# 一、简介

malloc函数原型：void \*malloc(size\_t size);

申请size个字节的虚拟地址空间，并返回指向这块内存的指针；如果申请失败则返回一个空指针free函数原型：void free(void \*ptr);

释放ptr指向的虚拟地址空间。问题：为啥只要传递地址，而不用传递释放多大空间？（在后文可以找到答案）

使用需要注意的一些地方：

A．申请内存空间后，必须检查是否分配成功

B．当不需要再使用的内存时，记得释放；释放后应该指向这块内存的指针指向NULL，防止程序后面不小心使用它

C．这两个函数应该配对，防止内存泄露

D．虽然malloc函数的类型是（void \*），为了明确用途最好在前面强制类型转换。

# 二、uclibc库实现

主要讲述标准封装库uClibc-0.9.33.2里支持MMU的malloc-standard的malloc和free讲解，在配置里有如下选择

choice

prompt "Malloc Implementation"

default MALLOC if ! ARCH\_USE\_MMU

default MALLOC\_STANDARD if ARCH\_USE\_MMU

1、数据结构

在标准库里有个专门的malloc数据结构管理申请到的虚拟内存，具体如下：

struct malloc\_state {

使用fastbin的阈值 64字节

size\_t max\_fast; /\* low 2 bits used as flags \*/

/\* Fastbins 管理<64字节的malloc申请 \*/

mfastbinptr fastbins[NFASTBINS];

/\* 指向最顶端的chunk，当这个chunk空闲的size大于trim\_threshold就会brk系统调用释放内存 \*/

mchunkptr top;

/\* The remainder from the most recent split of a small request \*/

mchunkptr last\_remainder;

/\* 该数组管理64字节以上通过brk产生的虚拟内存，mmap则一对一申请释放\*/

mchunkptr bins[NBINS \* 2];

/\* Bitmap of bins. Trailing zero map handles cases of largest binned size \*/

unsigned int binmap[BINMAPSIZE+1];

/\* Tunable parameters \*/

unsigned long trim\_threshold; //触发brk系统调用释放内存 = 256K

size\_t top\_pad; //最顶端的brk是否空闲，空闲就brk系统调用释放内存

size\_t mmap\_threshold; //触发mmap系统调用申请的阈值 = 256K

/\* Memory map support \*/

int n\_mmaps;

int n\_mmaps\_max;

int max\_n\_mmaps;

/\* Cache malloc\_getpagesize \*/

unsigned int pagesize;

/\* Track properties of MORECORE \*/

unsigned int morecore\_properties;

/\* Statistics \*/

size\_t mmapped\_mem;

size\_t sbrked\_mem;

size\_t max\_sbrked\_mem;

size\_t max\_mmapped\_mem;

size\_t max\_total\_mem;

};

使用该结构体主要解决如下问题：

1. 空闲块组织：我们如何记录空闲块

B．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

C．分割：在我们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剩余的部分

D．合并：我们如何处理一个刚刚被释放的块

2、malloc

malloc申请时在头上会多申请2个字用于管理该申请块，其结构如下：

An allocated chunk looks like this:

chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Size of previous chunk, if allocated | |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Size of chunk, in bytes |P|

mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| User data starts here... .

. .

. (malloc\_usable\_space() bytes) .

. |

nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Size of chunk |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

第一个字指向前一个chunk的大小，第二个字指向当前chunk的大小，P表示是否使用，mem即使malloc返回的获取到地址。因此，malloc(bytes)申请bytes个字节时，实际申请的nb=bytes + 2\*sizeof(size\_t)

根据源码，申请的情况如下：

A．当nb <= max\_fast (64字节)

直接从fastbins数组中分配，直接返回。如果没有，则从bins里分配

B．当nb > max\_fast

从bins数组里查找获取。

如果获取到一个比较大的chunk，则先进行分割，剩余的重新组织管理，直接返回给用户。

如果没有找到大的nb，则会努力对bins的空闲进行合并，如果还是没有则会通过系统调用brk或mmap从kernel申请。

C．系统调用brk或mmap

malloc管理的fastbins和bins里没有找到需要的虚拟内存时，只能通过系统调用从内核获取。

当nb > mmap\_threshold=256K时，使用mmap的方式，set\_head(p, size|IS\_MMAPPED);会将上面第1个字的bit[1]设置成IS\_MMAPPED（释放时判断）

当nb < mmap\_threshold=256K时，使用brk的方式，同时调整malloc管理的相关信息。

两者细节见《第三章》

3、free

free后chunk的布局结构如下：

Free chunks are stored in circular doubly-linked lists, and look like this:

chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Size of previous chunk |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

`head:' | Size of chunk, in bytes |P|

mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Forward pointer to next chunk in list |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Back pointer to previous chunk in list |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Unused space (may be 0 bytes long) .

