编译与底层

请你来说一下一个C++源文件从文本到可执行文件经历的过程？

预处理阶段：对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。

编译阶段：将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件

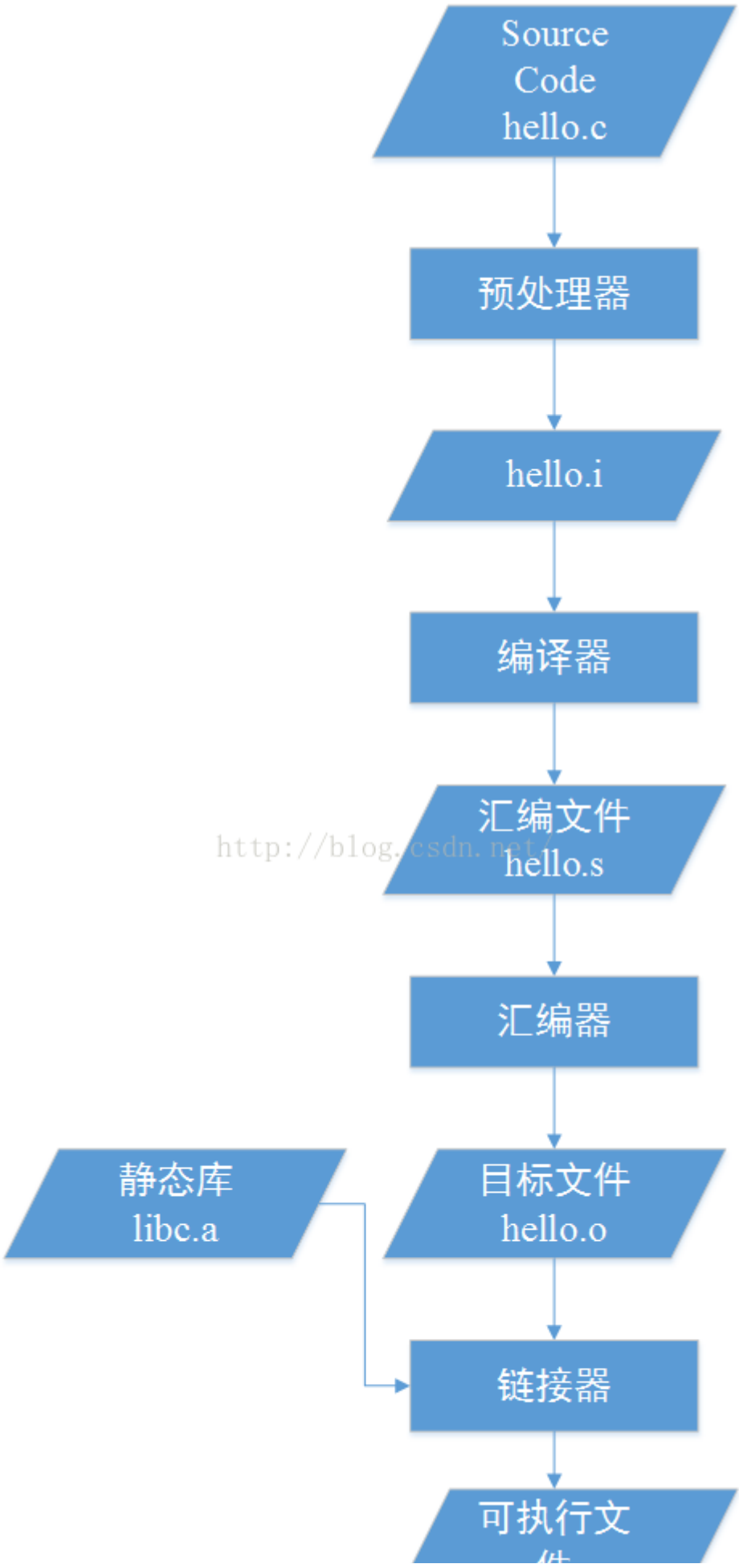
汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

参考网址

对于C/C++编写的程序，从源代码到可执行文件，一般经过下面四个步骤： 1).预处理，产生.ii文件 2).编译，产生汇编文件(.s文件) 3).汇编，产生目标文件(.o或.obj文件) 4).链接,产生可执行文件(.out或.exe文件)

以hello.c为例，这个过程可以用下面的图来表示



二.预处理 预处理主要包含下面的内容： a.对所有的“#define”进行宏展开； b.处理所有的条件编译指令，比如“#if”,“#ifdef”，“#elif”，“#else”,“#endif” c.处理“#include”指令，这个过程是递归的，也就是说被包含的文件可能还包含其他文件 d.删除所有的注释“//”和“/**/” e.添加行号和文件标识 f.保留所有的“#pragma”编译器指令 经过预处理后的.ii文件不包含任何宏定义，因为所有的宏已经被展开，并且包含的文件也已经被插入到.ii文件中。

三.编译 编译的过程就是将预处理完的文件进行一系列词法分析，语法分析，语义分析及优化后生成相应的汇编代码文件(.s文件)

四.汇编 汇编器是将汇编代码转变成机器可以执行的代码，每一个汇编语句几乎都对应一条机器指令。最终产生目标文件(.o或.obj文件)。

五.链接 链接的过程主要包括了地址和空间分配(Address and Storage Allocation)、符号决议(Symbol Resolution)和重定位(Relocation)

请你来回答一下include头文件的顺序以及双引号""和尖括号<>的区别?

Include头文件的顺序：对于include的头文件来说，如果在文件a.h中声明一个在文件b.h中定义的变量，而不引用b.h。那么要在a.c文件中引用b.h文件，并且要先引用b.h，后引用a.h,否则汇报变量类型未声明错误。

双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样。

对于使用双引号包含的头文件，查找头文件路径的顺序为：

当前头文件目录

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径

对于使用尖括号包含的头文件，查找头文件的路径顺序为：

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径

请你回答一下malloc的原理，另外brk系统调用和mmap系统调用的作用分别是什么?

malloc函数用于动态分配内存。为了减少内存碎片和系统调用的开销，malloc其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作为内存管理的基本单位。当用户申请内存时，直接从堆区分配一块合适的空闲块。malloc采用隐式链表结构将堆区分成连续的、大小不一的块，包含已分配块和未分配块；同时malloc采用显示链表结构来管理所有的空闲块，即使用一个双向链表将空闲块连接起来，每一个空闲块记录了一个连续的、未分配的地址。

当进行内存分配时，malloc会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc采用边界标记法，根据每个块的前后块是否已经分配来决定是否进行块合并。

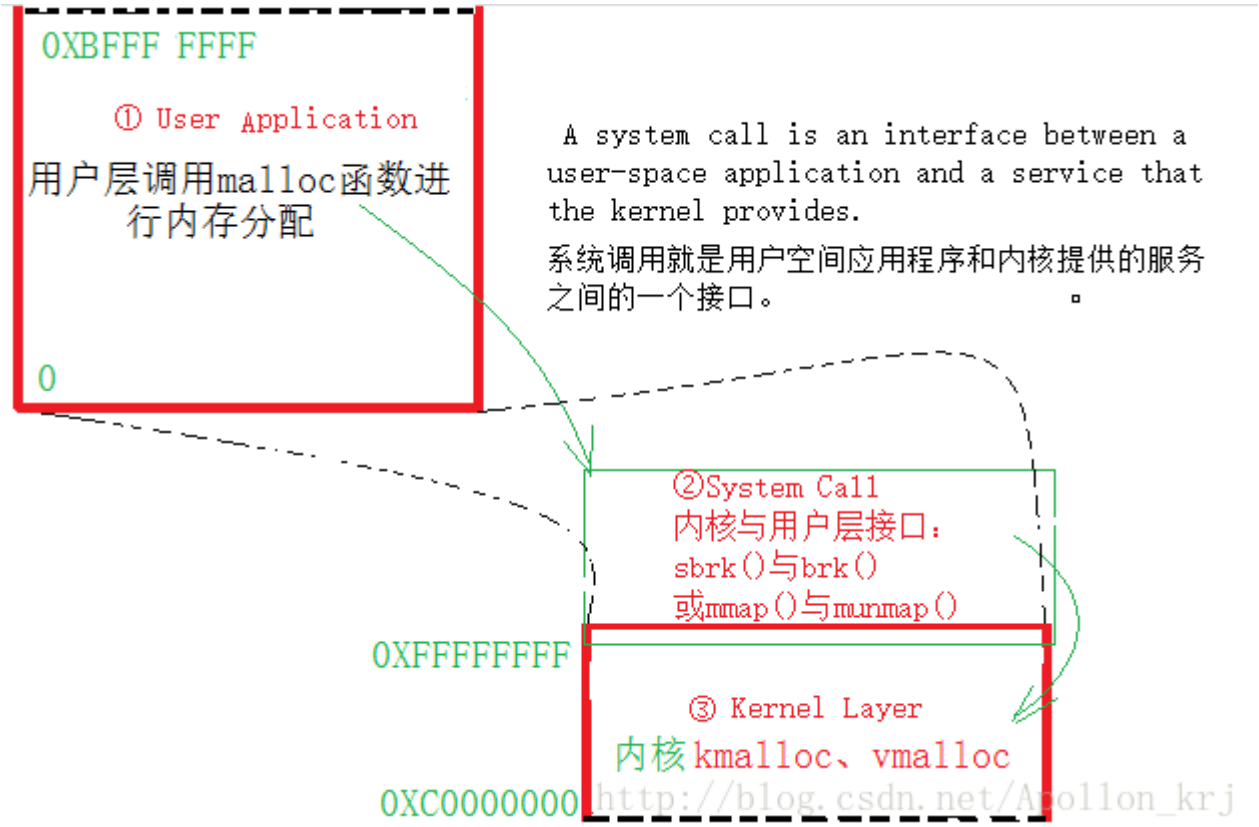
malloc在申请内存时，一般会通过brk或者mmap系统调用进行申请。其中当申请内存小于128K时，会使用系统函数brk在堆区中分配；而当申请内存大于128K时，会使用系统函数mmap在映射区分配。

