

malloc和free，brk和sbrk和mmap和munmap的使用和关系以及内存分配的原理

 blog.csdn.net/weixin_57023347/article/details/121291573

目录

一.使用

1.1 malloc和free

2.brk和sbrk

2.1 sbrk

2.2 brk

3. mmap/munmap

二.关系

三.内存分配原理

四.malloc底层

一.使用

1.1 malloc和free

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);CSDN @两片空白
```

参数：申请内存大小

返回值：成功返回申请空间起始指针，失败返回空。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5
6     char *str = (char *)malloc(10);
7     if(str == NULL){
8         perror("malloc fail");
9     }
10    int *ptr = (int *)malloc(10*sizeof(int));
11    if(ptr == NULL){
12        perror("malloc fail");
13    }
14
15
16
17    free(str);
18    //防止野指针
19    str = NULL;
20    free(ptr);
21    ptr = NULL
22    return 0;
23 }

```

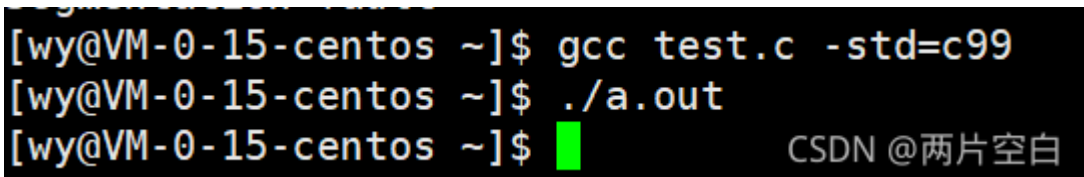
第一次调用malloc系统会分配33页(一页等于4kb)的空间。之后再申请空间会从33页空间剩余的空间中申请。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5
6     char *str = (char *)malloc(1);
7     if(str == NULL){
8         perror("malloc fail");
9     }
10    //只申请了1字节空间，但是这样遍历不会报错
11    for(int i=0; i<32*4096; i++){
12        str[i] = 'a';
13    }
14
15    return 0;
16 }

```

结果没有报错：



```

[wy@VM-0-15-centos ~]$ gcc test.c -std=c99
[wy@VM-0-15-centos ~]$ ./a.out
[wy@VM-0-15-centos ~]$

```

CSDN @两片空白

将代码改一下：

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5
6     char *str = (char *)malloc(1);
7     if(str == NULL){
8         perror("malloc fail");
9     }
10    //只申请了1字节空间，申请超过33页
11    for(int i=0; i<33*4096; i++){
12        str[i] = 'a';
13    }
14
15    return 0;
16 }

```

结果报错：

```

[wy@VM-0-15-centos ~]$ ./a.out
Segmentation fault

```

CSDN @两片空白

当你申请大小为n的内存时，起始系统给你的空间会大于你申请的空间数，一般后面会有12字节的控制信息。这个控制信息一般不可以修改。

2.brk和sbrk

brk和sbrk底层维护了一个堆上的指针，以增量的方式管理动态内存(堆)。

2.1 sbrk

```

#include <unistd.h>

int brk(void *addr);

void *sbrk(intptr_t increment);

```

CSDN @两片空白

参数：申请空间大小，0不申请空间，大于0申请空间，小于0 释放空间

返回值：申请内存空间的起始地址，失败返回-1。

来一段代码说明一下：

```

test.c+
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(){
    char *ptr0 = (char *)sbrk(0);
    printf("指针起始地址: %p\n", ptr0);
    //申请10字节空间
    char *ptr1 = (char *)sbrk(10);
    printf("ptr1: %p\n", ptr1);

    char *ptr2 = (char *)sbrk(0);
    printf("申请空间后的指针地址: %p\n", ptr2);
    //释放10字节的空间
    sbrk(-10);
    char *ptr3 = (char *)sbrk(0);
    printf("释放空间后的指针地址: %p\n", ptr3);

    return 0;
}

```

```

指针起始地址: 0x836000
ptr1: 0x836000
申请空间后的指针地址: 0x83600a
释放空间后的指针地址: 0x836000

```

sbrk管理堆上的一个指针，通过指针的移动来申请空间给用户

CSDN @两片空白

2.2 brk

```
#include <unistd.h>
```

```
int brk(void *addr);
```

CSDN @两片空白

参数：需要将堆顶指针向后移动到哪个地址。移动的空间大小，就是申请的空间大小。

返回值：失败返回-1，成功放回0。

```

int main(){
    char *ptr0 = (char *)sbrk(0);
    printf("指针起始地址: %p\n", ptr0);
    //申请10字节空间
    int res = brk(ptr0 + 10);
    printf("res: %d\n", res);

    char *ptr2 = (char *)sbrk(0);
    printf("申请空间后的指针地址: %p\n", ptr2);

    int res2 = brk(ptr2 - 10);

    char *ptr3 = (char *)sbrk(0);
    printf("申请空间后的指针地址: %p\n", ptr3);
    return 0;
}

