

malloc()和free()的原理

malloc()和free()的基本概念以及基本用法：

1、函数原型及说明：

void *malloc(long NumBytes): 该函数分配了NumBytes个字节，并返回了指向这块内存的指针。如果分配失败，则返回一个空指针（NULL）。

void free(void *FirstByte): 该函数是将之前用malloc分配的空间还给程序或者是操作系统，也就是释放了这块内存，让它重新得到自由。

2、函数的用法：

程序代码：

```
// Code...

char *Ptr = NULL;

Ptr = (char *)malloc(100 * sizeof(char)); //需要强制类型转换，malloc返回的是void类型，在c与c++中可以强制转化为任何类型。

if (NULL == Ptr)

{

    exit (1);

}

gets(Ptr);

// code...

free(Ptr);

Ptr = NULL;

// code...
```

3、关于函数使用需要注意的一些地方：

A、申请了内存空间后，必须检查是否分配成功。

B、当不需要再使用申请的内存时，记得释放；释放后应该把指向这块内存的指针指向NULL，防止程序后面不小心使用了它。

C、这两个函数应该是配对。如果申请后不释放就是内存泄露；如果无故释放那就是什么也没有做。释放只能一次，如果释放两次及两次以上会

出现错误（释放空指针例外，释放空指针其实也等于啥也没做，所以释放空指针释放多少次都没有问题）。

D、虽然malloc()函数的类型是(void *),任何类型的指针都可以转换成(void *),但是最好还是在前面进行强制类型转换。

二、malloc()到底从哪里得来了内存空间：

1、malloc()到底从哪里得到了内存空间？答案是从堆里面获得空间。也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

2、什么是堆：堆是大家共有的空间，分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间。堆在操作系统对进程初始化的时候分配，运行过程中也可以向系统要额外的堆，但是记得用完了要还给操作系统，要不然就是内存泄漏。

公告

昵称：only_eVonne
园龄：3年10个月
粉丝：40
关注：2
[+加关注](#)

< 2012年6月 >						
日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
[更多链接](#)

随笔分类

Android(1)
Linux编程(70)
Nginx源码(8)
QT编程(4)
STL学习(1)
程序人生(4)
多媒体音视频(1)
数据结构(11)
网络编程(12)
语言之美(12)

随笔档案

2014年11月 (1)
2014年8月 (1)
2014年7月 (1)
2014年6月 (1)
2013年11月 (3)
2013年7月 (7)
2013年5月 (1)
2013年4月 (17)
2013年3月 (13)
2013年2月 (7)
2012年6月 (1)
2012年5月 (4)
2012年4月 (3)
2012年3月 (4)
2012年2月 (7)

什么是栈：栈是线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈互相独立。每个函数都有自己的栈，栈被用来在函数之间传递参数。操作系统在切换线程的时候会自动的切换栈，就是切换SS/ESP寄存器。栈空间不需要在高级语言里面显式的分配和释放。

通过上面对概念的描述，可以知道：

栈是由编译器自动分配释放，存放函数的参数值、局部变量的值等。操作方式类似于数据结构中的栈。

堆一般由程序员分配释放，若不释放，程序结束时可能由OS回收。注意这里说是可能，并非一定。所以我想再强调一次，记得要释放！

注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。（这点我上面稍微提过）

所以，举个例子，如果你在函数上面定义了一个指针变量，然后在这个函数里申请了一块内存让指针指向它。实际上，这个指针的地址是在栈上，但是它所指向的内容却是在堆上面的！这一点要注意！所以，再想想，在一个函数里申请了空间后，比如说下面这个函数：

程序代码：

```
// code...

void Function(void)

{

    char *p = (char *)malloc(100 * sizeof(char));

}
```

就这个例子，千万不要认为函数返回，函数所在的栈被销毁指针也跟着销毁，申请的内存也就一样跟着销毁了！这绝对是错误的！因为申请的内存存在堆上，而函数所在的栈被销毁跟堆完全没有啥关系。所以，还是那句话：记得释放！

