浅谈 C 内存分配

很早之前写的了，现在发到 C 版来。

关于 C 语言内存方面的话题要真说起来的话那恐怕就没头了，所以本文仅仅是一个浅谈。

关于内存问题不同平台之间有一定的区别。本文所指的平台是 x86的 Linux 平台

用 C 语言做程序（其实其他语言也一样），不仅要熟悉语法，其实很多相关的背景知识也很重要。在学习和研究 C 语言中内存分配的问题前，首先要了解一下 Linux 分配给进程（运行中的程序）的地址空间是什么样的。

总的来说有3个段，即代码段，数据段和堆栈段（学过汇编的朋友一定很熟悉了）。代码段就是存储程序文本的，所以有时候也叫做文本段，指令指针中的指令就是从这里取得。这个段一般是可以被共享的，比如你在 Linux 开了2个 Vi 来编辑文本，那么一般来说这两个 Vi 是共享一个代码段的，但是数据段不同（这点有点类似 C++中类的不同对象共享相同成员函数）。数据段是存储数据用的，还可以分成初始化为非零的数据区，BSS，和堆(Heap)三个区域。初始化非零数据区域一般存放静态非零数据和全局的非零数据。BSS 是 Block Started by Symbol 的缩写，原本是汇编语言中的术语。该区域主要存放未初始化的全局数据和静态数据。还有就是堆了，这个区域是给动态分配内存是使用的，也就是用 malloc 等函数分配的内存就是在这个区域里的。它的地址是向上增长的。最后一个堆栈段（注意，堆栈是 Stack,堆是 Heap，不是同一个东西），堆栈可太重要了，这里存放着局部变量和函数参数等数据。例如递归算法就是靠栈实现的。栈的地址是向下增长的。具体如下：

需要注意的是，代码段和数据段之间有明确的分隔，但是数据段和堆栈段之间没有，而且栈是向下增长，堆是向上增长的，因此理论上来说堆和栈会“增长到一起”，但是操作系统会防止这样的错误发生，所以不用过分担心。

有了以上理论做铺垫，下面就说动态内存的分配。上面说了，动态内存空间是在堆中分配的。实现动态分配的也就是下面几个函数：

stdlib.h ：

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
```

```
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```
一个一个说吧。malloc 就是分配一个 size 大小的内存空间，并且用一个 void 类型的指针指向这个空间，然后返回这个指针。也就是说，malloc 返回了一个指向 size 大小的空间的 void 类型的指针，如果要使用这个空间，还得把 void*类型转换成一个你需要的类型，比如 int*之类。 calloc 和 malloc 基本一样，不同的是有两点，一是 calloc 分配的空间大小是由 nmemb*size 决定的，也就是说 nmemb 是条目个数，而 size 可以看成是条目的大小，计算总空间任务由 calloc 去做。二是 calloc 返回的空间都用0填充，而 malloc 则不确定内存中会有什么东西。realloc 是用来改变已经分配的空间的大小。指针 ptr 是 void 类型的，它应该指向一个需要重新分配大小的空间，而 size 参数则是重新分配之后的整个空间大小，而不是增加的大小。同样，返回的是一个指向新空间的指针。 free 用来释放由上面3个函数分配的空间，其参数就是指向某空间的指针。

基本就这些了，这些都是比较基础的话题，高级话题和细节问题还有很多，这里就不进行说明了，有机会我会继续总结一番的

```
1 #include
2 #include
3 #include
4
5 int main()
6 {
7         char *p = malloc(1024);
8         int array[1024];
9         int i;
10
11         memset(p, 'a', 1023);
12         p[1023] = 0;
13
14         for(i = 0; i 编译执行之后，查看/proc//maps 文件，可以得到类似这样的内容：
```

复制内容到剪贴板

```
代码:infohunter:/proc/7582# cat maps
08048000-08049000 r-xp 00000000 03:07 17086      /data/program/mem-eg
08049000-0804a000 rw-p 00000000 03:07 17086      /data/program/mem-eg
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7e55000-b7e56000 rw-p b7e55000 00:00 0
b7e56000-b7f7d000 r-xp 00000000 03:03 685000     /lib/tls/i686/cmov/libc-2.3.6.so

b7f7d000-b7f82000 r--p 00127000 03:03 685000     /lib/tls/i686/cmov/libc-2.3.6.so

b7f82000-b7f84000 rw-p 0012c000 03:03 685000     /lib/tls/i686/cmov/libc-2.3.6.so
```

```
b7f84000-b7f87000 rw-p b7f84000 00:00 0
b7f96000-b7f99000 rw-p b7f96000 00:00 0
b7f99000-b7f9a000 r-xp b7f99000 00:00 0          [vdso]
b7f9a000-b7faf000 r-xp 00000000 03:03 358430     /lib/ld-2.3.6.so
b7faf000-b7fb1000 rw-p 00014000 03:03 358430     /lib/ld-2.3.6.so
bfb56000-bfb6c000 rw-p bfb56000 00:00 0          [stack]
```

infohunter:/proc/7582#maps 文件的格式大致这样，各字段以空格分割：

：中的 r,w,x 代表读、写和可执行。最后一位可以是 p 或者是 s，代表私有或共享

根据 maps 的输出可以清楚的了解到进程的地址空间。在我的这个例子中，由上到下，地址增大

只读的代码段：08048000-08049000 r-xp 00000000 03:07 17086 /data/program/mem-eg

可读写的数据段：08049000-0804a000 rw-p 00000000 03:07 17086 /data/program/mem-eg

堆：0804a000-0806b000 rw-p 0804a000 00:00 0 [heap]

中间是库的代码段、数据段，省略.....

最后，也就是最高地址部分为栈空间： bfb56000-bfb6c000 rw-p bfb56000 00:00 0 [stack]

Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process or thread. Each row has the following fields:

address - This is the starting and ending address of the region in the process's address space

permissions - This describes how pages in the region can be accessed. There are four different permissions: read, write, execute, and shared. If read/write/execute are disabled, a '-' will appear instead of the 'r'/'w'/'x'. If a region is not shared, it is private, so a 'p' will appear instead of an 's'. If the process attempts to access memory in a way that is not permitted, a segmentation fault is generated. Permissions can be changed using the mprotect system call.

offset - If the region was mapped from a file (using mmap), this is the offset in the file where the mapping begins. If the memory was not mapped from a file, it's just 0.

device - If the region was mapped from a file, this is the major and minor device number (in hex) where the file lives.

inode - If the region was mapped from a file, this is the file number.

pathname - If the region was mapped from a file, this is the name of the file. This field is blank for anonymous mapped regions. There are also special regions with names like [heap], [stack], or [vdso]. [vdso] stands for virtual dynamic shared object. It's used by system calls to switch to kernel mode. Here's a good article about it.

You might notice a lot of anonymous regions. These are usually created by mmap but are not attached to any file. They are used for a lot of miscellaneous things like shared memory or buffers not allocated on the heap. For instance, I think the pthread library uses anonymous mapped regions as stacks for new threads.