参考网址

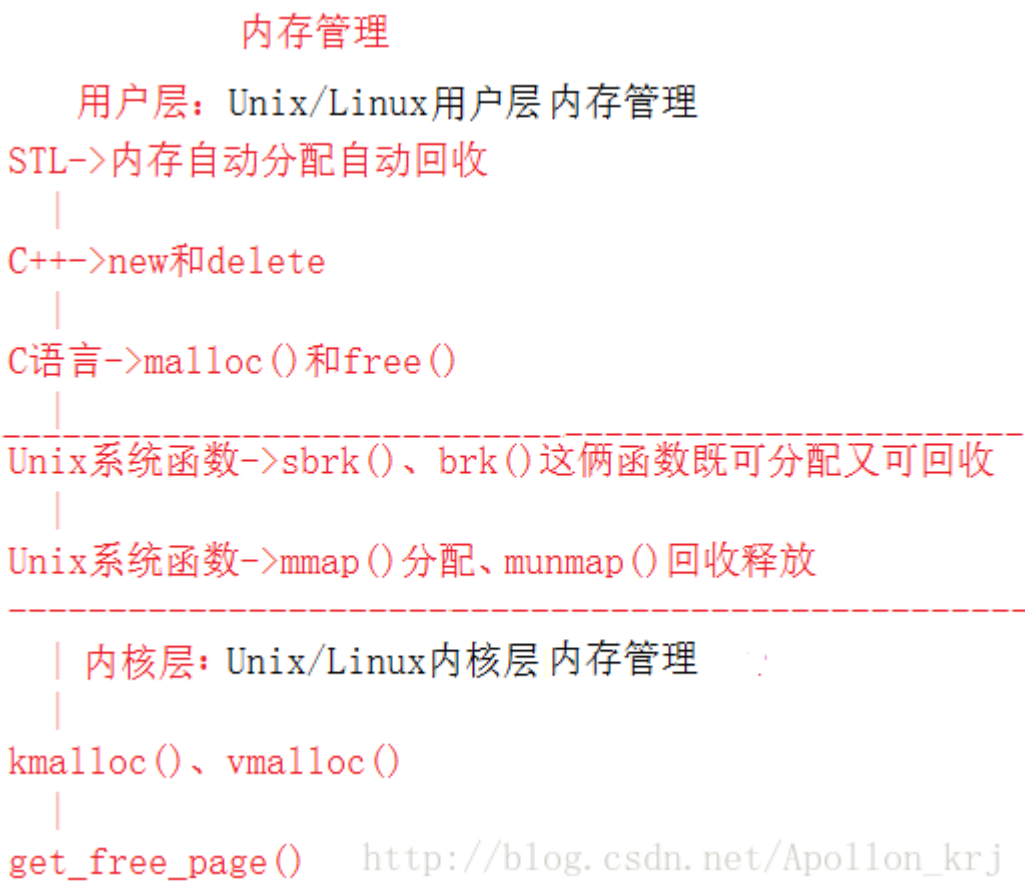
一、系统调用（System Call）：

在Linux中，4G内存可分为两部分——内核空间1G（3~4G）与用户空间3G（0~3G）,我们通常写的C代码都是在对用户空间即0~3G的内存进行操作。而且，用户空间的代码不能直接访问内核空间，因此内核空间提供了一系列的函数，实现用户空间进入内核空间的接口，这一系列的函数称为系统调用（System Call）。比如我们经常使用的open、close、read、write等函数都是系统级别的函数（man 2 function_name），而像fopen、fclose、fread、fwrite等都是用户级别的函数（man 3 function_name）。不同级别的函数能够操作的内存区域自然也就不同。

我们用一幅图来描述函数的调用过程：



对于C++中new与delete的底层则是用malloc和free实现。而我们所用的malloc()、free()与内核之间的接口（桥梁）就是sbrk()等系统函数；当然我们也可以直接调用系统调用（系统函数）,达到同样的作用。我们可以用下面这幅图来描述基本内存相关操作之间的关系：



虽然使用系统调用会带来一定的好处，但是物极必反，系统调用并非能频繁使用。由于程序由用户进入内核层时，会将用户层的状态先封存起来，然后到内核层运行代码，运行结束以后，从内核层出来到用户层时，再把数据加载回来。因此，频繁的系统调用效率很低。今天我们就系统调用层面来对内存操作做进一步的了解。

二、内存管理（Memory Management）系统调用：

1、brk()与sbrk():

（1）、函数原型与实现：

```
//函数原型:
#include<unistd.h>
int brk(void * addr);
void * sbrk(intptr_t increment);
```

由于sbrk()与brk()这两个系统函数有点所谓怪异，我们先来看看man手册对于sbrk()与brk()的描述：

DESCRIPTION

> brk() and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.
>
> brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see setrlimit(2)).
>
> sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

RETURN VALUE

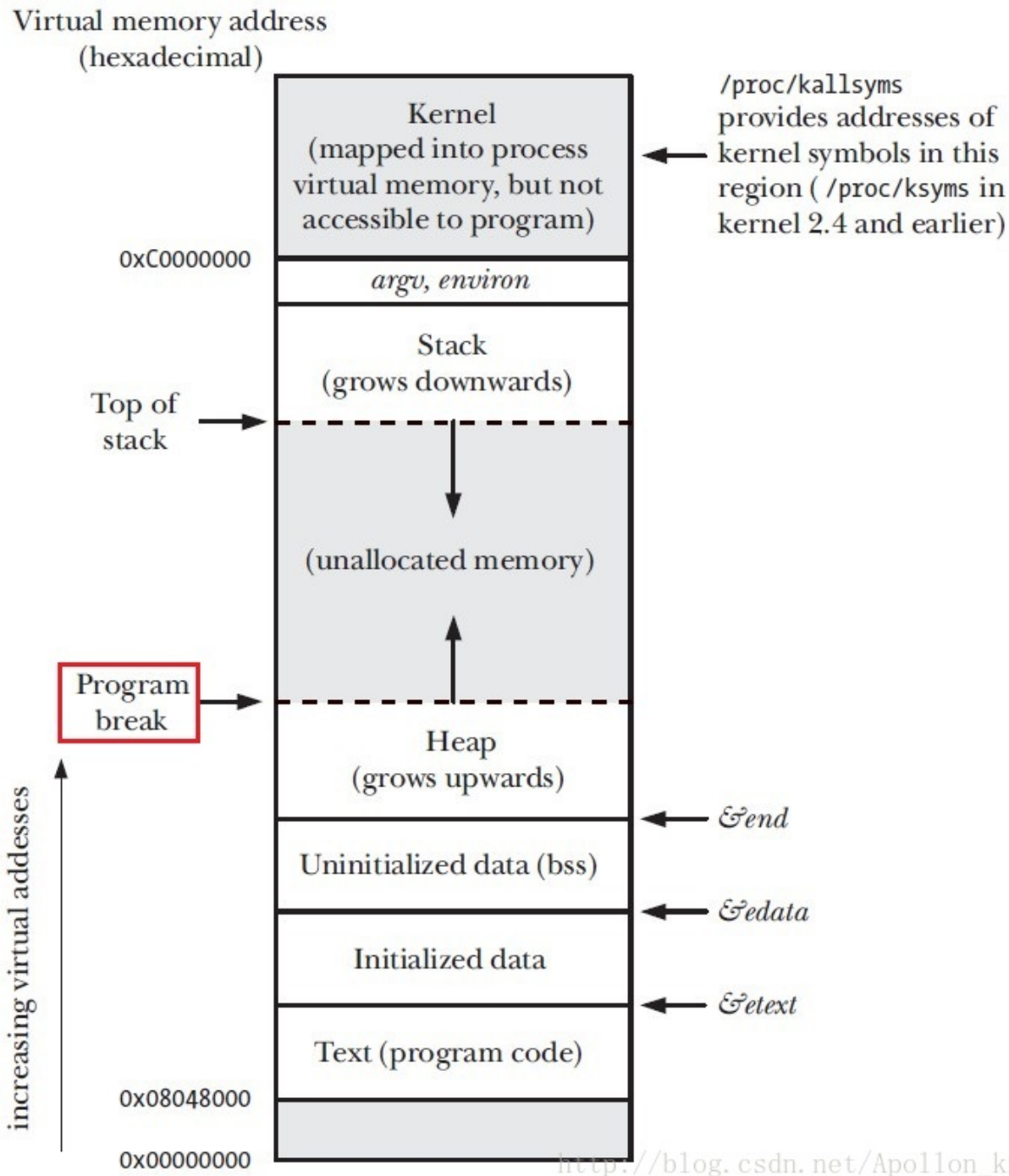
> On success, brk() returns zero. On error, -1 is returned, and errno is set to ENOMEM.
>
> On success, sbrk() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to ENOMEM.

