样：



从上面可以看出，空闲列表基本没有变化，只是第二个空闲区域的起始位置由 20 变成 21，长度由 10 变为 9 了^①。因此，如果请求的空间大小小于某块空闲块，分配程序通常会进行分割。

许多分配程序中因此也有一种机制，名为合并（coalescing）。还是看前面的例子（10 字节的空闲空间，10 字节的已分配空间，和另外 10 字节的空闲空间）。

对于这个（小）堆，如果应用程序调用 free(10)，归还堆中间的空间，会发生什么？如果只是简单地将这块空闲空间加入空闲列表，不多想想，可能得到如下的结果：



问题出现了：尽管整个堆现在完全空闲，但它似乎被分割成了 3 个 10 字节的区域。这时，如果用户请求 20 字节的空间，简单遍历空闲列表会找不到这样的空闲块，因此返回失败。

为了避免这个问题，分配程序会在释放一块内存时合并可用空间。想法很简单：在归还一块空闲内存时，仔细查看要归还的内存块的地址以及邻近的空闲空间块。如果新归还的空间与一个原有空闲块相邻（或两个，就像这个例子），就将它们合并为一个较大的空闲块。通过合并，最后空闲列表应该像这样：



实际上，这是堆的空闲列表最初的样子，在所有分配之前。通过合并，分配程序可以更好地确保大块的空闲空间能提供给应用程序。

追踪已分配空间的大小

你可能注意到，free(void *ptr) 接口没有块大小的参数。因此它是假定，对于给定的指针，内存分配库可以很快确定要释放空间的大小，从而将它放回空闲列表。

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，它在内存中，通常就在返回的内存块之前。我们再看一个例子（见图 17.1）。在这个例子中，我

^① 这里的讨论假设没有头块，这是我们现在做出的一个不现实但简化的假设。

们检查一个 20 字节的已分配块，由 `ptr` 指着，设想用户调用了 `malloc()`，并将结果保存在 `ptr` 中：`ptr = malloc(20)`。

该头块中至少包含所分配空间的大小（这个例子中是 20）。它也可能包含一些额外的指针来加速空间释放，包含一个幻数来提供完整性检查，以及其他信息。我们假定，一个简单的头块包含了分配空间的大小和一个幻数：

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

上面的例子看起来会像图 17.2 的样子。用户调用 `free(ptr)` 时，库会通过简单的指针运算得到头块的位置：

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

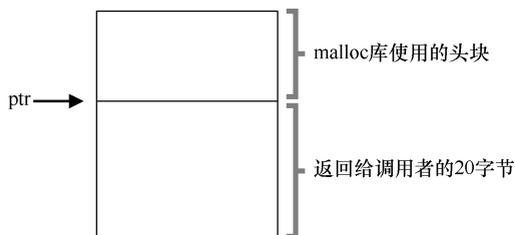


图 17.1 一个已分配的区域加上头块



图 17.2 头块的具体内容

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，作为正常性检查（`assert(hptr->magic == 1234567)`），并简单计算要释放的空间大小（即头块的大小加区域长度）。请注意前一句话中一个虽小但重要的细节：实际释放的是头块大小加上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，库不是寻找大小为 N 的空闲块，而是寻找 N 加上头块大小的空闲块。

嵌入空闲列表

到目前为止，我们这个简单的空闲列表还只是一个概念上的存在，它就是一个列表，描述了堆中的空闲内存块。但如何在空闲内存自己内部建立这样一个列表呢？

在更典型的列表中，如果要分配新节点，你会调用 `malloc()` 来获取该节点所需的内存。遗憾的是，在内存分配库内，你无法这么做！你需要在空闲空间本身中建立空闲空间列表。虽然听起来有点奇怪，但别担心，这是可以做到的。

假设我们需要管理一个 4096 字节的内存块（即堆是 4KB）。为了将它作为一个空闲空间列表来管理，首先要初始化这个列表。开始，列表中只有一个条目，记录了大小为 4096 的空间（减去头块的大小）。下面是该列表中一个节点描述：

```
typedef struct node_t {
    int size;
```

```

    struct    node_t *next;
} node_t;