```

堆顶指针向高地址移动了10字节

```

[wy@VM-0-15-centos ~]$ ./a.out
指针起始地址: 0x1fc4000
res: 0
申请空间后的指针地址: 0x1fc400a
申请空间后的指针地址: 0x1fc4000

```

释放10字节空间 堆顶指针向低地址方向移动了10字节

CSDN @两片空白

3. mmap/munmap

```
#include <sys/mman.h>
```

```

void *mmap(void *addr, size_t length, int prot, int flags,
            int fd, off_t offset);

```

```
int munmap(void *addr, size_t length);
```

CSDN @两片空白

mmap参数：

start：映射区的开始地址

length : 映射区的长度

prot : 期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过or运算合理地组合在一起

- 1 PROT_EXEC : 页内容可以被执行
- 2 PROT_READ : 页内容可以被读取
- 3 PROT_WRITE : 页可以被写入
- 4 PROT_NONE : 页不可访问

flags : 指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

- 1 MAP_FIXED //使用指定的映射起始地址，如果由start和len参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址必须落在页的边界上。
- 2 MAP_SHARED //与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。直到msync()或者munmap()被调用，文件实际上不会被更新。
- 3 MAP_PRIVATE //建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。
- 4 MAP_DENYWRITE //这个标志被忽略。
- 5 MAP_EXECUTABLE //同上
- 6 MAP_NORESERVE //不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。
- 7 MAP_LOCKED //锁定映射区的页面，从而防止页面被交换出内存。
- 8 MAP_GROWSDOWN //用于堆栈，告诉内核VM系统，映射区可以向下扩展。
- 9 MAP_ANONYMOUS //匿名映射，映射区不与任何文件关联。
- 10 MAP_ANON //MAP_ANONYMOUS的别称，不再被使用。
- 11 MAP_FILE //兼容标志，被忽略。
- 12 MAP_32BIT //将映射区放在进程地址空间的低2GB，MAP_FIXED指定时会被忽略。当前这个标志只在x86-64平台上得到支持。
- 13 MAP_POPULATE //为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违例阻塞。
- 14 MAP_NONBLOCK //仅和MAP_POPULATE一起使用时才有意义。不执行预读，只为已存在于内存中的页面建立页表入口。

fd : 有效的文件描述词。如果MAP_ANONYMOUS被设定，为了兼容问题，其值应为-1

offset : 被映射对象内容的起点

mmap返回值 :

成功返回指向该空间的起始地址，失败MAP_FAILED (that is, (void *) -1)被返回。

mmap作用 :

在文件映射区开辟一个大小为length的空间。

munmap参数 :

addr是调用mmap()时返回的地址，

len是映射区的大小

munmap返回值 :

成功执行时，munmap()返回0。失败时，munmap返回-1。

munmap作用：

该调用在进程地址空间中解除一个映射关系，即释放空间

```
int main(){
    char *ptr = (char *)mmap(NULL, 10, PROT_WRITE | PROT_READ, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
    printf("%p\n", ptr);
    munmap(ptr, 10);

    return 0;
}
```

不是文件必须加

CSDN @两片空白

操作文件：

```
int main(){
    int fd = open("./text.txt", O_CREAT | O_RDWR, 0644);
    char *ptr = (char *)mmap(NULL, 10, PROT_WRITE | PROT_READ, MAP_SHARED, fd, 0);
    printf("%p\n", ptr);
    strcpy(ptr, "hellow");
    munmap(ptr, 10);

    return 0;
}
```

```
[wy@VM-0-15-centos ~]$ ./a.out
0x7fcdc8879000
Bus error
```

报错原因时因为，文件大小为0，不能写入内容

往文件写入一些内容

```
1: text.txt
1 zscasd
```

在运行

```
[wy@VM-0-15-centos ~]$ ./a.out
0x7f556a818000
```

```
1: text.txt
1 hellow^@ zscasd
```

