练习 10

您熟悉 C 标准库中的 malloc()/free() 函数系列。这些函数以大块的形式从操作系统获取内存，在请求内存时将其分割成更小的区域并重新组织

释放内存时将其分成大的可用块。标准库必须管理一个数据结构来做到这一点，它
必须从它用来满足内存请求的同一内存区域构建该结构。我们要试试
这是我们自己。

在文件 myMalloc.h 和 myMalloc.c 中，我为您提供了我们自己的 malloc()、free()、calloc() 的部分实现
和重新分配（）。此实现的大部分已为您完成。它不像真正的堆分配子系统那么聪明
会的，但这足以让我们看看这种东西是如何工作的，以及为什么动态内存会出错
分配可能特别难以调试。你只需要完成两个功能的实现，
getBlock()，它从空闲列表中选择并删除一个节点，returnBlock() 将一个节点放回空闲列表
列表，可能将其与相邻块合并。

我们的实现将空闲内存组织为一个大的链表，将所有未使用的块链接在一起
内存并按内存地址对它们进行排序（因此，内存地址较低的块将在列表中靠前）。
每个空闲块中的部分内存用于构建此列表。每个空闲块的前几个字节用作
孔结构的实例。下面显示了 Hole 是如何定义的。它包含一个大小字段，其中包含大小
空洞和指向空闲列表中下一个节点的下一个指针。

```
typedef struct HoleStruct {
    size_t 尺寸；
    字符内存[0]；
    结构孔结构*下一个；
} 洞；
```

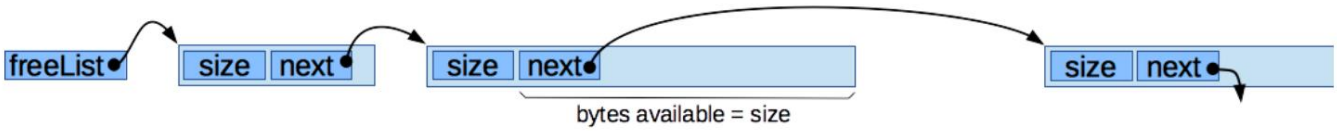
我是洞的大小，以字节为单位。

我不占用结构中的任何空间。

我是指向下一洞起点的指针。

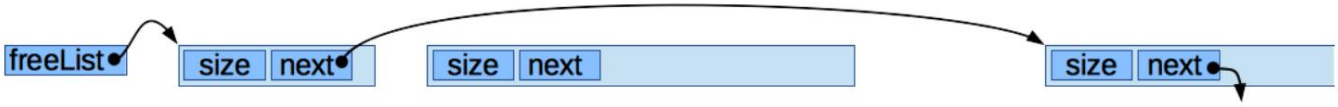
这个结构在 EOS Linux 机器上应该占用 16 个字节。一个空闲的内存块可能比那个大。那是
好的。我们使用块的第一部分作为 Hole 的实例，块的其余部分未使用，直到它通过
调用 malloc。

size 衡量空洞中的字节数，计算下一个指针的大小而不是 size 的大小
场地。



当我们的 malloc() 实现需要分配一个块时,它会调用 getBlock() 函数。这个函数的工作是遍历空闲块的链表并取消链接并返回第一个大到足以满足

要求。



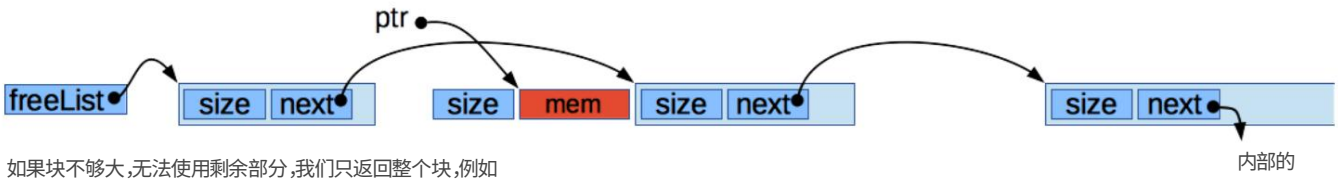
如果没有足够大的块（最初根本没有块）,getBlock() 应该返回 NULL。什么时候

getBlock()返回NULL,我们的malloc()就会去操作系统申请更多的内存。它将使用其中的一部分

内存以满足请求并使用 returnBlock() 将剩余内存放在 freeList 上。

getBlock() 返回的块可能大于当前请求。如果有足够的额外空间,它可能是

可以只放弃块的第一部分并将剩余部分作为新节点重新插入到 freeList 中。



如果块不够大,无法使用剩余部分,我们只返回整个块,例如

碎片化。如果可能,malloc() 函数负责拆分块,并且它使用 returnBlock() 将任何

空闲列表中的额外内存。

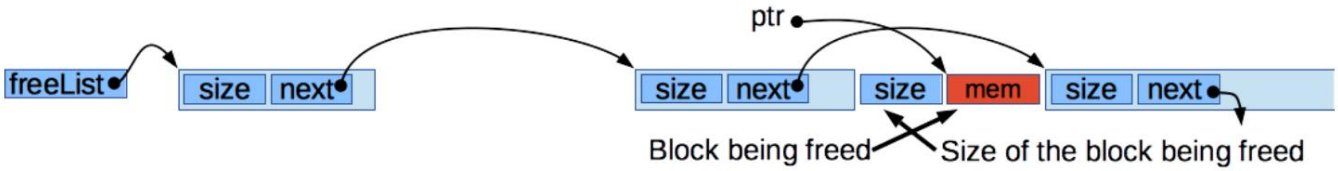
当 malloc() 给出一个指向内存块的指针时,它会返回内存块中大小字段之后的地址