描述： brk()和sbrk()改变程序间断点的位置。程序间断点就是程序数据段的结尾。（程序间断点是为初始化数据段的起始位置）.通过增加程序间断点进程可以更有效的申请内存 。当addr参数合理、系统有足够的内存并且不超过最大值时brk()函数将数据段结尾设置为addr,即间断点设置为addr。sbrk()将程序数据空间增加increment字节。当increment为0时则返回程序间断点的当前位置。

返回值： brk()成功返回0，失败返回-1并且设置errno值为ENOMEM（注：在mmap中会提到）。

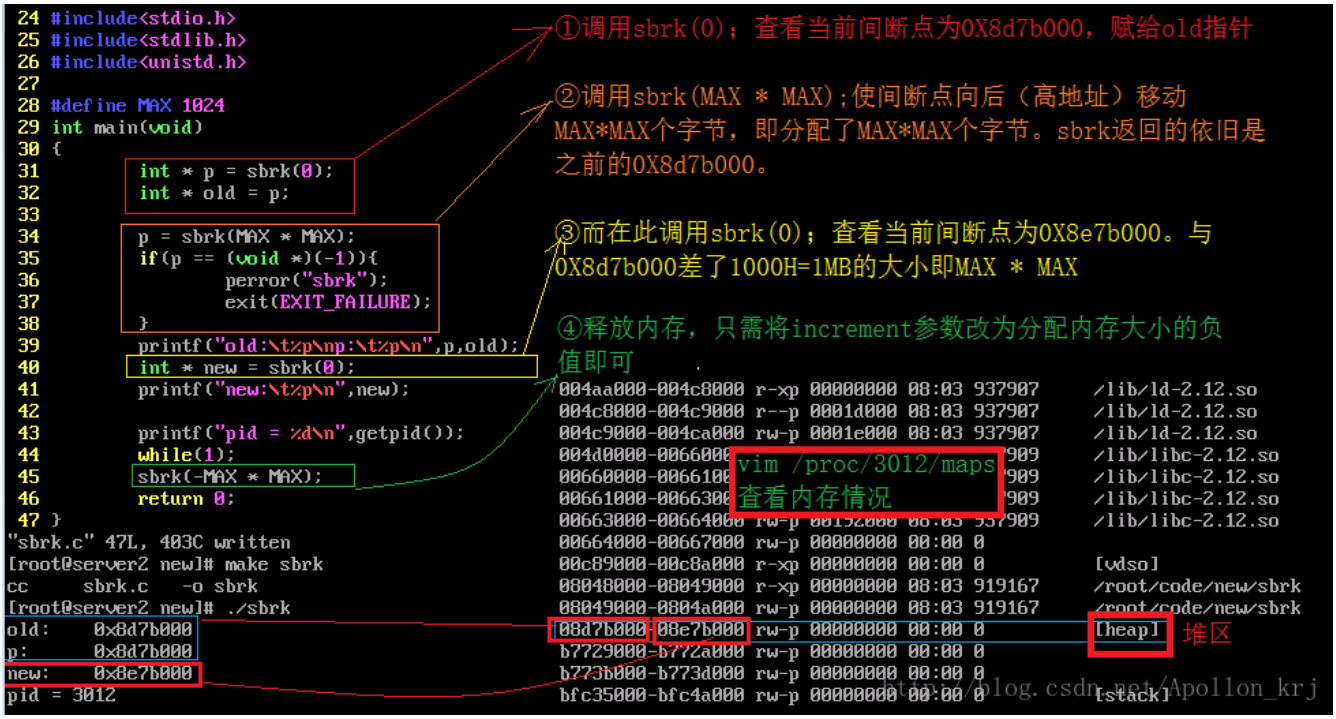
sbrk()成功返回之前的程序间断点地址。如果间断点值增加，那么这个指针（指的是返回的之前的间断点地址）是指向分配的新的内存的首地址。如果出错失败，就返回一个指针并设置errno全局变量的值为ENOMEM。

总结： 这两个函数都用来改变 “program break” (程序间断点)的位置，改变数据段长度（Change data segment size），实现虚拟内存到物理内存的映射。 brk()函数直接修改有效访问范围的末尾地址实现分配与回收。sbrk()参数函数中：当increment为正值时，间断点位置向后移动increment字节。同时返回移动之前的位置，相当于分配内存。当increment为负值时，位置向前移动increment字节，相当与于释放内存，其返回值没有实际意义。当increment为0时，不移动位置只返回当前位置。参数increment的符号决定了是分配还是回收内存。而关于program break的位置如图所示：



（2）、简单测试：

对于分配好的内存，我们只要有其首地址old与长度MAX*MAX即可不越界的准确使用（如下图所示），其效果与malloc相同，只不过sbrk()与brk()是C标准函数的底层实现而已，其机制较为复杂（测试中，死循环是为了查看maps文件，不至于进程消亡文件随之消失）。



虽然，sbrk()与brk()均可分配回收兼职，但是我们一般用sbrk()分配内存，而用brk()回收内存，上例中回收内存可以这样写：

```
int err = brk(old);
/**或者brk(p);效果与sbrk(-MAX*MAX);是一样的，但brk()更方便与清晰明了。**/
if(-1 == err){
    perror("brk");
    exit(EXIT_FAILURE);
}
```

2、mmap()与munmap(): mmap函数（地址映射）：mmap将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零(Linux堆空间未使用内存均清零)。这里我们只研究mmap的内存映射，而暂时不讨论文件方面的问题。关于mmap的文件映射的更详细的内容可参考认真分析mmap：是什么 为什么 怎么用，这个不错

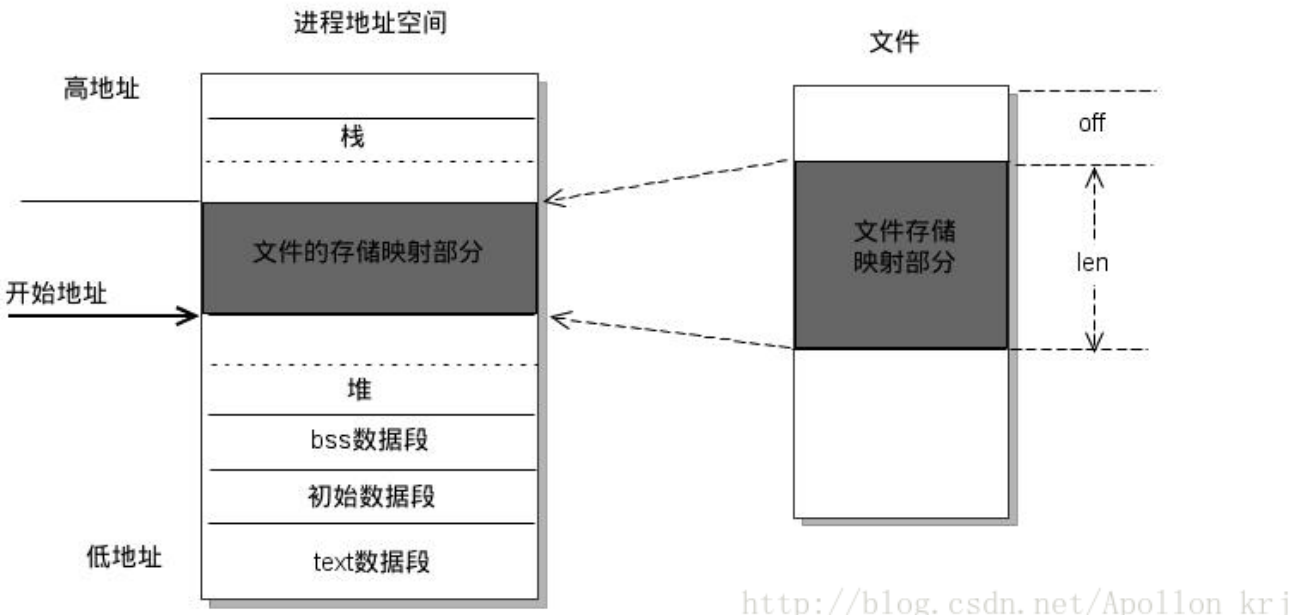
```
//函数原型：
#incldue<sys/mman.h>
void * mmap(void * addr, size_t length,int prot,int flags,int fd,off_t offset);
```

> 参数
>
> > （1）、addr: 起始地址，置零让系统自行选择并返回即可. （2）、length: 长度，不够一页会自动凑够一页的整数倍，我们可以宏定义#define MIN_LENGTH_MMAP 4096为一页大小 （3）、prot: 读写操作权限，PROT_READ可读、PROT_WRITE可写、PROT_EXEC可执行、PROT_NONE映射区域不能读取。（注意PROT_XXXXX与文件本身的权限不冲突，如果在程序中不设定任何权限，即使本身存在读写权限，该进程也不能对其操作） （4）、flags常用标志: ①MAP_SHARED【share this mapping】、MAP_PRIVATE【Create a private copy-on-write mapping】 MAP_SHARED只能设置文件共享，不能地址共享，即使设置了共享，对于两个进程来说，也不会生效。而MAP_PRIVATE则对于文件与内存都可以设置为私有。 ②MAP_ANON【Deprecated】、MAP_ANONYMOUS：匿名映射，如果映射地址需要加该参数，如果不加默认映射文件。MAP_ANON已经过时，只需使用MAP_ANONYMOUS即可 （5）、文件描述符：fd （6）、文件描述符偏移量：offset