3、free()到底释放了什么

这个问题比较简单，其实我是想和第二大部分的题目相呼应而已！哈哈！free()释放的是指针指向的内存！注意！释放的是内存，不是指针！这点非常非常重要！指针是一个变量，只有程序结束时才被销毁。释放了内存空间后，原来指向这块空间的指针还是存在！只不过现在指针指向的内容的垃圾，是未定义的，所以说是垃圾。因此，前面我已经说过了，释放内存后把指针指向NULL，防止指针在后面不小心又被解引用了。非常重要啊这一点。

三、malloc()以及free()的机制：

这个部分我今天才有了新的认识！而且是转折性的认识！所以，这部分可能会有更多一些认识上的错误！不对的地方请大家帮忙指出！

事实上，仔细看一下free()的函数原型，也许也会发现似乎很神奇，free()函数非常简单，只有一个参数，只要把指向申请空间的指针传递

给free()中的参数就可以完成释放工作！这里要追踪到malloc()的申请问题了。申请的时候实际上占用的内存要比申请的大。因为超出的空间是用来记录对这块内存的管理信息。先看一下在《UNIX环境高级编程》中第七章的一段话：

大多数实现所分配的存储空间比所要求的要稍大一些，额外的空间用来记录管理信息——分配块的长度，指向下一个分配块的指针等等。这就意味着如果写过一个已分配区的尾端，则会改写后一块的管理信息。这种类型的错误是灾难性的，但是因为这种错误不会很快就暴露出来，所以也就很难发现。将指向分配块的指针向后移动也可能会改写本块的管理信息。

以上这段话已经给了我们一些信息了。malloc()申请的空间实际我觉得就是分了两个不同性质的空间。一个就是用来记录管理信息的空间，另外一个就是可用空间了。而用来记录管理信息的实际上是一个结构体。在C语言中，用结构体来记录同一个对象的不同信息是天经地义的事！下面看看这个结构体的原型：

程序代码：

```
struct mem_control_block {

    int is_available;    //这是一个标记？

    int size;            //这是实际空间的大小

};
```

对于size,这个是实际空间大小。这里其实我有个疑问，is_available是否是一个标记？因为我看了free()的源代码之后对这个变量感觉有点纳闷（源代码在下面分析）。这里还请大家指出！

所以，free()就是根据这个结构体的信息来释放malloc()申请的空间！而结构体的两个成员的大小我想应该是操作系统的事了。但是这里有一个问题，malloc()申请空间后返回一个指针应该是指向第二种空间，也就是可用空间！不然，如果指向管理信息空间的话，写入的内容和结构体的类型有可能不一致，或者会把管理信息屏蔽掉，那就没法释放内存空间了，所以会发生错误！（感觉自己这里说的是废话）

好了！下面看看free()的源代码，我自己分析了一下，觉得比起malloc()的源代码倒是容易简单很多。只是有个疑问，下面指出！

2012年1月 (7)
2011年12月 (11)
2011年11月 (5)
2011年10月 (13)
2011年9月 (13)
2011年8月 (4)
2011年7月 (5)

最新评论

1. Re:IP头，TCP头，UDP头，MAC帧头定义
Mark
--networkcomms通信框架
2. Re:TIME_WAIT状态的作用
Nice, Thank you for your sharing.Easy to understand.
--宁采臣

阅读排行榜

1. linux下使用tar命令(101689)
2. 什么是A类、B类、C类地址？(16690)
3. 冯诺依曼体系结构与哈佛体系结构的区别(10225)
4. TCL脚本语言基础介绍(9578)
5. 改变linux终端颜色(9361)

评论排行榜

1. popen函数的实现(3)
2. TIME_WAIT状态的作用(1)
3. IP头，TCP头，UDP头，MAC帧头定义(1)

推荐排行榜

1. linux下使用tar命令(4)
2. 改变linux终端颜色(2)
3. TCL脚本语言基础介绍(2)
4. popen函数的实现(1)
5. fork与vfork的区别(1)