```

现在来看一些代码，它们初始化堆，并将空闲列表的第一个元素放在该空间中。假设堆构建在某块空闲空间上，这块空间通过系统调用 `mmap()` 获得。这不是构建这种堆的唯一选择，但在这个例子中很合适。下面是代码：

```

// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size    = 4096 - sizeof(node_t);
head->next    = NULL;

```

执行这段代码之后，列表的状态是它只有一个条目，记录大小为 4088。

是的，这是一个小堆，但对我们是一个很好的例子。`head` 指针指向这块区域的起始地址，假设是 16KB（尽管任何虚拟地址都可以）。堆看起来如图 17.3 所示。

现在，假设有一个 100 字节的内存请求。为了满足这个请求，库首先要找到一个足够大小的块。因为只有一个 4088 字节的块，所以选中这个块。然后，这个块被分割（split）为两块：一块足够满足请求（以及头块，如前所述），一块是剩余的空闲块。假设记录头块为 8 个字节（一个整数记录大小，一个整数记录幻数），堆中的空间如图 17.4 所示。

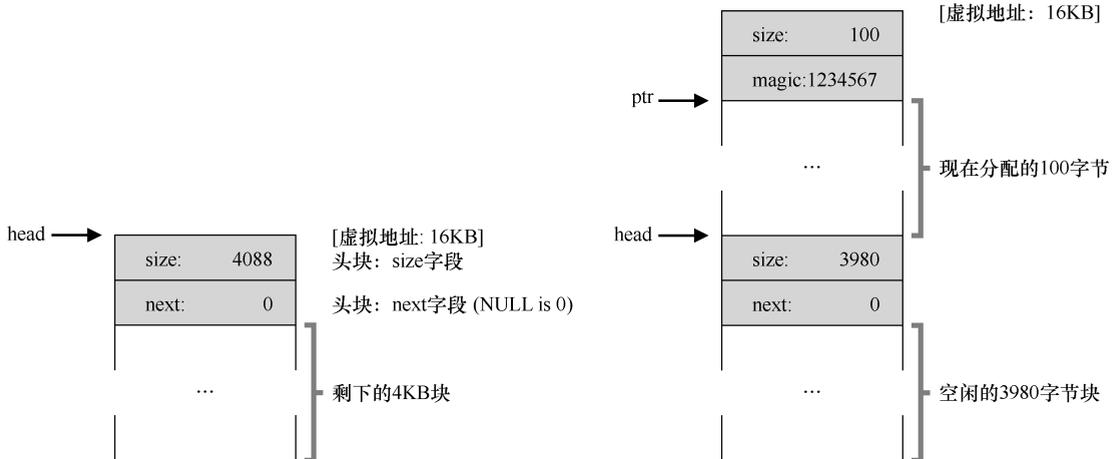


图 17.3 有一个空闲块的堆

图 17.4 在一次分配之后的堆

至此，对于 100 字节的请求，库从原有的一个空闲块中分配了 108 字节，返回指向它的一个指针（在上图中用 `ptr` 表示），并在其之前连续的 8 字节中记录头块信息，供未来的 `free()` 函数使用。同时将列表中的空闲节点缩小为 3980 字节（4088-108）。

现在再来看该堆，其中有 3 个已分配区域，每个 100（加上头块是 108）。这个堆如图 17.5 所示。

可以看出，堆的前 324 字节已经分配，因此我们看到该空间中有 3 个头块，以及 3 个 100 字节的用户使用空间。空闲列表还是无趣：只有一个节点（由 `head` 指向），但在 3 次分割后，现在大小只有 3764 字节。但如果用户程序通过 `free()` 归还一些内存，会发生什么？