字符串写入文件

CSDN @两片空白

二.关系

malloc的底层调用的是brk和mmap来申请内存。对应free用的是brk和munmap释放内存。

三.内存分配原理

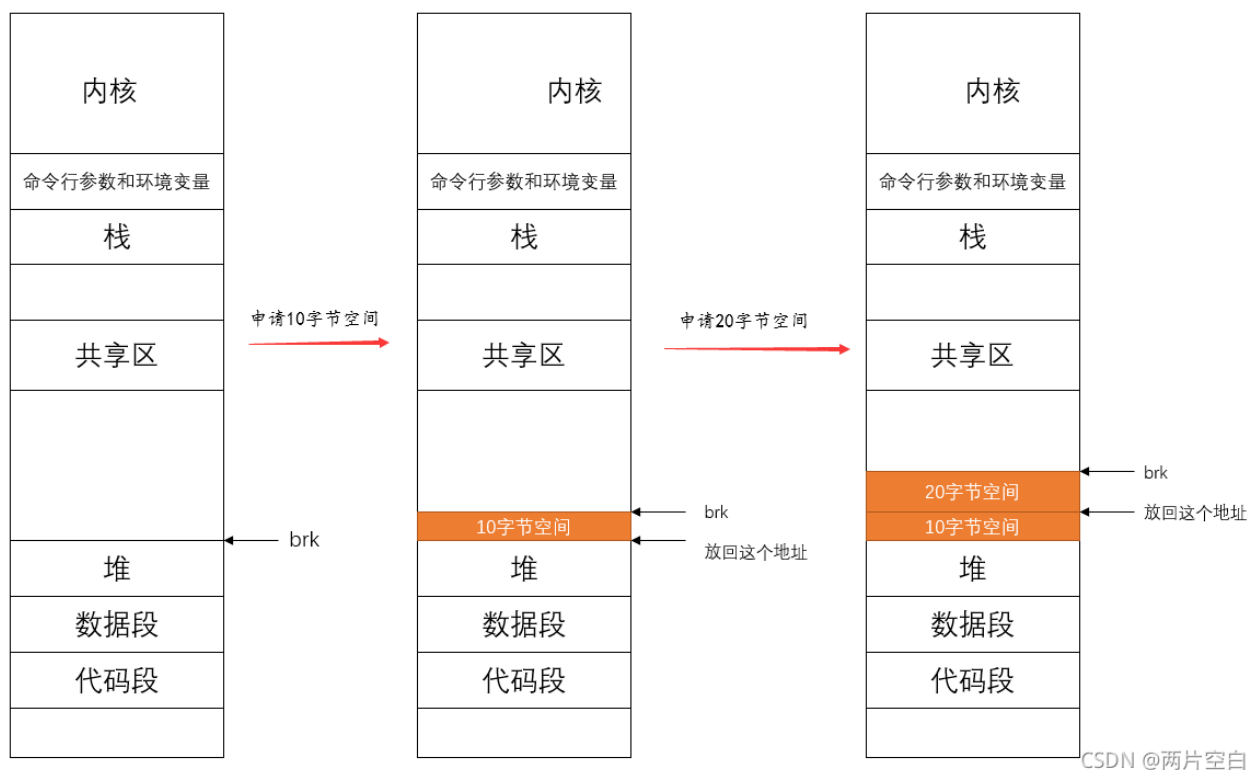
从操作系统的角度来看，进程分配内存有两种方式，分别由两个系统调用完成：**brk**和**mmap**（不考虑共享内存）。

