09 | 深入理解堆：malloc和内存池是怎么回事？

2021-11-12 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳
时长 20:43 大小 18.98M

你好，我是海纳。

在第 3 节课，我们讲到线性地址空间按照功能的不同，可以分为不同的区域。同时，我们还简单介绍了，如何使用 sbrk 和 mmap 这两个系统调用，向操作系统申请堆内存。

其实，堆内存是程序员打交道最多的一块区域，无论是哪种编程语言，正确合理并高效地使用堆内存，都是极具挑战的一件事情。对程序调优是系统程序员常见的工作任务，而堆内存的管理和分配恰恰是最容易出现性能瓶颈的模块。

不过，sbrk 和 mmap 这两个系统调用分配内存效率比较低，我们在第 5 节课讲过，进程的内核态和用户态的区别，执行系统调用是要进入内核态的，运行态的切换会耗费不少时间。为了解决这个问题，人们倾向于使用系统调用来分配大块内存，然后再把这块内存分割成更小的块，以方便程序员使用，这样可以提升分配的效率。

在 C 语言的运行时库里，这个工作是由 malloc 函数负责的。但有时候 C 语言的原生 malloc 实现还是不能满足特定应用的性能要求，这就需要程序员来实现符合自己应用要求的内存池，以便自己进行内存的分配和释放。

这节课，我们就一起来学习，如何对通过系统调用申请来的大块内存进行更精细化的管理。通过这节课的学习，你将了解到堆内存管理的常用方法，以及内存泄露、double free 等常见的内存问题产生的原因和排查方法，从而提高自己分析和解决内存问题的能力。

malloc 的基本功能

正如这节课开头讲的，向操作系统申请来的大块内存，需要分割成合适的大小，才能让程序员正常使用，其实这个任务是由 glibc 承担的。

glibc 是 C 语言的运行时库，C 语言中常用的函数，例如 printf、scanf、memcpy 和 strcat 等等，它们的实现都在 glibc.so 中。通过后缀名，你可能也猜到了，glibc 是一种动态链接库，它的工作原理与第 7 节课和第 8 节课里所讲的动态链接库是一样的，与普通的动态库相比，它并没有什么特别之处。

我们今天要讲的就是 glibc 中用于内存管理的两个重要函数：malloc 和 free。我们先展示一下 malloc 和 free 的用法，请看下面这段 C 代码：

复制代码

```
#include <stdio.h>
#include <malloc.h>

int main() {
  void *p = malloc(16);
  printf("%p\n", p);
  free(p);
  return 0;
}
```

上述代码中应用程序通过 malloc 函数申请了一块内存，并把这块内存的起始地址打印了出来，然后再通过 free 函数释放这块内存。通过第 1 节课和第 3 节课的学习，我们已经知道，打印的结果实际上是申请的内存块的线性地址。更具体一点，这块空间位于堆中。

malloc 实现的基本原理是先向操作系统申请一块比较大的内存，然后再通过各种优化手段让内存分配的效率最大化。在 glibc 的实现里，malloc 函数在向操作系统申请堆内存时，会使用 mmap，以 4K 的整数倍一次申请多个页。这样的话，mmap 的区域就会以页对齐，页与页之间的排列非常整齐，避免了出现内存碎片。

从这个角度看，glibc 中的 malloc 方法非常像批发商，它从供应商操作系统那里一次批发了很大的内存，然后以零销的方式一点点分配出去，而且它不光负责销售，还负责售后（分配到的内存可以使用 free 退货）。