在这个例子中，应用程序调用 `free(16500)`，归还了中间的一块已分配空间（内存块的起

始地址 16384 加上前一块的 108，和这一块的头块的 8 字节，就得到了 16500)。这个值在前图中用 `sptr` 指向。

库马上弄清楚了这块要释放空间的大小，并将空闲块加回空闲列表。假设我们将它插入到空闲列表的头位置，该空间如图 17.6 所示。

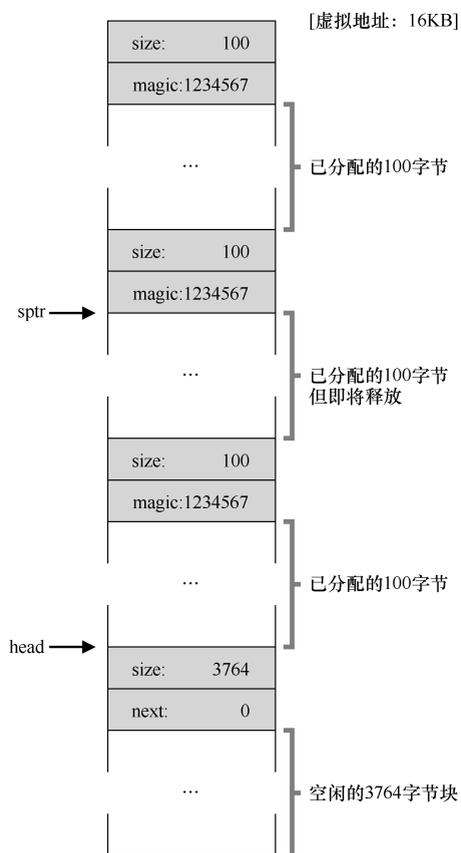


图 17.5 空闲空间和 3 个已分配块

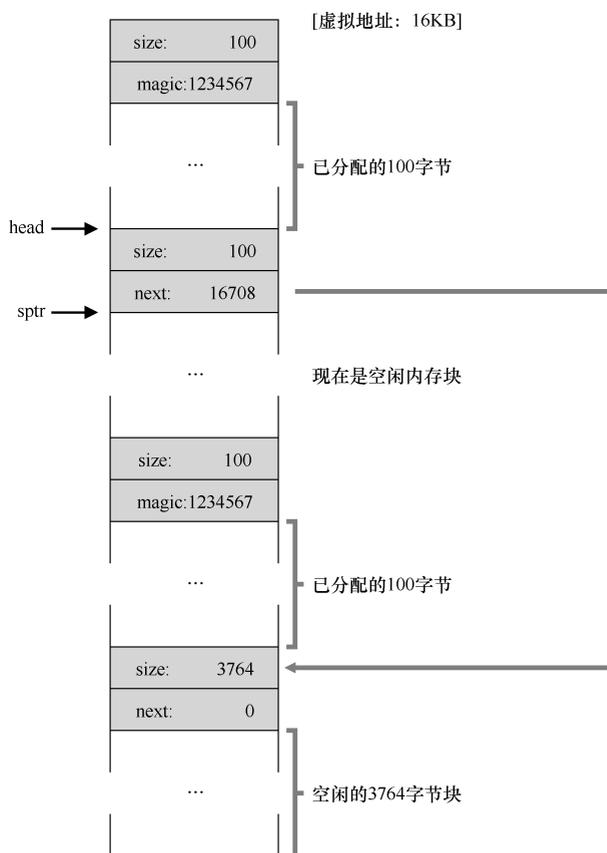


图 17.6 空闲空间和两个已分配的块

现在的空闲列表包括一个小空闲块（100 字节，由列表的头指向）和一个大空闲块（3764 字节）。

我们的列表终于有不止一个元素了！是的，空闲空间被分割成了两段，但很常见。

最后一个例子：现在假设剩余的两块已分配的空间也被释放。没有合并，空闲列表将非常破碎，如图 17.7 所示。

从图中可以看出，我们现在一团糟！为什么？简单，我们忘了合并（`coalesce`）列表项，虽然整个内存空间是空闲的，但却被分成了小段，因此形成了碎片化的内存空间。解决方案很简单：遍历列表，合并（`merge`）相邻块。完成之后，堆又成了一个整体。