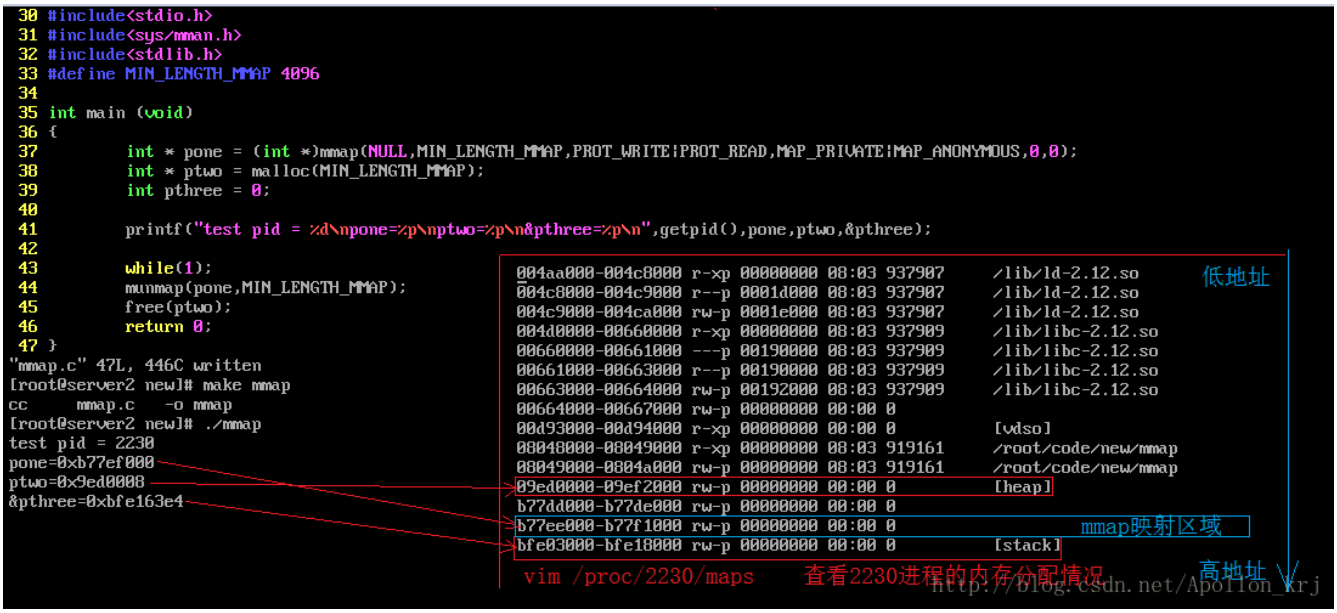
> （fd和offset对于一般性内存分配来说设置为0即可）
>
> 返回值：
>
> > 失败返回MAP_FAILED，即(void * (-1))并设置errno全局变量。 成功返回指向mmap area的指针pointer。
>
> 常见errno错误：
>
> > ①ENOMEM：内存不足； ②EAGAIN：文件被锁住或有太多内存被锁住； ③EBADF：参数fd不是有效的文件描述符； ④EACCES：存在权限错误，。如果是MAP_PRIVATE情况下文件必须可读；使用MAP_SHARED则文件必须能写入，且设置prot权限必须为PROT_WRITE。
> >
> > ⑤EINVAL：参数addr、length或者offset中有不合法参数存在。

munmap函数：解除映射关系

```
int munmap(void * addr, size_t length);//addr为mmap函数返回接收的地址，length为请求分配的长度。
```



这张图描述了mmap内存地址映射的位置关系（栈区以上为内核空间）。关于这一点我们可以作以简单的测试(我采用MIN_LENGTH_MMAP宏，当然你也可以用多少申请多少，系统总是以最小1页来映射的，关于内存分页与虚拟地址映射可参考Linux系统内存管理与内存分页机制)：



mmap映射的地址处于堆区与栈区中间，malloc映射的堆区内存为33页（最小映射大小），而mmap映射的内存为3页，也是4096B的整数倍。

参考网址 2

Linux 的虚拟内存管理有几个关键概念：

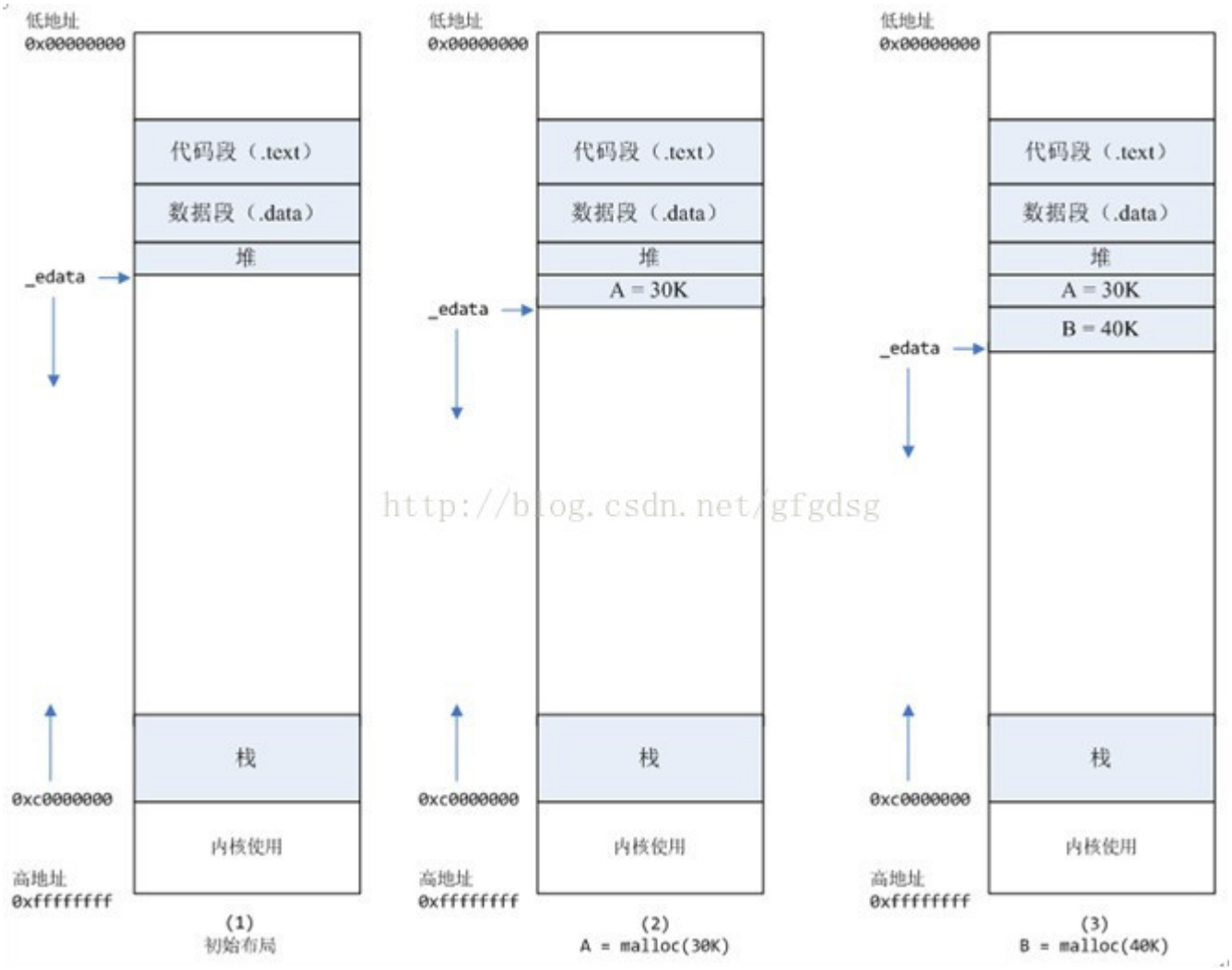
1、每个进程都有独立的虚拟地址空间，进程访问的虚拟地址并不是真正的物理地址； 2、虚拟地址可通过每个进程上的页表(在每个进程的内核虚拟地址空间)与物理地址进行映射，获得真正物理地址； 3、如果虚拟地址对应物理地址不在物理内存中，则产生缺页中断，真正分配物理地址，同时更新进程的页表；如果此时物理内存已耗尽，则根据内存替换算法淘汰部分页面至物理磁盘中。

基于以上认识，进行了如下分析： 一、Linux 虚拟地址空间如何分布？ Linux 使用虚拟地址空间，大大增加了进程的寻址空间，由低地址到高地址分别为： 1、只读段：该部分空间只能读，不可写；(包括：代码段、rodata 段(C常量字符串和#define定义的常量)) 2、数据段：保存全局变量、静态变量的空间； 3、堆 ：就是平时所说的动态内存，malloc/new 大部分都来源于此。其中堆顶的位置可通过函数 brk 和 sbrk 进行动态调整。 4、文件映射区域 ：如动态库、共享内存等映射物理空间的内存，一般是 mmap 函数所分配的虚拟地址空间。 5、栈：用于维护函数调用的上下文空间，一般为 8M ，可通过 ulimit –s 查看。