要想又好又快地正确使用堆内存，一个很重要的方式就是对内存做精细化管理。

malloc 的实现原理

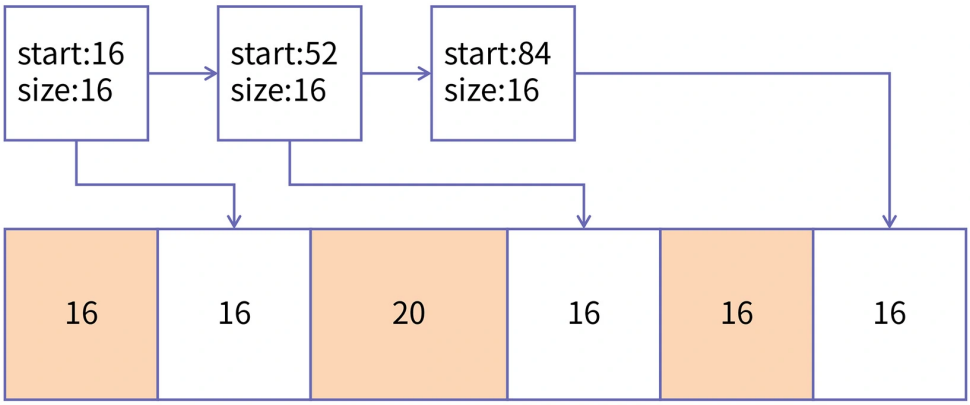
内存的精细化管理，我们要考虑两个因素，**一是分配和回收的效率，二是内存区域的有效利用率**，内存区域的有效利用率又包含两个方面，一个方面是每一小块内存内部是否被合理利用，另一个方面是块与块之间是否存在无法利用的小块内存。

你可以用建筑规划来进行类比。我们有一块很平整的地，如果一个建筑设计得不合理，比如本来只有一千个学生，但却修了一个五千人的体育场，这就是内部空间的浪费。如果各个建筑都是奇形怪状的（不能对齐），那么建筑与建筑之间的不可用的地块就会增多，这就是外部空间的浪费，或者称为碎片。

对小块内存进行精细化管理，最常用的数据结构就是链表。为了能够方便地进行分配和回收，人们把空闲区域记录到链表里，这就是空闲链表 (free list)。

空闲链表

空闲链表里的节点主要是为了记录内存的开始位置和长度，如下图所示：



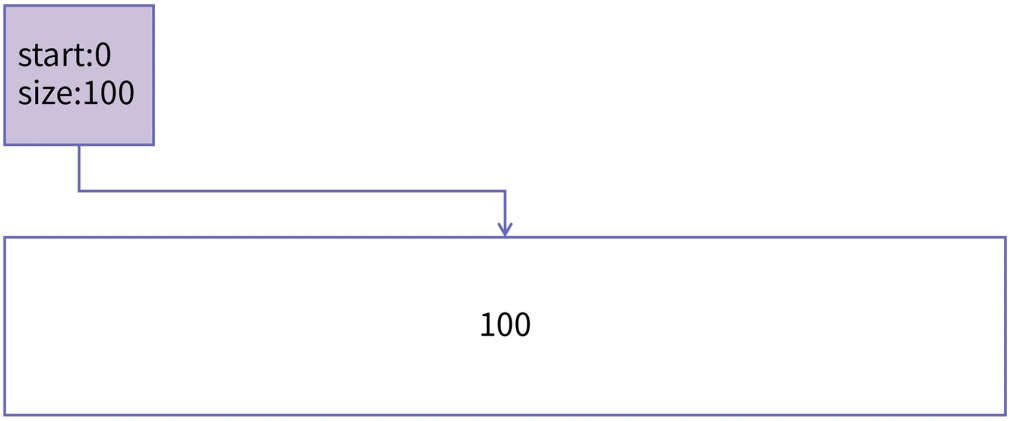
极客时间

图中展示了一个总长度为 100 的内存区域，已经分割成 16、16、20、16、16、16 六个小的内存块。其中着色部分，也就是第一、第三和第五块内存是已经分配出去的，正在使用的内存，而白色区域则是尚未分配的内存。图的上半部分代表空闲链表，每一块未分配的内存都会由一个空闲链表的节点进行管理。结点中记录了这块空闲内存区域的起始位置和长度。

当分配内存的请求到达以后，我们就通过遍历 free list 来查找可用的空闲内存区域，在找到合适的空闲区域以后，就将这一块区域从链表中摘下来。比如要请求的大小是 m，就将这个结点从链表中取下，把起始位置向后移动 m，大小也相应的减小 m。将修改后的结点重新挂到链表上。

在释放的时候，将这块区域按照起始起址的排序放回到链表里，并且检查它的前后是否有空闲区域，如果有就合并成一个更大的空闲区。

这种算法所使用的数据结构比较简单，算法也很直接，我们把这种算法称为简单算法(Naive Algorithm)。我们举个例子说明简单算法的运行过程，假如在算法开始时，内存的情况如下图所示：



极客时间

假设对内存的操作序列是这样的：

复制代码

```
void test() {
  void* p1 = malloc(16);
  void* p2 = malloc(16);
  void* p3 = malloc(20);

  free(p2);

  void* p4 = malloc(16);
  void* p5 = malloc(16);

  free(p4);
}
```