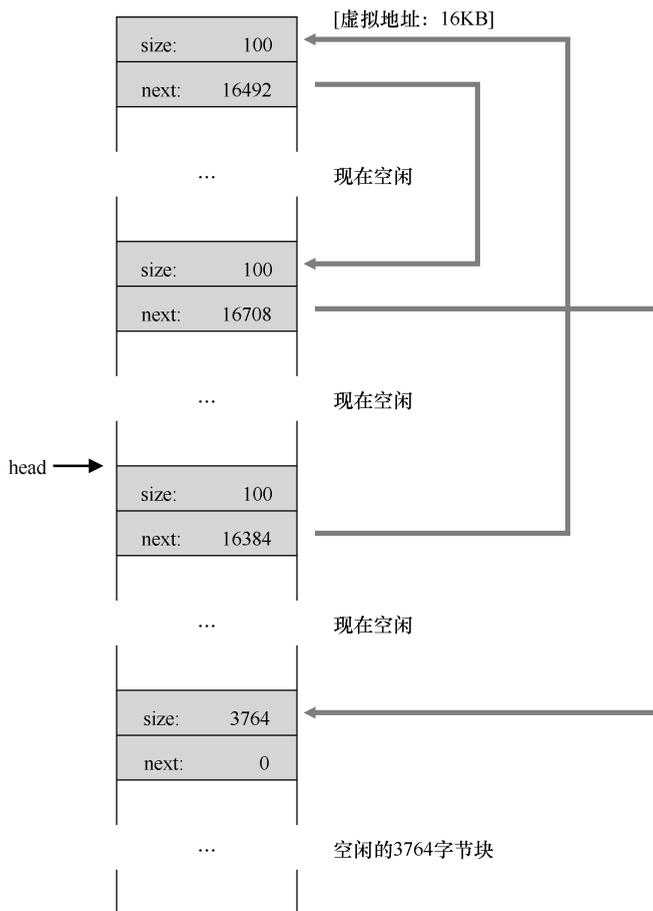


图 17.7 未合并的空闲空间列表

让堆增长

我们应该讨论最后一个很多内存分配库中都有的机制。具体来说，如果堆中的内存空间耗尽，应该怎么办？最简单的方式就是返回失败。在某些情况下这也是唯一的选择，因此返回 NULL 也是一种体面的方式。别太难过！你尽力了，即使失败，你也虽败犹荣。

大多数传统的分配程序会从很小的堆开始，当空间耗尽时，再向操作系统申请更大的空间。通常，这意味着它们进行了某种系统调用（例如，大多数 UNIX 系统中的 `sbrk`），让堆增长。操作系统在执行 `sbrk` 系统调用时，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。这时，就有了更大的堆，请求就可以成功满足。

17.3 基本策略

既然有了这些底层机制，让我们来看看管理空闲空间的一些基本策略。这些方法大多基于简单的策略，你也能想到。在阅读之前试试，你是否能想出所有的选择（也许还有新策略！）。

理想的分配程序可以同时保证快速和碎片最小化。遗憾的是，由于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。所以我们不会描述“最好”的策略，而是介绍一些基本的选择，并讨论它们的优缺点。

最优匹配

最优匹配（best fit）策略非常简单：首先遍历整个空闲列表，找到和请求大小一样或更大的空闲块，然后返回这组候选者中最小的一块。这就是所谓的最优匹配（也可以称为最小匹配）。只需要遍历一次空闲列表，就足以找到正确的块并返回。