- **brk**：将数据段最高地址往进程地址空间的高地址方向移动。
- **mmap**：在文件映射区申请一个空闲内存。

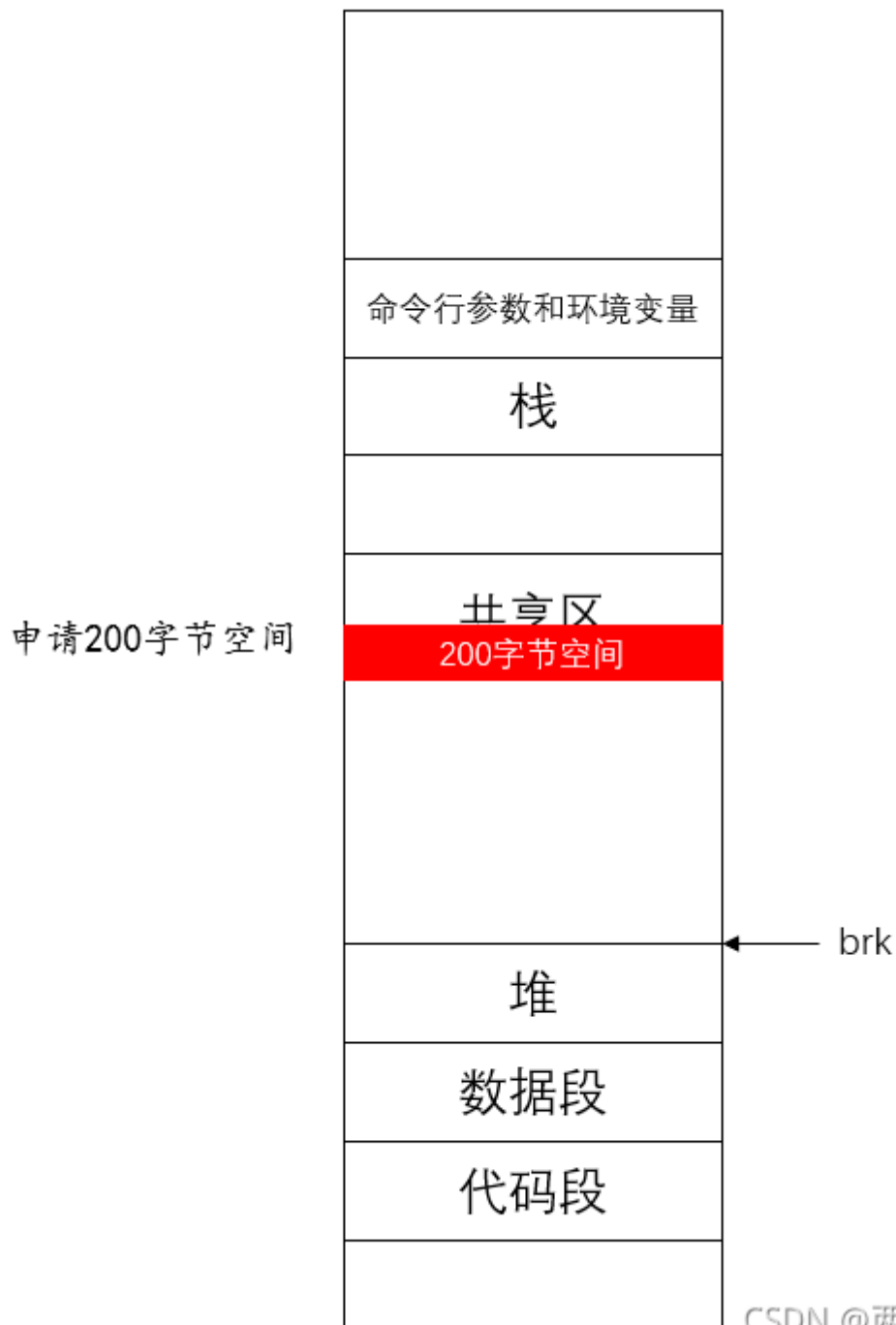
这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

分配内存时有两种情况：

情况一：申请的内存小于128KB，申请内存是将堆顶指针向高地址方向移动，不会作初始化。



情况二：大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)。



默认情况下，malloc函数分配内存，如果请求内存大于128K（可由M_MMAP_THRESHOLD选项调节），那就不是去推堆顶指针了，而是利用mmap系统调用，从堆和栈的中间分配一块虚拟内存。

原因：brk是以增量的形式申请空间的，brk分配的内存需要等到高地址内存释放以后才能释放。而mmap申请的内存可以单独释放。

既然堆内内存brk和sbrk不能直接释放，为什么不全部使用 mmap 来分配，munmap直接释放呢？

首先brk释放的空间，并不一定真正释放了。如果没有释放可以被重复利用。而mmap申请的空间，直接调用munmap是将空间真正释放了。

首先申请空间需要用用户态进入内核态，需要成本。而频繁调用mmap有用户态变为内核态的频率肯定会比用brk高。即CPU消耗会很高。

在glibc的malloc实现中，充分考虑了sbrk和mmap行为上的差异及优缺点，默认分配大块内存(128k)才使用mmap获得地址空间，也可通过mallopt(M_MMAP_THRESHOLD, <SIZE>)来修改这个临界值。

借鉴：[Linux内存分配小结--malloc、brk、mmap_gfgdsg的专栏-CSDN博客](#)

四.malloc底层

malloc底层管理的是一个链表数组，类似于哈希桶的结构。哈希桶的位置，映射的是内存块的大小。一般这个内存块的大小不会正好一字节一字节的增长，而是会设置一个对齐数，桶位置表示的内存块的大小绘制这个对齐数的整数倍。比如，对齐数为8字节，哈希桶的第一个位置连接的内存块都是8字节大小的内存块，第二个桶连接的都是16字节大小的内存，一次类推。

当用户malloc申请内存，通过用户申请的内存大小，通过算法，找到对应的桶的位置。**如果桶下有内存块**，将该内存块从哈希桶中弹出，然后返回该内存块的起始地址给用户。**如果当前桶没有内存块**，会向桶后找大的内存块，如果有内存块，会将内存块切割成两个内存块，间用户需要的内存块的起始地址返回给用户，将另外一个内存块连接到对应桶内。**如果桶后也没有大的内存块**，就会向系统申请一个较大的内存块，进行切割，返回用户需要的，新的连接到对应哈希桶位置。

用户free释放内存块并不是直接还给了系统，而是还给了哈希桶。如果有其它内存块和释放的内存块相邻，底层会将相邻的内存块合并，再连接到哈希桶对应位置。

上面是我自己简单总结，如有不足，希望大佬们可以帮忙指出，谢谢。

详细可以看[malloc 底层实现](#)