6、内核虚拟空间：用户代码不可见的内存区域，由内核管理(页表就存放在内核虚拟空间)。

下面以一个例子来说明内存分配的原理：

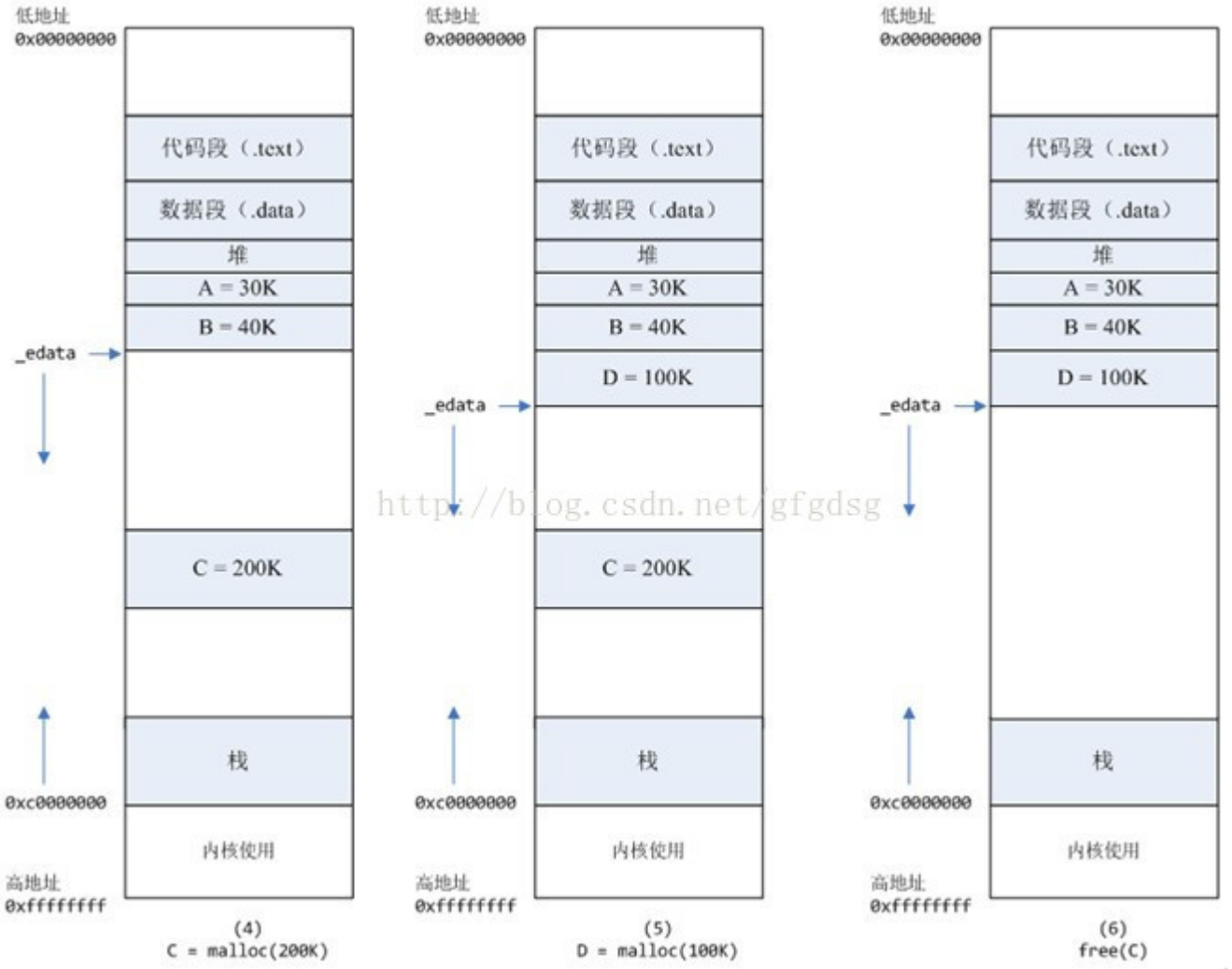
情况一、malloc小于128k的内存，使用brk分配内存，将_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)，如下图：



1、进程启动的时候，其（虚拟）内存空间的初始布局如图1所示。　　其中，mmap内存映射文件是在堆和栈的中间（例如libc-2.2.93.so，其它数据文件等），为了简单起见，省略了内存映射文件。　　*edata指针（glibc里面定义）指向数据段的最高地址。 2、进程调用A=malloc(30K)以后，内存空间如图2：　　malloc函数会调用brk系统调用，将*edata指针往高地址推30K，就完成虚拟内存分配。　　你可能会问：只要把*edata+30K就完成内存分配了？　　*事实是这样的，*edata+30K只是完成虚拟地址的分配，A这块内存现在还是没有物理页与之对应的，等到进程第一次读写A这块内存的时候，发生缺页中断，这个时候，内核才分配A这块内存对应的物理页。也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。

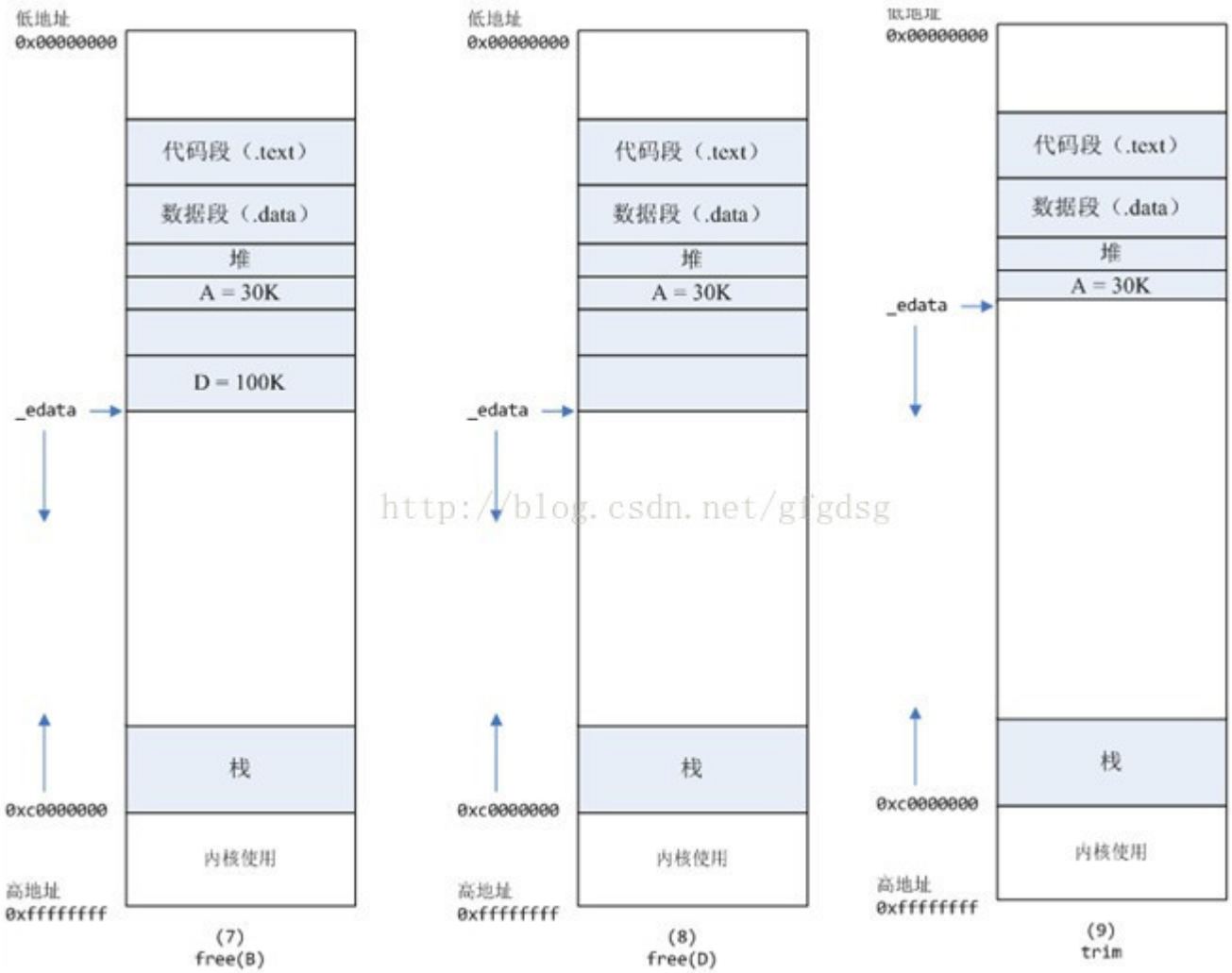
3、进程调用B=malloc(40K)以后，内存空间如图3。

情况二、malloc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)，如下图：



4、进程调用C=malloc(200K)以后，内存空间如图4：　　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_edata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。　　这样子做主要是因为::　　brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候紧缩看下面），而mmap分配的内存可以单独释放。　　当然，还有其它的好处，也有坏处，再具体下去，有兴趣的同学可以去看glibc里面malloc的代码了。 5、进程调用D=malloc(100K)以后，内存空间如图5；