最优匹配背后的想法很简单：选择最接近用户请求大小的块，从而尽量避免空间浪费。然而，这有代价。简单的实现在遍历查找正确的空闲块时，要付出较高的性能代价。

最差匹配

最差匹配（worst fit）方法与最优匹配相反，它尝试找最大的空闲块，分割并满足用户需求后，将剩余的块（很大）加入空闲列表。最差匹配尝试在空闲列表中保留较大的块，而不是向最优匹配那样可能剩下很多难以利用的小块。但是，最差匹配同样需要遍历整个空闲列表。更糟糕的是，大多数研究表明它的表现非常差，导致过量的碎片，同时还有很高的开销。

首次匹配

首次匹配（first fit）策略就是找到第一个足够大的块，将请求的空间返回给用户。同样，剩余的空闲空间留给后续请求。

首次匹配有速度优势（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。因此，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有序，合并操作会很容易，从而减少了内存碎片。

下次匹配

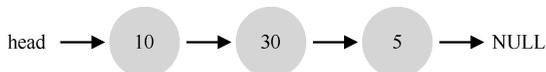
不同于首次匹配每次都从列表的开始查找，下次匹配（next fit）算法多维护一个指针，指向上一次查找结束的位置。其想法是对空闲空间的查找操作扩散到整个列表中去，避免对列表开头频繁的分割。这种策略的性能与首次匹配很接近，同样避免了遍历查找。

例子

下面是上述策略的一些例子。设想一个空闲列表包含 3 个元素，长度依次为 10、30、20（我们暂时忽略头块和其他细节，只关注策略的操作方式）：



假设有一个 15 字节的内存请求。最优匹配会遍历整个空闲列表，发现 20 字节是最优匹配，因为它是满足请求的最小空闲块。结果空闲列表变为：



本例中发生的情况，在最优匹配中常常发生，现在留下了一个小空闲块。最差匹配类似，但会选择最大的空闲块进行分割，在本例中是 30。结果空闲列表变为：



在这个例子中，首次匹配会和最差匹配一样，也发现满足请求的第一个空闲块。不同的是查找开销，最优匹配和最差匹配都需要遍历整个列表，而首次匹配只找到第一个满足需求的块即可，因此减少了查找开销。

这些例子只是内存分配策略的肤浅分析。真实场景下更详细的分析和更复杂的分配行为（如合并），需要更深入的理解。也许可以作为作业，你说呢？

17.4 其他方式

除了上述基本策略外，人们还提出了许多技术和算法，来改进内存分配。这里我们列出一些来供你考虑（就是让你多一些思考，不只局限于最优匹配）。

分离空闲列表

一直以来有一种很有趣的方式叫作分离空闲列表（*segregated list*）。基本想法很简单：如果某个应用程序经常申请一种（或几种）大小的内存空间，那就用一个独立的列表，只管理这样大小的对象。其他大小的请求都交给更通用的内存分配程序。

这种方法的好处显而易见。通过拿出一部分内存专门满足某种大小的请求，碎片就不再是问题了。而且，由于没有复杂的列表查找过程，这种特定大小的内存分配和释放都很快。

就像所有好主意一样，这种方式也为系统引入了新的复杂性。例如，应该拿出多少内存来专门为某种大小的请求服务，而将剩余的用来满足一般请求？超级工程师 Jeff Bonwick 为 Solaris 系统内核设计的厚块分配程序（*slab allocator*），很优雅地处理了这个问题[B94]。

具体来说，在内核启动时，它为可能频繁请求的内核对象创建一些对象缓存（*object cache*），如锁和文件系统 *inode* 等。这些的对象缓存每个分离了特定大小的空闲列表，因此能够很快地响应内存请求和释放。如果某个缓存中的空闲空间快耗尽时，它就向通用内存分配程序申请一些内存厚块（*slab*）（总量是页大小和对象大小的公倍数）。相反，如果给定厚块中对象的引用计数变为 0，通用的内存分配程序可以从专门的分配程序中回收这些空间，这通常发生在虚拟内存系统需要更多的空间的时候。