. .

. |

nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

`foot:' | Size of chunk, in bytes |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The P (PREV\_INUSE) bit

问题：为啥只要传递地址，而不用传递释放多大空间？

答：地址前面预留了2个字，其中一个字就保存了申请到的内存块大小，所以只需要传递地址就行。

根据源码，释放的情况如下：

A．当size <= max\_fast时

直接释放到fastbins数组里

B．当size > max\_fast && 不是mmap时

先检查前后项是否属于空闲，如果是则可能进行合并，合并规则如下：

前面的块和后面的块都已经分配的，则只释放自己

前面的块是已分配的，后面的块空闲，则和后面的空闲块合并

前面的块是空闲，后面的块是分配的，则和前面的块合并

前后块都是空闲的，则进行前后合并

进行上面的合并操作后，判断最顶端top的空闲chunk是否超过trim\_threshold，如果超过就进行brk系统调用释放的内存。

注意点：

结合brk系统调用服务程序（见下面章节），brk是线性向上增长，如果最顶端的brk在使用而brk下面的其他地址释放了，即使超过trim\_threshold这时仍无法进行brk系统调用释放，这时就会导致这部分物理内存占着，从而引发内存碎片。

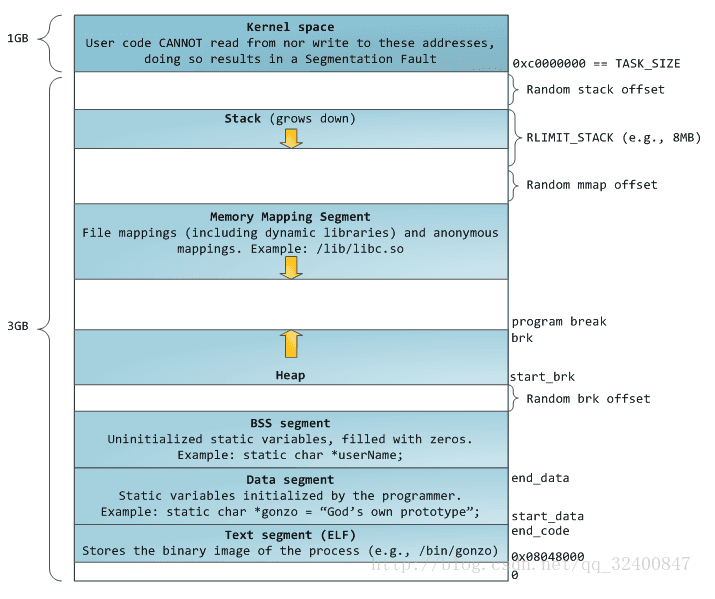
所以一个简单原则：通过brk申请的内存malloc后尽快free（即小内存申请<256K），对应不free的申请使用mmap的方式。

C．如果是mmap出来的内存

直接调用系统调用munmap进行内存释放

# 三、brk和mmap系统调用

在user/kernel为3G/1G比例的情况下，进程的内存分布图（摘自网络）如下：



这是灵活的内存增长方式，栈stack和mmap区域向下增长，堆heap向上增长。

brk和mmap的系统调用分别对应heap区域和memory mmaping segment区域。通过cat /proc/pid/maps可以看到该进程的详细内容：