6、进程调用free(C)以后，C对应的虚拟内存和物理内存一起释放。



7、进程调用free(B)以后，如图7所示：　　B对应的虚拟内存和物理内存都没有释放，因为只有一个_edata指针，如果往回推，那么D这块内存怎么办呢？ 当然，B这块内存，是可以重用的，如果这个时候再来一个40K的请求，那么malloc很可能就把B这块内存返回回去了。 8、进程调用free(D)以后，如图8所示：　　B和D连接起来，变成一块140K的空闲内存。 9、默认情况下：　　当最高地址空间的空闲内存超过128K（可由M_TRIM_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩，变成图9所示。

三、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？

既然堆内碎片不能直接释放，导致疑似“内存泄露”问题，为什么 malloc 不全部使用 mmap 来实现呢(mmap分配的内存可以会通过 munmap 进行 free ，实现真正释放)？而是仅仅对于大于 128k 的大块内存才使用 mmap ？

其实，进程向 OS 申请和释放地址空间的接口 sbrk/mmap/munmap 都是系统调用，频繁调用系统调用都比较消耗系统资源的。并且， mmap 申请的内存被 munmap 后，重新申请会产生更多的缺页中断。例如使用 mmap 分配 1M 空间，第一次调用产生了大量缺页中断 (1M/4K 次) ，当munmap 后再次分配 1M 空间，会再次产生大量缺页中断。缺页中断是内核行为，会导致内核态CPU消耗较大。另外，如果使用 mmap 分配小内存，会导致地址空间的分片更多，内核的管理负担更大。

同时堆是一个连续空间，并且堆内碎片由于没有归还 OS ，如果可重用碎片，再次访问该内存很可能不需产生任何系统调用和缺页中断，这将大大降低 CPU 的消耗。 因此， 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,) 来修改这个临界值。

五、C语言的内存分配方式与malloc

C语言跟内存分配方式 （1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。 （2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运 算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。 （3）从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多

C语言跟内存申请相关的函数主要有 alloc,calloc,malloc,free,realloc,sbrk等.其中alloc是向栈申请内存,因此无需释放. malloc分配的内存是位于堆中的,并且没有初始化内存的内容,因此基本上malloc之后,调用函数memset来初始化这部分的内存空间.calloc则将初始化这部分的内存,设置为0. 而realloc则对malloc申请的内存进行大小的调整.申请的内存最终需要通过函数free来释放. 而sbrk则是增加数据段的大小; malloc/calloc/free基本上都是C函数库实现的,跟OS无关.C函数库内部通过一定的结构来保存当前有多少可用内存.如果程序 malloc的大小超出了库里所留存的空间,那么将首先调用brk系统调用来增加可用空间,然后再分配空间.free时,释放的内存并不立即返回给os, 而是保留在内部结构中. 可以打个比方: brk类似于批发,一次性的向OS申请大的内存,而malloc等函数则类似于零售,满足程序运行时的要求.这套机制类似于缓冲. 使用这套机制的原因: 系统调用不能支持任意大小的内存分配(有的系统调用只支持固定大小以及其倍数的内存申请,这样的话,对于小内存的分配会造成浪费; 系统调用申请内存代价昂贵,涉及到用户态和核心态的转换. 函数malloc()和calloc()都可以用来分配动态内存空间，但两者稍有区别。

在Linux系统上，程序被载入内存时，内核为用户进程地址空间建立了代码段、数据段和堆栈段，在数据段与堆栈段之间的空闲区域用于动态内存分配。 内核数据结构mm_struct中的成员变量start_code和end_code是进程代码段的起始和终止地址，start_data和 end_data是进程数据段的起始和终止地址，start_stack是进程堆栈段起始地址，start_brk是进程动态内存分配起始地址（堆的起始 地址），还有一个 brk（堆的当前最后地址），就是动态内存分配当前的终止地址。 C语言的动态内存分配基本函数是malloc()，在Linux上的基本实现是通过内核的brk系统调用。brk()是一个非常简单的系统调用，只是简单地改变mm_struct结构的成员变量brk的值。 mmap系统调用实现了更有用的动态内存分配功能，可以将一个磁盘文件的全部或部分内容映射到用户空间中，进程读写文件的操作变成了读写内存的操作。在 linux/mm/mmap.c文件的do_mmap_pgoff()函数，是mmap系统调用实现的核心。do_mmap_pgoff()的代码，只是新建了一个vm_area_struct结构，并把file结构的参数赋值给其成员变量m_file，并没有把文件内容实际装入内存。

Linux内存管理的基本思想之一，是只有在真正访问一个地址的时候才建立这个地址的物理映射。

请你说一说C++的内存管理是怎样的?

在C++中，虚拟内存分为代码段、数据段、BSS段、堆区、文件映射区以及栈区六部分。

代码段:包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

数据段：存储程序中已初始化的全局变量和静态变量

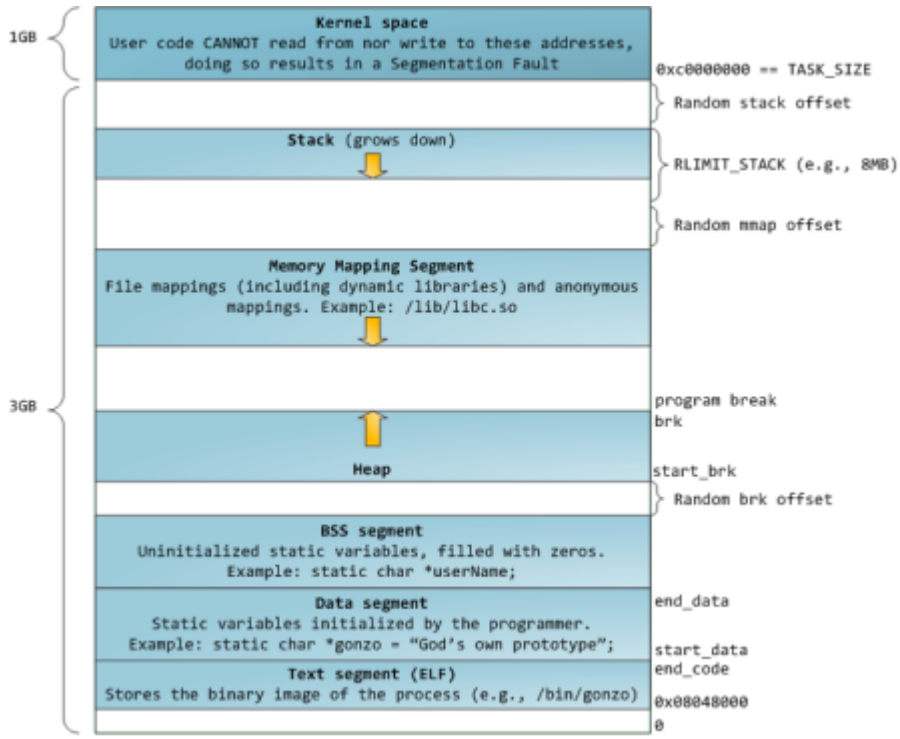
bss 段：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为0的全局变量和静态变量。

堆区：调用new/malloc函数时在堆区动态分配内存，同时需要调用delete/free来手动释放申请的内存。

映射区:存储动态链接库以及调用mmap函数进行的文件映射

栈：使用栈空间存储函数的返回地址、参数、局部变量、返回值

请你来说一下C++/C的内存分配



32bitCPU可寻址4G线性空间，每个进程都有各自独立的4G逻辑地址，其中0~3G是用户态空间，3~4G是内核空间，不同进程相同的逻辑地址会映射到不同的物理地址中。其逻辑地址其划分如下：

各个段说明如下：

3G用户空间和1G内核空间

静态区域：

text segment(代码段):包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

data segment(数据段)：存储程序中已初始化的全局变量和静态变量

bss segment：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为0的全局变量和静态变量，对于未初始化的全局变量和静态变量，程序运行main之前时会统一清零。即未初始化的全局变量编译器会初始化为0

动态区域：

heap（堆）： 当进程未调用malloc时是没有堆段的，只有调用malloc时采用分配一个堆，并且在程序运行过程中可以动态增加堆大小(移动break指针)，从低地址向高地址增长。分配小内存时使用该区域。 堆的起始地址由mm_struct 结构体中的start_brk标识，结束地址由brk标识。

memory mapping segment(映射区):存储动态链接库等文件映射、申请大内存（malloc时调用mmap函数）

stack（栈）：使用栈空间存储函数的返回地址、参数、局部变量、返回值，从高地址向低地址增长。在创建进程时会有一个最大栈大小，Linux可以通过ulimit命令指定。

请你回答一下如何判断内存泄漏?