补充：了不起的工程师真的了不起

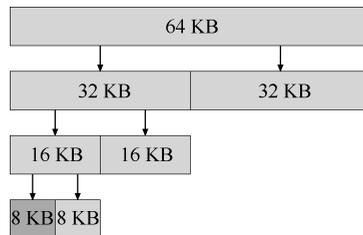
像 Jeff Bonwick 这样的工程师（Jeff Bonwick 不仅写了上面提到的厚块分配程序，还是令人惊叹的文件系统 ZFS 的负责人），是硅谷的灵魂。在每一个伟大的产品或技术后面都有这样一个人（或一小群人），他们的天赋、能力和奉献精神远超众人。Facebook 的 Mark Zuckerberg 曾经说过：“那些在自己的领域中超凡脱俗的人，比那些相当优秀的人强得不是一点点。”这就是为什么，会有人成立自己的公司，然后永远地改变了这个世界（想想 Google、Apple 和 Facebook）。努力工作，你也可能成为这种“以一当百”的人。做不到的话，就和这样的人一起工作，你会明白什么是“听君一席话，胜读十年书”。如果都做不到，那就太难过了。

厚块分配程序比大多数分离空闲列表做得更多，它将列表中的空闲对象保持在预初始化的状态。Bonwick 指出，数据结构的初始化和销毁的开销很大[B94]。通过将空闲对象保持在初始化状态，厚块分配程序避免了频繁的初始化和销毁，从而显著降低了开销。

伙伴系统

因为合并对分配程序很关键，所以人们设计了一些方法，让合并变得简单，一个好例子就是二分伙伴分配程序（binary buddy allocator）[K65]。

在这种系统中，空闲空间首先从概念上被看成大小为 2^N 的大空间。当有一个内存分配请求时，空闲空间被递归地一分为二，直到刚好可以满足请求的大小（再一分为二就无法满足）。这时，请求的块被返回给用户。在下面的例子中，一个 64KB 大小的空闲空间被切分，以便提供 7KB 的块：



在这个例子中，最左边的 8KB 块被分配给用户（如上图中的深灰色部分所示）。请注意，这种分配策略只允许分配 2 的整数次幂大小的空闲块，因此会有内部碎片（internal fragment）的麻烦。

伙伴系统的漂亮之处在于块被释放时。如果将这个 8KB 的块归还给空闲列表，分配程序会检查“伙伴”8KB 是否空闲。如果是，就合二为一，变成 16KB 的块。然后会检查这个 16KB 块的伙伴是否空闲，如果是，就合并这两块。这个递归合并过程继续上溯，直到合并整个内存区域，或者某一个块的伙伴还没有被释放。

伙伴系统运转良好的原因，在于很容易确定某个块的伙伴。怎么找？仔细想想上面例子中的各个块的地址。如果你想得够仔细，就会发现每对互为伙伴的块只有一位不同，正是这一位决定了它们在整个伙伴树中的层次。现在你应该已经大致了解了二分伙伴分配程序的工作方式。更多的细节可以参考 Wilson 的调查[W+95]。

其他想法

上面提到的众多方法都有一个重要的问题，缺乏可扩展性（scaling）。具体来说，就是查找列表可能很慢。因此，更先进的分配程序采用更复杂的数据结构来优化这个开销，牺牲简单性来换取性能。例子包括平衡二叉树、伸展树和偏序树[W+95]。

考虑到现代操作系统通常会有多核，同时会运行多线程的程序（本书之后关于并发的章节将会详细介绍），因此人们做了许多工作，提升分配程序在多核系统上的表现。两个很棒的例子参见 Berger 等人的[B+00]和 Evans 的[E06]，看看文章了解更多细节。

这只是人们为了优化内存分配程序，在长时间内提出的几千种想法中的两种。感兴趣的话可以深入阅读。或者阅读 glibc 分配程序的工作原理[S15]，你会更了解现实的情形。