00008000-00013000 r-xp 00000000 1f:05 2722616 /usr/bin/sysapp

0001a000-0001b000 rw-p 0000a000 1f:05 2722616 /usr/bin/sysapp

00a38000-00a39000 rw-p 00000000 00:00 0 [heap]

b6cac000-b6caf000 r-xp 00000000 1f:04 70 /lib/libdl.so.0

b6caf000-b6cb6000 ---p 00000000 00:00 0

b6cb6000-b6cb7000 rw-p 00002000 1f:04 70 /lib/libdl.so.0

b6cb7000-b6cb8000 r-xp 00000000 1f:04 69 /lib/libnsl.so.0

b6cb8000-b6cbf000 ---p 00000000 00:00 0

b6cbf000-b6cc0000 rw-p 00000000 1f:04 69 /lib/libnsl.so.0

b6cc0000-b6d59000 r-xp 00000000 1f:04 65 /lib/libc.so.0

b6d59000-b6d60000 ---p 00000000 00:00 0

b6d60000-b6d62000 rw-p 00098000 1f:04 65 /lib/libc.so.0

b6d62000-b6d67000 rw-p 00000000 00:00 0

b6d67000-b6d87000 r-xp 00000000 1f:04 71 /lib/libgcc\_s.so.1

b6d87000-b6d8e000 ---p 00000000 00:00 0

b6d8e000-b6d8f000 rw-p 0001f000 1f:04 71 /lib/libgcc\_s.so.1

b6d8f000-b6da0000 r-xp 00000000 1f:04 67 /lib/libm.so.0

b6da0000-b6da7000 ---p 00000000 00:00 0

b6da7000-b6da8000 rw-p 00010000 1f:04 67 /lib/libm.so.0

b6da8000-b6e5c000 r-xp 00000000 1f:04 79 /lib/libstdc++.so.6

b6e5c000-b6e63000 ---p 00000000 00:00 0

b6e63000-b6e67000 r--p 000b3000 1f:04 79 /lib/libstdc++.so.6

b6e67000-b6e69000 rw-p 000b7000 1f:04 79 /lib/libstdc++.so.6

b6e69000-b6e6f000 rw-p 00000000 00:00 0

b6e6f000-b6e83000 r-xp 00000000 1f:04 74 /lib/libpthread.so.0

b6e83000-b6e8a000 ---p 00000000 00:00 0

b6e8a000-b6e8b000 rw-p 00013000 1f:04 74 /lib/libpthread.so.0

b6e8b000-b6e8d000 rw-p 00000000 00:00 0

b6e8d000-b6f03000 r-xp 00000000 1f:05 4633696 /usr/lib/liblog4cpp.so.5

b6f03000-b6f0a000 ---p 00000000 00:00 0

b6f0a000-b6f0d000 rw-p 00075000 1f:05 4633696 /usr/lib/liblog4cpp.so.5

b6f0d000-b6f14000 r-xp 00000000 1f:04 77 /lib/ld-uClibc.so.0

b6f19000-b6f1b000 rw-p 00000000 00:00 0

b6f1b000-b6f1c000 rw-p 00006000 1f:04 77 /lib/ld-uClibc.so.0

be912000-be933000 rw-p 00000000 00:00 0 [stack]

ffff0000-ffff1000 r-xp 00000000 00:00 0 [vectors]

蓝色：代码和数据段

红色：堆

紫色：通过mmap出来的存放共享库文件

绿色：栈

橙色：中断向量表

1、brk系统调用

brk从heap向上简单的线性增长，从上面可以看出start\_brk表示当前current task的heap起始地址，brk指向已经分配了的heap。

申请：将申请的地址和start\_brk和brk进行比较，如果合法就修改brk指向地址，并返回告之成功

释放：修改brk指向，调用do\_munmap进行内存释放。

2、mmap系统调用

mmap是把一个文件或posix共享内存区映射到调用进程的地址空间。三个目的：

A．使用普通文件提供内存映射IO

B．使用特殊文件提供匿名映射IO

C．使用shm\_open以提供无亲缘关系的进程间posix共享内存区

其他细节见网上说明

malloc使用brk或mmap都是从current task的线性地址空间申请一块虚拟地址，并没有相应的物理内存。当task任务运行使用到上面的虚拟地址时，MMU转化虚拟地址，根据页表查找虚拟地址对应的物理内存，如果不存在就会产生缺页异常。缺页异常的处理大致逻辑如下图：



如果申请到了，就会将这个虚拟地址和物理内存的映射添加到current task的页表，同时刷新cache，下次访问就直接映射。