内存泄漏通常是由于调用了malloc/new等内存申请的操作，但是缺少了对应的free/delete。为了判断内存是否泄露，我们一方面可以使用linux环境下的内存泄漏检查工具Valgrind,mtrace检测。另一方面我们在写代码时可以添加内存申请和释放的统计功能，统计当前申请和释放的内存是否一致，以此来判断内存是否泄露。

请你来说一下什么时候会发生段错误

段错误通常发生在访问非法内存地址的时候，具体来说分为以下几种情况：

使用野指针

试图修改字符串常量的内容

请你来回答一下什么是memory leak，也就是内存泄漏

内存泄漏(memory leak)是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的分类：

1. 堆内存泄漏 （Heap leak）。对内存指的是程序运行中根据需要分配通过malloc,realloc new等从堆中分配的一块内存，再是完成后必须通过调用对应的 free或者delete 删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生Heap Leak.
2. 系统资源泄露（Resource Leak）。主要指程序使用系统分配的资源比如 Bitmap,handle ,SOCKET等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。
3. 没有将基类的析构函数定义为虚函数。当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确是释放，因此造成内存泄露。

请你来回答一下new和malloc的区别

1、new分配内存按照数据类型进行分配，malloc分配内存按照指定的大小分配；

2、new返回的是指定对象的指针，而malloc返回的是void*，因此malloc的返回值一般都需要进行类型转化。

3、new不仅分配一段内存，而且会调用构造函数，malloc不会。

4、new分配的内存要用delete销毁，malloc要用free来销毁；delete销毁的时候会调用对象的析构函数，而free则不会。

5、new是一个操作符可以重载，malloc是一个库函数。

6、malloc分配的内存不够的时候，可以用realloc扩容。扩容的原理？new没用这样操作。

7、new如果分配失败了会抛出bad_malloc的异常，而malloc失败了会返回NULL。

8、申请数组时： new[]一次分配所有内存，多次调用构造函数，搭配使用delete[]，delete[]多次调用析构函数，销毁数组中的每个对象。而malloc则只能sizeof(int) * n。

请你来说一下共享内存相关api

Linux允许不同进程访问同一个逻辑内存，提供了一组API，头文件在sys/shm.h中。

1）新建共享内存shmget

int shmget(key_t key,size_t size,int shmflg);

key：共享内存键值，可以理解为共享内存的唯一性标记。

size：共享内存大小

shmflag：创建进程和其他进程的读写权限标识。

返回值：相应的共享内存标识符，失败返回-1

2）连接共享内存到当前进程的地址空间shmat

void *shmat(int shm_id,const void *shm_addr,int shmflg);

shm_id：共享内存标识符

shm_addr：指定共享内存连接到当前进程的地址，通常为0，表示由系统来选择。

shmflg：标志位

返回值：指向共享内存第一个字节的指针，失败返回-1

3）当前进程分离共享内存shmdt

int shmdt(const void *shmaddr);

4）控制共享内存shmctl

和信号量的semctl函数类似，控制共享内存

int shmctl(int shm_id,int command,struct shmid_ds *buf);

shm_id：共享内存标识符

command: 有三个值

IPC_STAT:获取共享内存的状态，把共享内存的shmid_ds结构复制到buf中。

IPC_SET:设置共享内存的状态，把buf复制到共享内存的shmid_ds结构。

IPC_RMID:删除共享内存

buf：共享内存管理结构体。

请你来说一下reactor模型组成

参考网址

反应式编程

反应式编程介绍

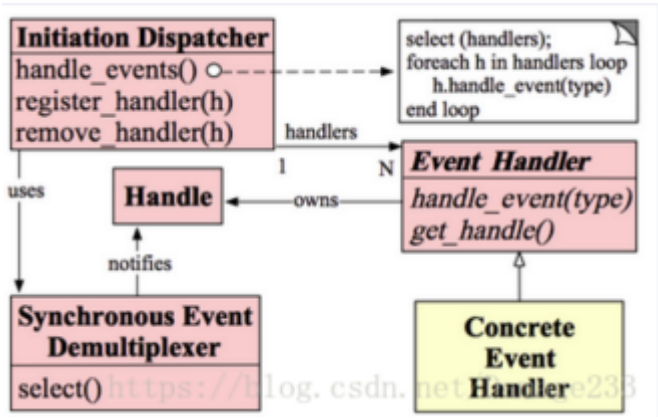
反应式编程来源于数据流和变化的传播，意味着由底层的执行模型负责通过数据流来自动传播变化。比如求值一个简单的表达式 c=a+b，当 a 或者 b 的值发生变化时，传统的编程范式需要对 a+b 进行重新计算来得到 c 的值。如果使用反应式编程，当 a 或者 b 的值发生变化时，c 的值会自动更新。反应式编程最早由 .NET 平台上的 Reactive Extensions (Rx) 库来实现。后来迁移到 Java 平台之后就产生了著名的 RxJava 库，并产生了很多其他编程语言上的

对应实现。在这些实现的基础上产生了后来的反应式流（Reactive Streams）规范。该规范定义了反应式流的相关接口，并将集成到 Java 9 中。

在传统的编程范式中，我们一般通过迭代器（Iterator）模式来遍历一个序列。这种遍历方式是由调用者来控制节奏的，采用的是拉的方式。每次由调用者通过 next()方法来获取序列中的下一个值。使用反应式流时采用的则是推的方式，即常见的发布者-订阅者模式。当发布者有新的数据产生时，这些数据会被推送到订阅者来进行处理。在反应式流上可以添加各种不同的操作来对数据进行处理，形成数据处理链。这个以声明式的方式添加的处理链只在订阅者进行订阅操作时才会真正执行。

反应式流中第一个重要概念是负压（backpressure）。在基本的消息推送模式中，当消息发布者产生数据的速度过快时，会使得消息订阅者的处理速度无法跟上产生的速度，从而给订阅者造成很大的压力。当压力过大时，有可能造成订阅者本身的奔溃，所产生的级联效应甚至可能造成整个系统的瘫痪。负压的作用在于提供一种从订阅者到生产者的反馈渠道。订阅者可以通过 request()方法来声明其一次所能处理的消息数量，而生产者就只会产生相应数量的消息，直到下一次 request()方法调用。这实际上变成了推拉结合的模式。

reactor模型要求主线程只负责监听文件描述上是否有事件发生，有的话就立即将该事件通知工作线程，除此之外，主线程不做任何其他实质性的工作，读写数据、接受新的连接以及处理客户请求均在工作线程中完成。其模型组成如下：



1）Handle：即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer等。由于Reactor模式一般使用在网络编程中，因而这里一般指Socket Handle，即一个网络连接。

2）Synchronous Event Demultiplexer（同步事件复用器）：阻塞等待一系列的Handle中的事件到来，如果阻塞等待返回，即表示在返回的Handle中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的select来实现。

3）Initiation Dispatcher：用于管理Event Handler，即EventHandler的容器，用以注册、移除EventHandler等；另外，它还作为Reactor模式的入口调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handle将其分发给对应的Event Handler处理，即回调EventHandler中的handle_event()方法。

4）Event Handler：定义事件处理方法：handle_event()，以供InitiationDispatcher回调使用。

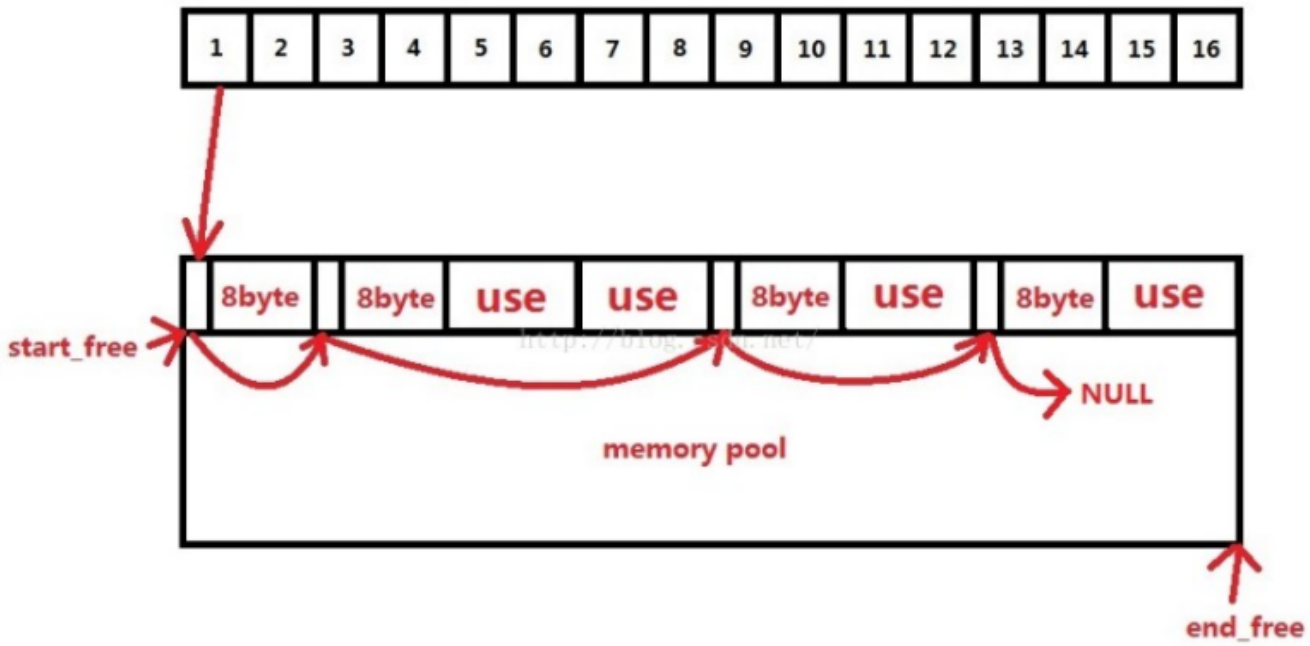
5）Concrete Event Handler：事件EventHandler接口，实现特定事件处理逻辑。