执行完 test 函数以后，内存的划分就会和这节课的第一幅图一样了。做为练习，请你自己画出每一个步骤 free list 和内存的变化情况，这里不再给出。

如果此时，又到达了一个内存分配请求，要申请一个大小为 20 的内存区域，虽然所有空闲区域的大小之和是 48，是超过 20 的，但是由于这三块空闲区域并不连续，所以，我们已经无法从这 100 字节的内存中再分配出一块 20 字节的内存区域了，相对于这次请求，这三块 16 字节的空闲区域就是**内存碎片**。这就是我们所介绍的简单算法的第一个缺陷：**会产生内存碎片**。

每一次分配内存时，我们都需要遍历 free list，最差情况下的时间复杂度显然是 O(n)。如果是多线程同时分配的话，free list 会被多线程并发访问，为了保护它，就必须使用各种同步机制，比如锁或者无锁的 concurrent linked list 等。可见上述算法的第二个缺陷是**分配效率一般，且多线程并发场景下性能还会恶化**。

为了改进以上两个问题，人们想了很多办法，我们举几个历史上曾经出现的改进方案。

其中一种方案是直接对简单算法进行优化。简单算法中找到第一个可用的区域就返回，这个策略被称为 First Fit，优化的具体做法是把它改成最佳匹配 (Best Fit)，改造后，它要找到能满足条件的最小的空闲区域才返回。

从直观上说，这种分配策略能尽可能地保留大块内存，避免它被快速地分割成小块内存，这就能更好地对抗内存碎片。严格的理论证明也证明了这一点。但是这种策略需要遍历整个链表，时间复杂度反而变差。

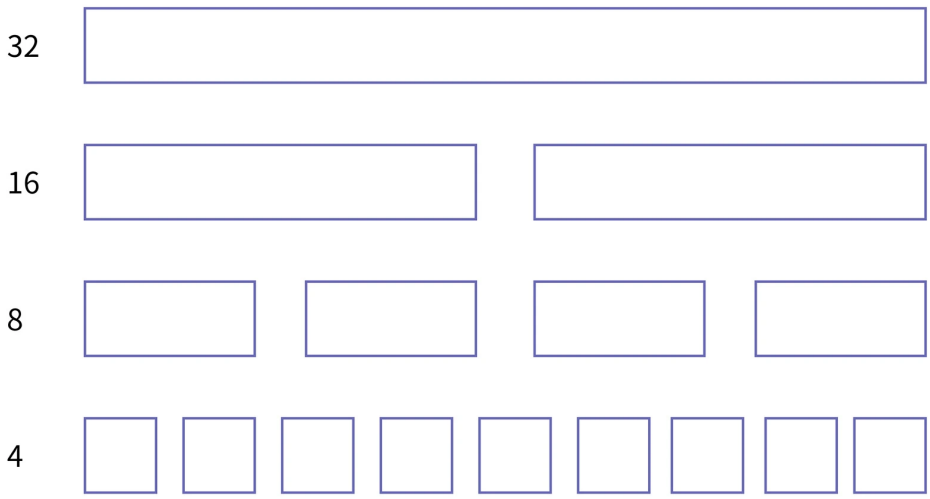
另一种方案是 Knuth 提出的 Next Fit 策略，即每次查找不必从头开始，而是从上一次查找的位置继续向后查找。实验也证明，这种策略会比从头开始的算法有更高的效率。但它依然不能解决内存碎片的问题。

还有一种改进方案，名字叫分桶式管理，**这种改进是一种相对均衡的做法，在对抗内存碎片和分配释放的时间复杂度两个方向都有改善**。这也是在现实中被使用的最广泛的一种方法。接下来，我们就重点分析分桶式管理算法。

分桶式内存管理

分桶式内存管理采用了多个链表，对于单个链表，它内部的所有结点所对应的内存区域的大小是相同的。换句话说，相同大小的区域会挂载到同一个链表上。

最常见的方式是以 4 字节为最小单位，把所有 4 字节的区域挂到同一个链表上，再把 8 字节的区域挂到一起，然后是 16 字节，32 字节，这样以 2 次幂向上增长。如下图所示：