孔结构（上图中的橙色块）。大小字段的内容留在内存中,就在之前

地址 malloc() 返回。这对于最终释放块的时间很重要。 free() 函数可以备份

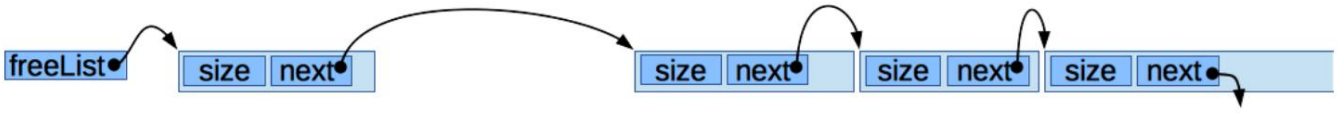
在内存中并查看此大小字段以确定有多少内存返回到空闲列表。然后它调用 returnBlock() 将释放的内存重新链接回空闲列表中的正确位置（下图中的块是

应该是与上面分配的不同的块）。



returnBlock() 函数有两个任务。它必须将释放的块重新链接回空闲列表中的正确位置,使用

按内存地址排序的块。我已经为您编写了这部分功能。

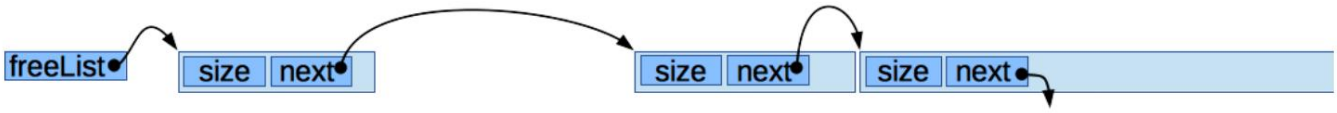


然后,如果释放块和它在 freeList 上的前任和/或后继块之间没有间隙,

returnBlock() 需要将块合并为一个更大的连续块。我已经为

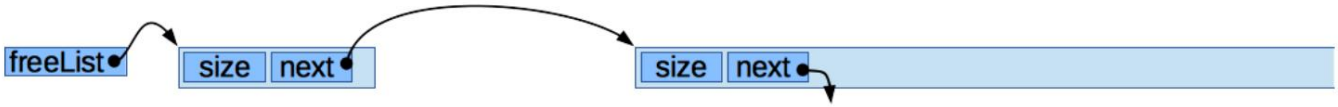
你。在一个块被链接回空闲列表后,我检查释放块和它的块之间是否没有间隙

接班人。如果释放的块与其后继块相邻,我将它们合并为一个大块。



同样的事情也需要发生在前任身上。如果有前导块并且它和

释放的块,它们应该合并成一个大块。您可以添加代码来执行此操作。



请注意,一旦您开始使用自己的 malloc()/free(),您的程序就会将它用于所有事情,包括

您未编写的代码的内存请求(例如,如果 printf() 需要分配一些内存)。你可能想要调试

此代码使用 gdb,而不是依赖于代码中的打印输出。如果 malloc() 是

坏了,即使它坏了,你也不希望它在调试 malloc() 时尝试分配内存

执行。

为了帮助您试用您的替代堆分配子系统,我编写了一个测试程序,memTest.c。这个程序

使用 reportFreeList() 函数,这是一个调试函数,可以让我们看到空闲列表上的所有漏洞,以确保

malloc 实现工作正常。该程序首先尝试一些简单的分配/释放操作,在这些操作中很容易预测空闲列表应该是什么样子。然后,它会执行一系列随机的

malloc()/free() 请求来尝试

找到其他不太明显的错误。您应该能够使用您的 malloc 实现构建此测试驱动程序

命令如:

```
$ gcc -Wall -std=c99 -g -D_DEFAULT_SOURCE memTest.c myMalloc.c -o memTest
```

运行此测试时,您应该会得到如下所示的输出。第一个free list报告为空;不应该有

通过第一个测试 (babyTest) 中途空闲列表中的任何块。在 babyTest 结束时,所有的空闲内存

应该在空闲列表的一个块中,恰好是一页的大小减去块的大小字段的大小(8 字节)。

```
$ ./内存测试
------ babyTest 1 --------
------ babyTest 2 --------
0x600000000000 : FF8
-------- toddlerTest 1 - ------
0x6000000000080 :F78
-------- toddlerTest 2 ------
0x600000000028 :34
0x600000000080 :F78
-------- toddlerTest 3 ------ --
0x600000000000: 5C
```

0x600000000080 : F78
-------- toddlerTest 4 --------
0x600000000000 : FF8
-------- randomTest --------
0x600000000000 : 188FF8

在调试实现时,请查看 memTest.c 程序中的测试。它们相当简单，

了解每个测试在做什么将帮助您找出问题所在。你甚至可以创建一些

自己做简单的测试用例。用一个很小的测试用例调试可能比尝试调试

随机测试在最后。

一旦您的替换 malloc 开始工作,只需将 myMalloc.c 文件提交给名为 EX10 的 Gradescope 作业。

这应该是您需要更改的唯一文件。