17.5 小结

在本章中，我们讨论了最基本的内存分配程序形式。这样的分配程序存在于所有地方，与你编写的每个 C 程序链接，也和管理其自身数据结构的内存的底层操作系统链接。与许多系统一样，在构建这样一个系统时需要做许多折中。对分配程序提供的确切工作负载了解得越多，就越能调整它以更好地处理这种工作负载。在现代计算机系统中，构建一个适用于各种工作负载、快速、空间高效、可扩展的分配程序仍然是一个持续的挑战。

参考资料

[B+00] “Hoard: A Scalable Memory Allocator for Multithreaded Applications” Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson ASPLOS-IX, November 2000

Berger 和公司的优秀多处理器系统分配程序。它不仅是一篇有趣的论文，也是能用于指导实战的！

[B94] “The Slab Allocator: An Object-Caching Kernel Memory Allocator” Jeff Bonwick
USENIX '94

一篇关于如何为操作系统内核构建分配程序的好文章，也是如何专门针对特定通用对象大小的一个很好的例子。

[E06] “A Scalable Concurrent malloc(3) Implementation for FreeBSD” Jason Evans

本文详细介绍如何构建一个真正的现代分配程序以用于多处理器。“jemalloc”分配程序今天在 FreeBSD、NetBSD、Mozilla Firefox 和 Facebook 中已广泛使用。

[K65] “A Fast Storage Allocator” Kenneth C. Knowlton

Communications of the ACM, Volume 8, Number 10, October 1965

伙伴分配的常见引用。一个奇怪的事实是：Knuth 不是把这个想法归功于 Knowlton，而是归功于获得诺贝尔奖的经济学家 Harry Markowitz。另一个奇怪的事实是：Knuth 通过秘书收发他的所有电子邮件。他不会

自己发送电子邮件，而是告诉他的秘书要发送什么邮件，然后秘书负责发送电子邮件。最后一个关于 Knuth 的事实：他创建了 TeX，这是用于排版本书的工具。这是一个惊人的软件^①。

[S15] “Understanding glibc malloc” Sploitfun

深入了解 glibc malloc 是如何工作的。本文详细得令人惊讶，一篇非常好的阅读材料。

[W+95] “Dynamic Storage Allocation: A Survey and Critical Review” Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles International Workshop on Memory Management

Kinross, Scotland, September 1995

对内存分配的许多方面进行了卓越且深入的调查，比这个小小的章节中所含的内容拥有更多的细节！

作业

程序 `malloc.py` 让你探索本章中描述的简单空闲空间分配程序的行为。有关其基本操作的详细信息，请参见 README 文件。

问题

1. 首先运行 `flag -n 10 -H 0 -p BEST -s 0` 来产生一些随机分配和释放。你能预测 `malloc()/free()` 会返回什么吗？你可以在每次请求后猜测空闲列表的状态吗？随着时间的推移，你对空闲列表有什么发现？

2. 使用最差匹配策略搜索空闲列表（`-p WORST`）时，结果有何不同？什么改变了？

3. 如果使用首次匹配（`-p FIRST`）会如何？使用首次匹配时，什么变快了？

4. 对于上述问题，列表在保持有序时，可能会影响某些策略找到空闲位置所需的时间。使用不同的空闲列表排序（`-l ADDRSORT`，`-l SIZESORT +`，`-l SIZESORT-`）查看策略和列表排序如何相互影响。

5. 合并空闲列表可能非常重要。增加随机分配的数量（比如说 `-n 1000`）。随着时间的推移，大型分配请求会发生什么？在有和没有合并的情况下运行（即不用和采用 `-C` 标志）。你看到了什么结果差异？每种情况下的空闲列表有多大？在这种情况下，列表的排序是否重要？

6. 将已分配百分比 `-P` 改为高于 50，会发生什么？它接近 100 时分配会怎样？接近 0 会怎样？

7. 要生成高度碎片化的空闲空间，你可以提出怎样的具体请求？使用 `-A` 标志创建碎片化的空闲列表，查看不同的策略和选项如何改变空闲列表的组织。

^① 实际上我们使用 LaTeX，它基于 L^AT_EX 对 TeX 的补充，但二者非常相似。