一、实验目的与要求：

实验目的

加深对内存分配与使用操作的直观认识；
掌握 Linux 操作系统的内存分配与使用的编程接口；
了解 Linux 操作系统中进程的逻辑编程地址和物理地址间的映射；

实验要求

可以使用 Linux 或其它 Unix 类操作系统；
学习该操作系统提供的分配、释放的函数使用方法；
学习该操作系统提供的进程地址映射情况的工具；

二、方法、步骤：(说明程序相关的算法原理或知识内容，程序设计的思路和方法，可以用流程图表述，程序主要数据结构的设计、主要函数之间的调用关系等)

操作部分（参考）：

- 借助 google 工具查找资料，学习使用 Linux 进程的内存分配、释放函数；(10 分)
- 借助 google 工具查找资料，学习 Linux proc 文件系统中关于内存影射的部分内容（了解/proc/pid/目录下的 maps、status、smap 等几个文件内部信息的解读）；(10 分)
- 编写程序，连续申请分配六个 128MB 空间（记为 1~6 号），然后释放第 2、3、5 号的 128MB 空间。然后再分配 1024MB，记录该进程的虚存空间变化（/proc/pid/maps），每次操作前后检查/proc/pid/status 文件中关于内存的情况，简要说明虚拟内存变化情况。推测此时再分配 64M 内存将出现在什么位置，实测后是否和你的预测一致？解释说明用户进程空间分配属于课本中的离散还是连续分配算法？首次适应还是最佳适应算法？用户空间存在碎片问题吗？(40 分)
- 设计一个程序测试出你的系统单个进程所能分配到的最大虚拟内存空间为多大。(20 分)
- 编写一个程序，分配 256MB 内存空间（或其他足够大的空间），检查分配前后/proc/pid/status 文件中关于虚拟内存和物理内存的使用情况，然后每隔 4KB 间隔将相应地址进行写操作，再次检查/proc/pid/status 文件中关于内存的情况，对比前后两次内存情况，说明所分配物理内存（物理内存块）的变化。然后重复上面操作，不过此时为读操作，再观察其变化。(20 分)

提高部分：（选做）

- 编写并运行（在第 5 步的程序未退出前）另一进程，分配等于或大于物理内存的空间，然后每隔 4KB 间隔将相应地址的字节数值增 1，此时再查看前一个程序的

物理内存变化，观察两个进程竞争物理内存的现象。

- 分配足够大的内存空间，其容量超过系统现有的空闲物理内存的大小，1）按 4KB 的间隔逐个单元进行写操作，重复访问数遍（使得程序运行时间可测量）；2）与前面访问总量和次数不变，但是将访问分成 16 个连续页为一组，逐组完成访问，记录运行时间。观察系统的状态，比较两者运行时间，给出判断和解释。

三．实验过程及内容：(对程序代码进行说明和分析，越详细越好，代码排版要整齐，可读性要高)

1. 学习使用 Linux 进程的内存分配、释放函数；(10 分)

C 和 C++程序的内存分配函数及对应释放函数：

realloc() / malloc() / calloc() / free()

三个函数的申明分别是: 它们的返回值都是请求系统分配的地址,如果请求失败就返回 NULL

```
1.  void* realloc(void* ptr, unsigned newsize);
2.  void* malloc(unsigned size);
3.  void* calloc(size_t numElements, size_t sizeOfElement);
```

malloc 用于申请一段新的地址,参数 size 为需要内存空间的长度,如:

```
1.  char* p;
2.  p=(char*)malloc(20);
```

calloc 与 malloc 相似,参数 sizeOfElement 为申请地址的单位元素长度,numElements 为元素个数,如:

```
1.  char* p;
2.  p=(char*)calloc(20,sizeof(char));
```

realloc 是给一个已经分配了地址的指针重新分配空间,参数 ptr 为原有的空间地址,newsize 是重新申请的地址长度

```
1.  char* p;
2.  p=(char*)malloc(sizeof(char)*20);
3.  p=(char*)realloc(p,sizeof(char)*40);
```

free 的调用形式为 free(void*ptr)：释放 ptr 所指向的一块内存空间。

区别:

(1)函数 malloc 不能初始化所分配的内存空间,而函数 calloc 能.如果由 malloc()函数分配的内存空间原来没有被使用过，则其中的每一位可能都是 0;反之，如果这部分内存曾经被分配过,则其中可能遗留有各种各样的数据.也就是说，使用 malloc()函数的程序开始时(内存空间还没有被重新分配)能正常进行,但经过一段时间(内存空间还已经被重新分配)可能会出现问题.