极客时间

采用了新的数据结构以后，分配和回收的算法也相应地发生了变化。

首先，分配的时候，我们要只要找到能满足这一次分配请求的最小区域，然后去相应的链表里把整块区域都取下来。比如，分配一个 7 字节的内存块时，我们就可以从 8 字节大小的空闲链表里直接取出链表头上的那块区域，分配给应用程序。由于从链表头上删除元素的时间复杂度是 O(1)，所以我们分配内存的效率就大大提高了。

由于整个大块内存被提前分割成了整齐的小块（比如是以 4 字节对齐），所以整个区域里不存在块与块之间内存碎片。但是这种做法还是会产生区域内部的空间浪费，比如上面举的例子，当申请的内存大小是 7 时，按当前算法，只能分配给它大小为 8 的块，这就造成了一个字节的内部浪费，或者称之为内部碎片。

内部碎片带来的问题是内存使用率没有达到 100%，在最差情况下，可能只有 50%。但是内部碎片随着这一块区域的释放也就消失了，所以不会因为长时间运行而积累成严重的问题。

释放时，只需要把要释放的内存直接挂载到相应的链表里就可以了。 这个速度和分配是一样的，效率非常高。

分桶式内存管理比简单算法无论是在算法效率方面，还是在碎片控制方面都有很大的提升。但它的缺陷也很明显：区域内部的使用率不够高和动态扩展能力不够好。例如，4 字

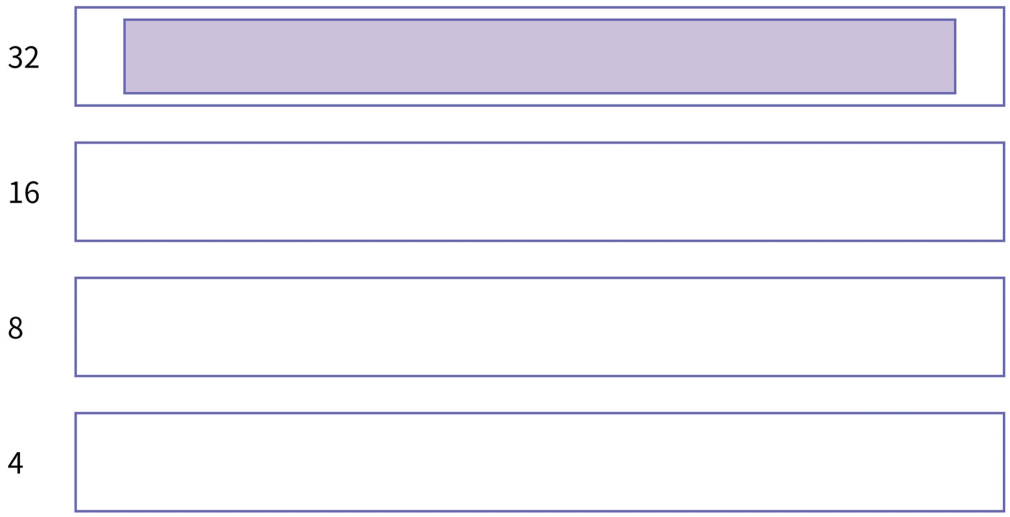
节的区域提前消耗完了，但 8 字节的空闲区域还有很多，此时就会面临两难选择，如果直接分配 8 字节的区域，则区域内部浪费就比较多，如果不分配，则明明还有空闲区域，却无法成功分配。