程序代码：

// code...

```
void free(void *ptr)

{

    struct mem_control_block *free;

    free = ptr - sizeof(struct mem_control_block); //把指向可用空间的指针倒回去， 让它指向管理信息

    free->is_available = 1;

    return;

}
```

看一下函数第二句，这句非常重要和关键。其实这句就是把指向可用空间的指针倒回去，让它指向管理信息的那块空间，因为这里是在值上减去了一个结构体的大小！后面那一句free->is_available = 1;我有点纳闷！我的想法是：这里is_available应该只是一个标记而已！因为从这个变量的名称上来看，is_available 翻译过来就是“是可以用”。不要说我土！我觉得变量名字可以反映一个变量的作用，特别是严谨的代码。这是源代码，所以我觉得绝对是严谨的！！这个变量的值是1，表明是可以用的空间！只是这里我想了想，如果把它改为0或者是其他值不知道会发生什么事？！但是有一点我可以肯定，就是释放绝对不会那么顺利进行！因为这是一个标记！

当然，这里可能还是有人会有疑问，为什么这样就可以释放呢？？我刚才也有这个疑问。后来我想到，释放是操作系统的事，那么就free()这个源代码来看，什么也没有释放，对吧？但是它确实是确定了管理信息的那块内存的内容。所以，free()只是记录了一些信息，然后告诉操作系统那块内存可以去释放，具体怎么告诉操作系统的我不清楚，但我觉得这个已经超出了我这篇文章的讨论范围了。

那么，我之前有个错误的认识，就是认为指向那块内存的指针不管移到那块内存中的哪个位置都可以释放那块内存！但是，这是大错特错！释放是不可以释放一部分的！首先这点应该要明白。而且，从free()的源代码看，ptr只能指向可用空间的首地址，不然，减去结构体大小之后一定不是指向管理信息空间的首地址。所以，要确保指针指向可用空间的首地址！不信吗？自己可以写一个程序然后移动指向可用空间的指针，看程序会有会崩！

最后可能想到malloc()的源代码看看malloc()到底是怎么分配空间的，这里面涉及到很多其他方面的知识！有兴趣的朋友可以自己去下载源代码去看看。

=====

C语言的malloc分配的的内存大小

没读过malloc()的源码，所以这里纯粹是"理论研究"。

malloc()在运行期动态分配分配内存,free()释放由其分配的内存。malloc()在分配用户传入的大小的时候，还分配的一个相关的用于管理的额外内存，不过，用户是看不到的。所以，

实际申请空间的大小 = 管理空间 + 用户空间

那么，这个管理内存放在什么位置呢,它要让free()函数能够找到，这样才能知道有多少内存要释放，所以一种可能的方案是在分配内存的初始部分用若干个字节来存储分配的内存的大小。这里要注意一个问题，就是，在malloc()将这个分配的空间返回给某个指针后，这个指针的使用与其它指针应该没有差别的，所以，管理空间应该在这个指针指向的空间之外，但又要free()从这个指针可以找到管理信息，所以，这个管理空间的大小放在指针指向的相反方向。故malloc()的具体操作应该就是分配一块内存，在前面若干字节中写入管理信息，然后返回管理信息所占字节之后的地址指针。

=====

malloc（）工作机制

malloc函数的实质体现在它有一个将可用的内存块连接为一个长长的列表的所谓空闲链表。调用malloc函数时，它沿连接表寻找一个大到足以满足用户请求所需要的内存块。然后，将该内存块一分为二（一块的大小与用户请求的大小相等，另一块的大小就是剩下的字节）。接下来，将分配给用户的那块内存传给用户，并将剩下的那块（如果有的话）返回到连接表上。调用free函数时，它将用户释放的内存块连接到空闲链上。到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段，那么空闲链上可能没有可以满足用户要求的片段了。于是，malloc函数请求延时，并开始在空闲链上翻箱倒柜地检查各内存片段，对它们进行整理，将相邻的小空闲块合并成较大的内存块。

malloc（）在操作系统中的实现

在 C程序中，多次使用malloc () 和 free()。不过，您可能没有用一些时间去思考它们在您的操作系统中是如何实现的。本节将向您展示 malloc 和 free 的一个最简化实现的代码，来帮助说明管理内存时都涉及到了哪些事情。