(2)函数 calloc() 会将所分配的内存空间中的每一位都初始化为零,也就是说,如果你是为字符类型或整数类型的元素分配内存,那么这些元素将保证会被初始化为 0;如果你是为指针类型的元素分配内存,那么这些元素通常会被初始化为空指针;如果你为实型数据分配内存,则这些元素会被初始化为浮点型的零.

(3)函数 malloc 向系统申请分配指定 size 个字节的内存空间.返回类型是 void*类型.void*表示未确定类型的指针.C,C++规定，void* 类型可以强制转换为任何其它类型的指针.

(4)realloc 可以对给定的指针所指的空间进行扩大或者缩小，无论是扩张或是缩小，原有内存的中内容将保持不变.当然，对于缩小，则被缩小的那一部分的内容会丢失.realloc 并不保证

调整后的内存空间和原来的内存空间保持同一内存地址.相反，realloc 返回的指针很可能指向一个新的地址.

(5)realloc 是从堆上分配内存的.当扩大一块内存空间时，realloc()试图直接从堆上现存的数据后面的那些字节中获得附加的字节，如果能够满足，自然天下太平；如果数据后面的字节不够，问题就出来了，那么就使用堆上第一个有足够大小的自由块，现存的数据然后就被拷贝至新的位置，而老块则放回到堆上.这句话传递的一个重要的信息就是数据可能被移动.

- **new / delete**

new 和 delete 是 C++中的运算符，不是库函数，不需要库的支持，同时，他们是封装好的重载运算符，并且可以再次进行重载。

（1）new 是动态分配内存的运算符，自动计算需要分配的空间，在 C++中，它属于重载运算符，可以对多种数据类型形式进行分配内存空间，比如 int 型、char 型、结构体型和类等的动态申请的内存分配，分配类的内存空间时，同时调用类的构造函数，对内存空间进行初始化，即完成类的初始化工作。

（2）delete 是撤销动态申请的内存运算符。delete 与 new 通常配对使用，与 new 的功能相反，可以对多种数据类型形式的内存进行撤销，包括类，撤销类的内存空间时，它要调用其析构函数，完成相应的清理工作，收回相应的内存资源。

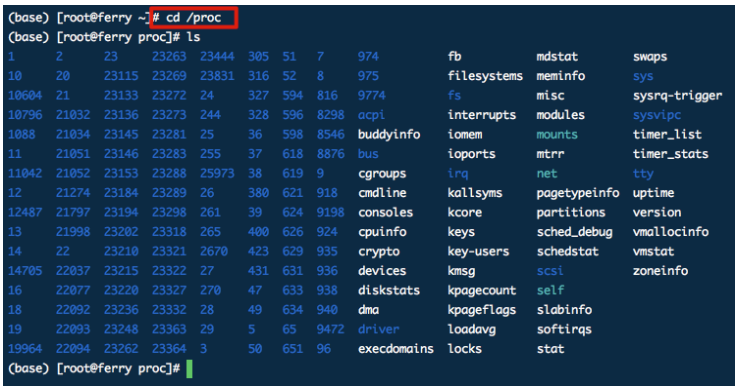
（3）典型用法

```
1.  int *p = new int;          delete p;
2.  char *p = new char;        delete p;
3.  类的类型 *p = new 类的类型;  delete p;
4.  //注意，指针 p 存于栈中，p 所指向的内存空间却是在堆中。
5.  Obj * p = new Obj[100];    delete [ ]p;
6.  //注意，new 申请数组，delete 删除的形式需要加括号“[ ]”，表示对数组空间的操作，
    总之，申请形式如何，释放的形式就如何。
```

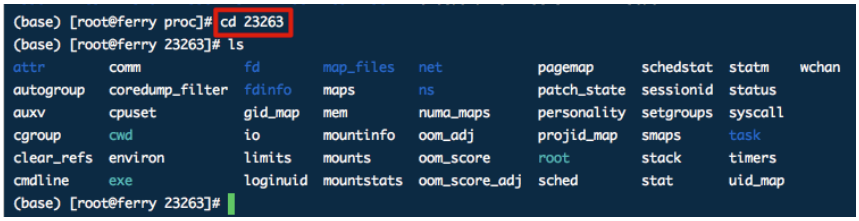
2. 学习 Linux proc 文件系统中关于内存影射的部分内容 (10 分)

Linux 内核提供了一种通过 proc 文件系统，在运行时访问内核内部数据结构、改变内核设置的机制。proc 文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过 proc 得到系统的信息，并可以改变内核的某些参数。由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取 proc 文件时，proc 文件系统是动态从系统内核读出所需信息并提交的。

/proc 目录下，每个进程都会有一个以 pid 命名的目录，存放这个进程的信息，同时还有许多相对固定的目录，以存放与系统相关的信息；



以其中一个进程为例，打开进程文件目录，可以看到其中包含许多文件和目录，比如 exe 文件（进程相关的可执行文件）、root（指向进程根目录）、maps（可执行文件和库文件图）、status（进程状态）、smaps（maps 扩展，显示 mapping 的内存）等：