为了解决上述两个问题，人们在分桶的基础上继续改进，让内存可以根据需求动态地决定小的内存区域和大的内存区域的比例。这种设计的典型就是伙伴系统，我们一起来看下。

伙伴系统

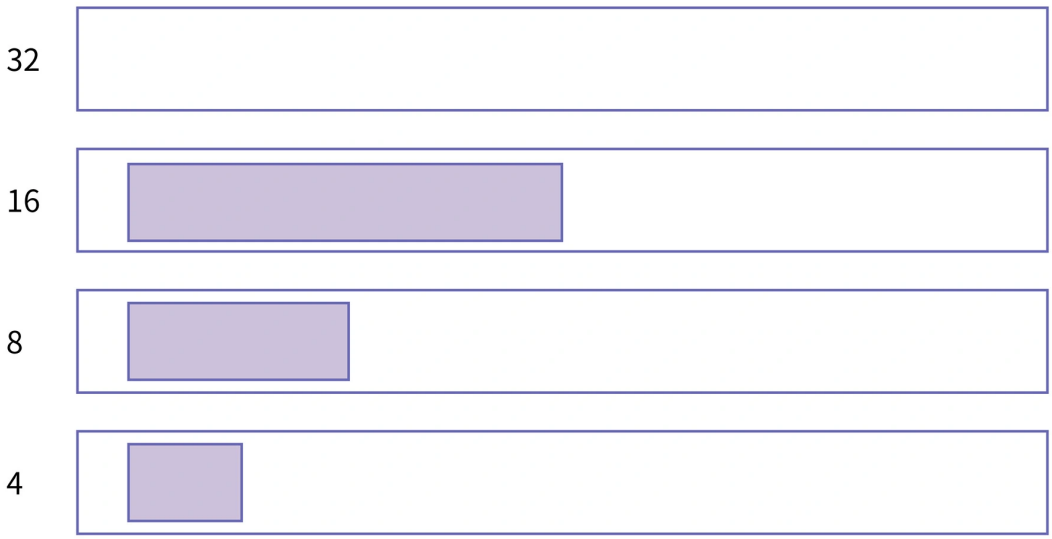
正如上面的例子所讲的，当系统中还有很多 8 字节的空闲块，而 4 字节的空闲块却已经耗尽，这时再有一个 4 字节的请求，则会出现 malloc 失败的情况。为了避免分配失败，我们其实还可以考虑将大块的内存做一次拆分。

如下图所示。分配一块 4 字节大小的空间，在 4 字节的 free list 上找不到空闲区域，系统就会往上找，假如 8 字节和 16 字节的 free list 中也没有空闲区域，就会一直向上找到 32 字节的 free list。



伙伴系统不会直接把 32 的空闲区域分配出去，因为这样做的话，会带来巨大的浪费。它会先把 32 字节分成两个 16 字节，把后边一个挂入到 16 字节的 free list 中。然后继续拆分前一半。前一半继续拆成两个 8 字节，再把后一半挂入到 8 字节的 free list，最后，把前

一半 8 字节拿去分配，当然这里也要继续拆分成两个 4 字节的空闲区域，其中一个用于本次 malloc 分配，另一个则挂入到 4 字节的 free list。分配后的内存的状态如下所示：



这种不断地把一块内存分割成更小的两块内存的做法，就是伙伴系统，这两块更小的内存就是伙伴。 它的好处是可以动态地根据分配请求将大的内存分割成小的内存。当释放内存时，如果系统发现与被释放的内存相邻的那个伙伴也是空闲的，就会把它们合并成一个更大的连续内存。通过这种拆分，系统就变得更加富有弹性。

malloc 的实现，在历史上先后共有几十种策略，这些策略往往就是上述三种算法的组合。具体到 glibc 中的 malloc 实现，它就采用了分桶的策略，但是它的每个桶里的内存不是固定大小的，而是采用了将 1 ~ 4 字节的块挂到第一个链表里，将 5 ~ 8 字节的块挂到第二个链表里，将 9~16 字节的块挂到第三个链表里，依次类推。

在单个链表内部则采用 naive 的分配方式，比如要分配 5 个字节的内存块，我们会先在 5 ~ 8 这个链表里查找，如果查找到的内存大小是 8 字节的，那就会将这个区域分割成 5 字节和 3 字节两个部分，其中 5 字节用于分配，剩余的 3 字节的空闲区域则会挂载到 1~4 这个链表里。

可见 malloc 的实现策略是比较灵活的，针对不同的场景，不同的分配策略的性能表现也是不一样的。很多公司的基础平台都选择自己实现内存池来提供 malloc 接口，这样可以更好地服务本公司的业务。最著名的例子就是 Google 公司实现的 Tcmalloc 库。

Tcmalloc 相比起其他的 malloc 实现，最大的改进是在多线程的情况下性能提升。我们知道，**在多线程并发地分配内存时，每次分配都要对 free list 进行加锁以避免并发程序带来的问题，这就容易形成性能瓶颈。**

为了解决这个问题，Tcmalloc 引入了线程本地缓存 (Thread Local Cache)，每个线程在分配内存的时候都先在自己的本地缓存中寻找，如果找到就结束，只有找不到的情况才会继续向全局管理器申请一块大的空闲区域，然后按照伙伴系统的方式继续添加到本地缓存中去。