请自己设计一下如何采用单线程的方式处理高并发

在单线程模型中，可以采用I/O复用来提高单线程处理多个请求的能力，然后再采用事件驱动模型，基于异步回调来处理事件

请你说一说C++ STL 的内存优化

1）二级配置器结构 STL内存管理使用二级内存配置器。 1、第一级配置器 第一级配置器以malloc()，free()，realloc()等C函数执行实际的内存配置、释放、重新配置等操作，并且能在内存需求不被满足的时候，调用一个指定的函数。 一级空间配置器分配的是大于128字节的空间 如果分配不成功，调用句柄释放一部分内存 如果还不能分配成功，抛出异常 2、第二级配置器 在STL的第二级配置器中多了一些机制，避免太多小区块造成的内存碎片，小额区块带来的不仅是内存碎片，配置时还有额外的负担。区块越小，额外负担所占比例就越大。 3、分配原则 如果要分配的区块大于128bytes，则移交给第一级配置器处理。 如果要分配的区块小于128bytes，则以内存池管理（memory pool），又称之次层配置（sub-allocation）：每次配置一大块内存，并维护对应的16个空闲链表（free-list）。下次若有相同大小的内存需求，则直接从free-list中取。如果有小额区块被释放，则由配置器回收到free-list中。 当用户申请的空间小于128字节时，将字节数扩展到8的倍数，然后在自由链表中查找对应大小的子链表 如果在自由链表查找不到或者块数不够，则向内存池进行申请，一般一次申请20块 如果内存池空间足够，则取出内存 如果不够分配20块，则分配最多的块数给自由链表，并且更新每次申请的块数 如果一块都无法提供，则把剩余的内存挂到自由链表，然后向系统heap申请空间，如果申请失败，则看看自由链表还有没有可用的块，如果也没有，则最后调用一级空间配置器 2）二级内存池 二级内存池采用了16个空闲链表，这里的16个空闲链表分别管理大小为8、16、24……120、128的数据块。这里空闲链表节点的设计十分巧妙，这里用了一个联合体既可以表示下一个空闲数据块（存在于空闲链表中）的地址，也可以表示已经被用户使用的数据块（不存在空闲链表中）的地址。



1、空间配置函数allocate 首先先要检查申请空间的大小，如果大于128字节就调用第一级配置器，小于128字节就检查对应的空闲链表，如果该空闲链表中有可用数据块，则直接拿来用（拿取空闲链表中的第一个可用数据块，然后把该空闲链表的地址设置为该数据块指向的下一个地址），如果没有可用数据块，则调用refill重新填充空间。 2、空间释放函数deallocate 首先先要检查释放数据块的大小，如果大于128字节就调用第一级配置器，小于128字节则根据数据块的大小来判断回收后的空间会被插入到哪个空闲链表。 3、重新填充空闲链表refill 在用allocate配置空间时，如果空闲链表中没有可用数据块，就会调用refill来重新填充空间，新的空间取自内存池。缺省取20个数据块，如果内存池空间不足，那么能取多少个节点就取多少个。 从内存池取空间给空闲链表用是chunk_alloc的工作，首先根据end_free-start_free来判断内存池中的剩余空间是否足以调出nobjs个大小为size的数据块出去，如果内存连一个数据块的空间都无法供应，需要用malloc取堆中申请内存。 假如山穷水尽，整个系统的堆空间都不够用了，malloc失败，那么chunk_alloc会从空闲链表中找是否有大的数据块，然后将该数据块的空间分给内存池（这个数据块会从链表中去除）。 3、总结：

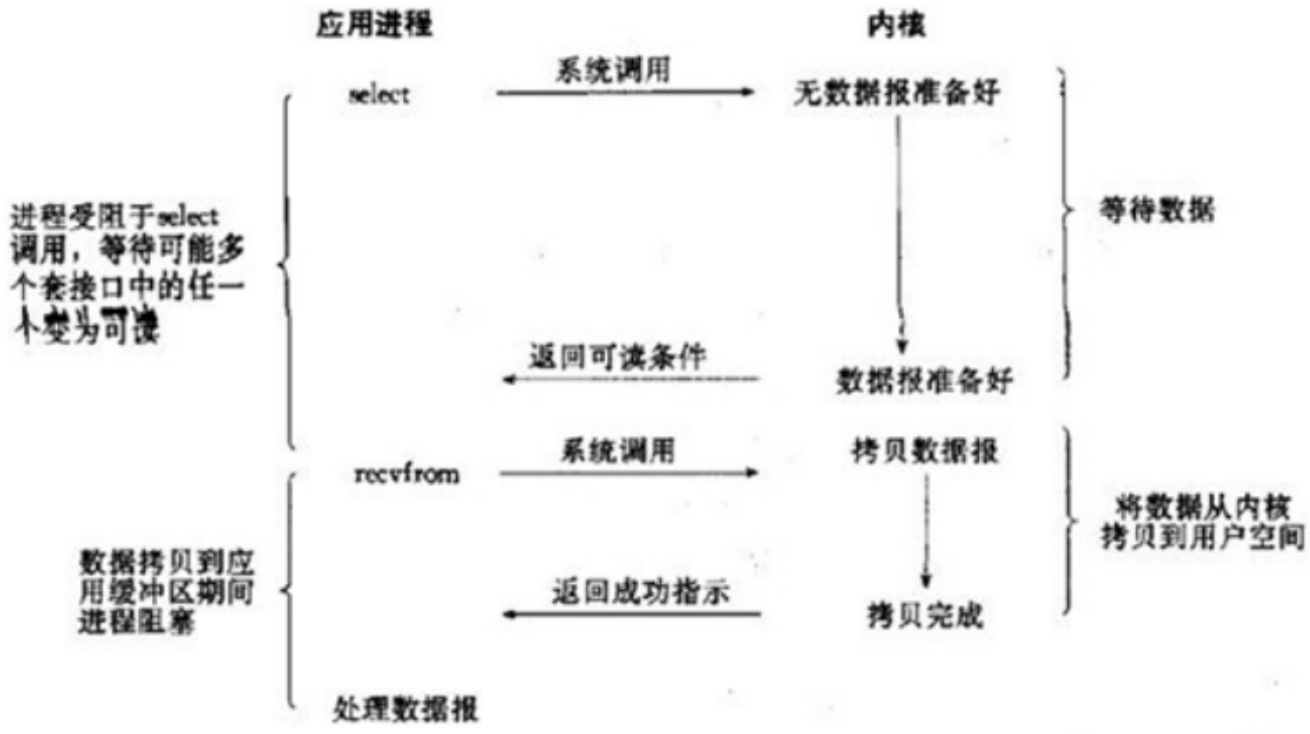
1. 使用allocate向内存池请求size大小的内存空间，如果需要请求的内存大小大于128bytes，直接使用malloc。
2. 如果需要的内存大小小于128bytes，allocate根据size找到最适合的自由链表。 a. 如果链表不为空，返回第一个node，链表头改为第二个node。 b. 如果链表为空，使用blockAlloc请求分配node。 x. 如果内存池中有大于一个node的空间，分配竟可能多的node(但是最多20个)，将一个node返回，其他的node添加到链表中。 y. 如果内存池只有一个node的空间，直接返回给用户。 z. 若果如果连一个node都没有，再次向操作系统请求分配内存。 ①分配成功，再次进行b过程。 ②分配失败，循环各个自由链表，寻找空间。 I. 找到空间，再次进行过程b。 II. 找不到空间，抛出异常。
3. 用户调用deallocate释放内存空间，如果要求释放的内存空间大于128bytes，直接调用free。
4. 否则按照其大小找到合适的自由链表，并将其插入。

请你说说select，epoll的区别，原理，性能，限制都说一说

1）IO多路复用

IO复用模型在阻塞IO模型上多了一个select函数，select函数有一个参数是文件描述符集合，意思就是对这些的文件描述符进行循环监听，当某个文件描述符就绪的时候，就对这个文件描述符进行处理。

这种IO模型是属于阻塞的IO。但是由于它可以对多个文件描述符进行阻塞监听，所以它的效率比阻塞IO模型高效。



IO多路复用就是我们说的select，poll，epoll。select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select，poll，epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。

当用户进程调用了select，那么整个进程会被block，而同时，kernel会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

所以，I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

I/O多路复用和阻塞I/O其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。

所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

2、select

select：是最初解决IO阻塞问题的方法。用结构体fd_set来告诉内核监听多个文件描述符，该结构体被称为描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义中。通过轮寻来查找是否有描述符要被处理。

存在的问题：

1. 内置数组的形式使得select的最大文件数受限与FD_SIZE；
2. 每次调用select前都要重新初始化描述符集，将fd从用户态拷贝到内核态，每次调用select后，都需要将fd从内核态拷贝到用户态；
3. 轮寻排查当文件描述符个数很多时，效率很低；

3、poll

poll：通过一个可变长度的数组解决了select文件描述符受限的问题。数组中元素是结构体，该结构体保存描述符的信息，每增加一个文件描述符就向数组中加入一个结构体，结构体只需要拷贝一次到内核态。poll解决了select重复初始化的问题。轮寻排查的问题未解决。

4、epoll

epoll：轮寻排查所有文件描述符的效率不高，使服务器并发能力受限。因此，epoll采用只返回状态发生变化的文件描述符，便解决了轮寻的瓶颈。

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式

1. LT模式

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的。

2. ET模式

ET(edge-triggered)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

3、LT模式与ET模式的区别如下：

LT模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll_wait时，会再次响应应用程序并通知此事件。 ET模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll_wait时，不会再次响应应用程序并通知此事件。