在大部分操作系统中，内存分配由以下两个简单的函数来处理：

void *malloc (long numbytes): 该函数负责分配 **numbytes** 大小的内存，并返回指向第一个字节的指针。

void free(void *firstbyte): 如果给定一个由先前的 **malloc** 返回的指针，那么该函数会将分配的空间归还给进程的“空闲空间”。

malloc_init 将是初始化内存分配程序的函数。它要完成以下三件事：将分配程序标识为已经初始化，找到系统中最后一个有效内存地址，然后建立起指向我们管理的内存的指针。这三个变量都是全局变量：

如前所述，被映射的内存的边界（最后一个有效地址）常被称为系统中断点或者 当前中断点。在很多 **UNIX?** 系统中，为了指出当前系统中断点，必须使用 **sbrk(0)** 函数。**sbrk** 根据参数中给出的字节数移动当前系统中断点，然后返回新的系统中断点。使用参数 **0** 只是返回当前中断点。这里是我们的 **malloc** 初始化代码，它将找到当前中断点并初始化我们的变量：

清单 2. 分配程序初始化函数

```
/* Include the sbrk function */

#include

void malloc_init()

{

/* grab the last valid address from the OS */

last_valid_address = sbrk(0);

/* we don't have any memory to manage yet, so

*just set the beginning to be last_valid_address

*/

managed_memory_start = last_valid_address;

/* Okay, we're initialized and ready to go */

has_initialized = 1;

}
```

现在，为了完全地管理内存，我们需要能够追踪要分配和回收哪些内存。在对内存块进行了 **free** 调用之后，我们需要做的是诸如将它们标记为未被使用的等事情，并且，在调用 **malloc** 时，我们要能够定位未被使用的内存块。因此， **malloc** 返回的每块内存的起始处首先要有这个结构：

//清单 3. 内存控制块结构定义

```
struct mem_control_block {

    int is_available;

    int size;

};
```

现在，您可能会认为当程序调用 **malloc** 时这会引发问题 —— 它们如何知道这个结构？答案是它们不必知道；在返回指针之前，我们会将其移动到这个结构之后，把它隐藏起来。这使得返回的指针指向没有用于任何其他用途的内存。那样，从调用程序的角度来看，它们所得到的全部是空闲的、开放的内存。然后，当通过 **free()** 将该指针传递回来时，我们只需要倒退几个内存字节就可以再次找到这个结构。

在讨论分配内存之前，我们将先讨论释放，因为它更简单。为了释放内存，我们必须要做的惟一一件事情就是，获得我们给出的指针，回退 **sizeof(struct mem_control_block)** 个字节，并将其标记为可用的。这里是对应的代码：

清单 4. 解除分配函数

```
void free(void *firstbyte) {

    struct mem_control_block *mcb;

/* Backup from the given pointer to find the

* mem_control_block

*/

    mcb = firstbyte - sizeof(struct mem_control_block);

/* Mark the block as being available */

    mcb->is_available = 1;

/* That's It! We're done. */
```

```
return;
```

```
}
```

如您所见，在这个分配程序中，内存的释放使用了一个非常简单的机制，在固定时间内完成内存释放。分配内存稍微困难一些。我们主要使用连接的指针遍历内存来寻找开放的内存块。这里是代码：