在实际工作中，你可能会遇到这两个问题而束手无策：

> 第一个问题是，系统所提供的 malloc，其性能不足以支撑自己的业务，或者自己的业务在分配内存时有其特殊的规律，需要为它做专门的订制和优化；
>
> 第二个问题是，在 malloc 和 free 里做一些统计动作以排查问题，比如打印日志。

下面，我来带你自己动手实现内存管理库，让你更好解决内存问题的同时，还可以深入地理解内存管理的更多技术细节。

自己动手实现内存管理库

第 1 个问题的典型代表就是上面所提到的 Tcmalloc。我们这里举一个第二个问题的例子。比如，我曾经遇到过一个 double free 的错误，在申请了一段内存以后，经过复杂的逻辑，有两个指针指向了同一块内存，当我对两个指针都调用 free 方法的时候，错误就发生了，我把这个错误示例进行了简化，并把它的代码放在下面：

复制代码

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* p = (int*)malloc(sizeof(int));
    char* q = (char*)p;

    free(p);
    free(q);
}
```

很明显，第 8 行第 9 行释放的是同一块内存，运行这个例子，我们会看到进程 crash 了，并且系统提示为：

```
1 *** Error in `./dfree': double free or corruption (fasttop): 0x000000000196a01
2 Aborted
```

这就是 double free 的错误，也就是说一块内存被释放了两次。这个例子比较简单，但是在复杂逻辑中，我们往往很难判断多个指针是否指向相同的地址。

如果我们在所有调用 free 的地方增加日志，把要释放的指针记录下来，就会比较有助于分析和定位问题。幸运的是，我们确实有这种手段，那就是 Linux 的 preload 机制。

在第 8 节课，我们深入地学习了 Loader 的原理，我们知道对于未定义的引用，动态链接器要先进行解析，它会先搜索 LD_PRELOAD 目录下的动态库，然后再搜索其他的库，所以我们就有办法对 malloc 和 free 函数进行替换。比如自己提供 free 的实现：

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <dlfcn.h>
5
6  void free(void *ptr) {
7      void(*freep)() = NULL;
8
9      printf("ready to do free: %p\n", ptr);
10     freep = dlsym(RTLD_NEXT, "free");
11     freep(ptr);
12     printf("free done: %p\n", ptr);
13 }
```

我们先来分析一下上面的代码，我要提醒一下你，第 1 行的 _GNU_SOURCE 是一定要添加的，因为第 10 行的 RTLD_NEXT 宏依赖于这个宏。

接着，我们定义了一个函数指针（第 7 行），它可以指向真正的 free 的实现。然后分别在调用 free 前和调用 free 以后打印一次要释放的指针（第 9 行和第 12 行），再通过

dlsym 打开了 glibc 中的 free 方法（第 10 行），dlsym 的作用是通过符号名称找到符号对应的地址。

你还要注意的是，第 10 行使用 RTLD_NEXT 就是告诉 ld-linux.so 不要在当前文件中找 free 这个符号，而是要按照动态库的搜索顺序找到下一个动态库，并在它里面寻找 free 函数，实际上，这里找到的就是 glibc 里的 free 函数了。

分析完这个代码以后，我们发现这里自己定义的 free 方法不过是 glibc 里的 free 方法的一个包装 (wrapper)。接着，我们使用以下命令编译 myfree 库，并设置 preload 再执行 dfree 用例：

复制代码

```
$ gcc -shared -fpic -o myfree.so myfree.c -ldl
$ LD_PRELOAD="./myfree.so" ./dfree
ready to do free: 0x1b44010
free done: 0x1b44010
ready to do free: 0x1b44010
*** Error in `./dfree': double free or corruption (fasttop): 0x0000000001b4401
Aborted
```

运行上面的代码，你可以看到，我们新添加的日志已经可以正确输出了。通过这种方式，我们就重载了 free 方法。如果你特别感兴趣的话，你可以自己重新实现一个 malloc 的例子，我的代码仓可以给你作为参考。

总结

我们这节课深入地介绍了 malloc 方法的实现原理，并以一个例子来说明如何设计自己的动态内存管理器。

通过这节课的学习，我们了解到 sbrk 和 mmap 是操作系统提供的系统调用，系统调用性能不够且不能对分配的内存进行有效管理，所以必须有人在用户态将大块的内存分割成小块，然后进行更精细的分配和回收，这个工作通常情况下是由 glibc 承担的。

glibc 所提供的 malloc 在管理内存时，采用了空闲链表 (free list) 的方式。对 free list 的组织有简单算法、分桶管理和伙伴系统等不同的策略。

我们评价一个管理内存算法的因素主要有以下三个维度：

1. **数据结构如何设计，是否存在内存碎片**；
2. **分配的效率**；
3. **释放的效率**。

我把三种算法按照上面三个维度总结成以下表格，你可以参考一下：

	简单算法	分桶	伙伴系统
碎片	会产生块间碎片，块内无碎片	块间无碎片，但会有块内碎片。块内浪费有可能很严重	可以动态割内存区域，块内无碎片，块间碎片不严重
分配	分为First Fit/Next Fit/Best Fit等查询方法，最差情况下是O(n)	效率高，时间复杂度是O(1)	最优情况下是O(1)，最差情况下是O(lg n)
释放	要在链表中查询它的前一个区域和后一个区域是否是空闲区域，如果是则合并成一个更大的区域	直接还到相应大小的链里，时间复杂度是O(1)	要判断伙伴是否空闲，如果空闲则合并。最好情况是O(1)，最差情况是O(lg n)

极客时间

由于动态链接器只识别第一次遇到的符号，所以我们就有机会通过重写 glibc 中的方法来实现自己的内存管理库。这主要是为了提升性能，或者是为了排查错误。我们以 double free 为例介绍了典型的内存错误，并通过覆写 free 函数来解决它。这展示了一个完整的设计内存管理库的过程。

思考题

我们在实战环节介绍了如何在 free 方法里增加 log 以统计哪些内存曾经释放过，以便于排查哪些内存被重复释放了。

如果我们申请了一块内存，但是忘记释放了，这种情况就是内存泄露。请自己设计一个方案，统计内存泄露的情况。只要描述做法即可，如果你特别感兴趣的话，可以在我们这节课的代码里自己添加实现。欢迎你在留言区分享你的想法和收获，我在留言区等你。

吊打面试官

- 请简述mmap和malloc的区别与联系。

先说区别，首先，它们的实现机制不同，mmap是操作系统提供的系统调用，而malloc则是glibc提供的分配内存的接口；其次，它们的作用不同，malloc主要是为了分配堆内存，但是mmap除了可以用于分配内存，还可以用于加载动态链接库，进行驱动文件映射以加速IO，还可以用于创建共享内存进行进程间通信。

它们的联系是malloc和mmap都有分配堆内存的能力，然后malloc在向操作系统申请大块内存的时候还是要依赖于mmap的。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 08 | 动态链接（下）：延迟绑定与动态链接器是什么？

下一篇 10 | 页中断：fork、mmap背后的保护神

精选留言 (5)

写留言

qinsi
2021-11-12

看完感觉自己可以实现个valgrind了，不过又觉得少了亿点点细节。于是查了下valgrind的实现，发现只有用到了preload这点是一样的。剩下的部分valgrind相当于实现了一个虚拟机，将机器指令转成虚拟机IR，插桩，再通过JIT生成机器指令执行。这样看来Java字节码增强就是个弟弟阿...

...

展开

作者回复: 厉害，你调查得很深入了。

2

2021-11-18

老师，你好，请教个问题，之前函数重载这块使用比较多的是通过编译器wrap方式重载，但这块只能对可执行文件进行重载，不能实现glibc重载，这里所说的LD_PRELOAD是不是也只是针对可执行文件进行重载不能实现像动态库重载，如果是某个glibc中的函数使用malloc，这种方案是不是不能进行重载的；

展开

共 1 条评论 >

流浪地球

2021-11-17

请问老师，tcmalloc的线程本地缓存会不会导致相同的一份代码产生的进程，分配的虚拟内存空间会大一些呢？看描述像是增加预分配了一些空间

展开

慢动作

2021-11-12

error里也有地址，这额外日志没有给出额外的信息？

大豆

2021-11-12

老师，我有个疑问，通过mmap分配的内存是在进程的映射区还是堆中，还是都有？

> 作者回复: 在进程的映射区中。不用太在意这个概念的区分，你要在意的是，mmap出来的内存区域被拿去做什么了。它可以用于malloc分配堆内存，也可以用于协程的栈内存。所以一块内存区域到底是什么，取决于你怎么用他，而不是它自己的地址在哪里。换句话说，根据地址划分的区域不是绝对的，根据它的作用，内存区域的性质也是可以发生变化的。