//清单 6. 主分配程序

```
void *malloc(long numbytes) {

    /* Holds where we are looking in memory */

    void *current_location;

    /* This is the same as current_location, but cast to a

    * memory_control_block

    */

    struct mem_control_block *current_location_mcb;

    /* This is the memory location we will return.  It will

    * be set to 0 until we find something suitable

    */

    void *memory_location;

    /* Initialize if we haven't already done so */

    if(! has_initialized) {

        malloc_init();

    }

    /* The memory we search for has to include the memory

    * control block, but the users of malloc don't need

    * to know this, so we'll just add it in for them.

    */

    numbytes = numbytes + sizeof(struct mem_control_block);

    /* Set memory_location to 0 until we find a suitable

    * location

    */

    memory_location = 0;

    /* Begin searching at the start of managed memory */

    current_location = managed_memory_start;

    /* Keep going until we have searched all allocated space */

    while(current_location != last_valid_address)

    {

        /* current_location and current_location_mcb point

        * to the same address.  However, current_location_mcb

        * is of the correct type, so we can use it as a struct.

        * current_location is a void pointer so we can use it

        * to calculate addresses.

        */

        current_location_mcb =

            (struct mem_control_block *)current_location;

        if(current_location_mcb->is_available)

        {

            if(current_location_mcb->size >= numbytes)

            {

                /* Woohoo!  We've found an open,
```

```

    * appropriately-size location.

    */

    /* It is no longer available */

    current_location_mcb->is_available = 0;

    /* We own it */

    memory_location = current_location;

    /* Leave the loop */

    break;

}

}

/* If we made it here, it's because the Current memory

* block not suitable; move to the next one

*/

current_location = current_location +

    current_location_mcb->size;

}

/* If we still don't have a valid location, we'll

* have to ask the operating system for more memory

*/

if(! memory_location)

{

    /* Move the program break numbytes further */

    sbrk(numbytes);

    /* The new memory will be where the last valid

    * address left off

    */

    memory_location = last_valid_address;

    /* We'll move the last valid address forward

    * numbytes

    */

    last_valid_address = last_valid_address + numbytes;

    /* We need to initialize the mem_control_block */

    current_location_mcb = memory_location;

    current_location_mcb->is_available = 0;

    current_location_mcb->size = numbytes;

}

/* Now, no matter what (well, except for error conditions),

* memory_location has the address of the memory, including

* the mem_control_block

*/

/* Move the pointer past the mem_control_block */

memory_location = memory_location + sizeof(struct mem_control_block);

/* Return the pointer */

return memory_location;

}

```

这就是我们的内存管理器。现在，我们只需要构建它，并在程序中使用它即可.多次调用**malloc**（）后空闲内存被切成很多的小内存片段，这就使得用户在申请内存使用时，由于找不到足够大的内存空间，**malloc**（）需要

进行内存整理，使得函数的性能越来越低。聪明的程序员通过总是分配大小为2的幂的内存块，而最大限度地降低潜在的malloc性能丧失。也就是说，所分配的内存块大小为4字节、8字节、16字节、18446744073709551616字节，等等。这样做最大限度地减少了进入空闲链的怪异片段（各种尺寸的小片段都有）的数量。尽管看起来这好像浪费了空间，但也容易看出浪费的空间永远不会超过50%

分类: [Linux编程](#)

绿色通道:

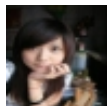
[好文要顶](#)

[关注我](#)

[收藏该文](#)

[与我联系](#)





only_eVonne

关注 - 2

粉丝 - 40

[+加关注](#)

1

0

(请您对文章做出评价)

« 上一篇: [\[转\]U-BOOT内存布局及启动过程浅析](#)
» 下一篇: [系统什么时候发送sigkill信号](#)
posted @ 2012-06-03 16:46 only_eVonne 阅读(1502) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问](#)网站首页。

【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库
融云，免费为你的App加入IM功能——让你的App“聊”起来！！
【活动】刮上代金券，使用再送100元大礼包！



最新IT新闻:

- “绿色面子工程”被烧焦 苹果的绿电梦想遭遇挫折
- 别再管什么用户体验了
- 苹果回应AW新系统心率监测bug 附解决方法
- 如何证明自己是一个高大上的赚钱的科技公司？ 答案：盖楼
- “铁路12306”客户端升级 老版本6月8日停用

» 更多新闻...



最新知识库文章:

- 新手学习编程的最佳方式是什么？
- 领域驱动设计系列（1）通过现实例子显示领域驱动设计的威力
- 操作系统课程是如何改变我的
- 微服务实战（一）：微服务架构的优势与不足
- 编写高质量的代码——从命名入手

» 更多知识库文章...