(1) cat /proc/[pid]/maps 显示进程映射了的内存区域和访问权限：

该文件有 6 列，每项都与一个 vm_area_struct 结构成员对应,分别为：

地址：库在进程里地址范围

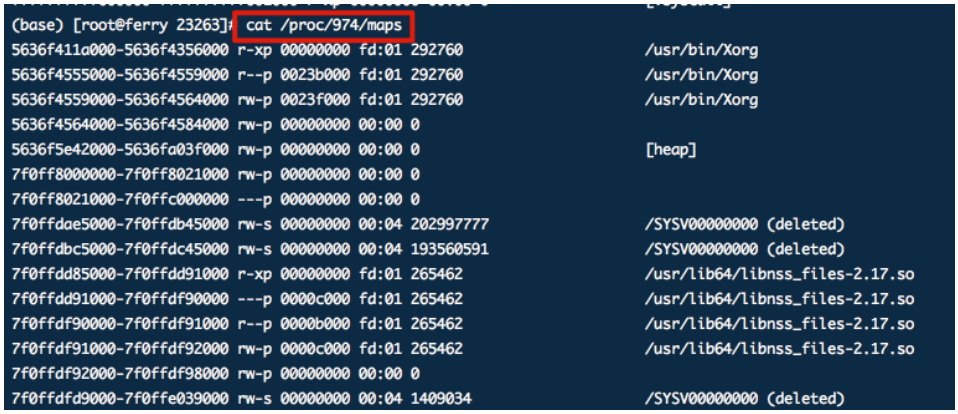
权限：虚拟内存的权限，r=读，w=写,x=,s=共享,p=私有；

偏移量：库在进程里地址范围

设备：映像文件的主设备号和次设备号；

节点：映像文件的节点号；

路径: 映像文件的路径



(2) cat /proc/[pid]/status 显示进程状态，占用虚拟内存情况，占用物理内存情况等等信息,包含了所有 CPU 活跃的信息，该文件中的所有值都是从系统启动开始累计到当前时刻:

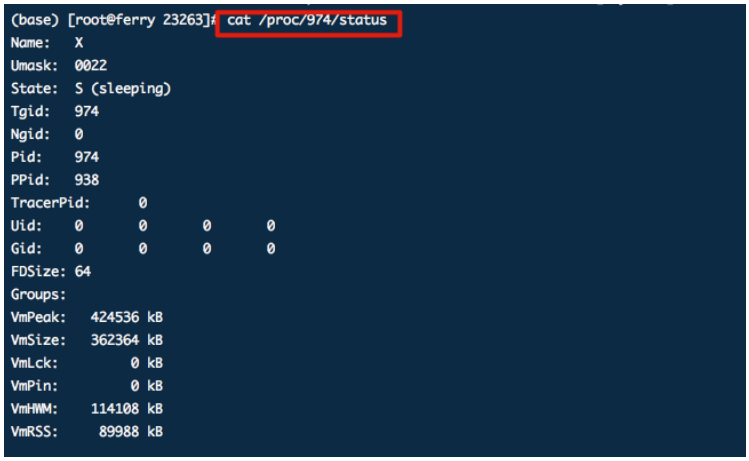
Name 应用程序或命令的名字

State 任务的状态，运行/睡眠/僵死/

SleepAVG 任务的平均等待时间(以 nanosecond 为单位)，交互式任务因为休眠次数多、时间长，它们的 sleep_avg 也会相应地更大一些，所以计算出来的优先级也会相应高一些。

Tgid 线程组号

Pid 任务 ID
Ppid 父进程 ID
TracerPid 接收跟踪该进程信息的进程的 ID 号
Uid Uid euid suid fsuid
Gid Gid egid sgid fsgid
FDSize 文件描述符的最大个数，file->fds
VmSize(KB) 任务虚拟地址空间的大小 (total_vm-reserved_vm)，其 total_vm 为进程的地址空间的大小，
reserved_vm：进程在预留或特殊的内存间的物理页
VmLck(KB) 任务已经锁住的物理内存的大小。锁住的物理内存不能交换到硬盘 (locked_vm)
VmRSS(KB) 应用程序正在使用的物理内存的大小，就是用 ps 命令的参数 rss 的值 (rss)
VmData(KB) 程序数据段的大小（所占虚拟内存的大小），存放初始化了的数据；(total_vm-shared_vm-stack_vm)
VmStk(KB) 任务在用户态的栈的大小 (stack_vm)
VmExe(KB) 程序所拥有的可执行虚拟内存的大小，代码段，不包括任务使用的库 (end_code-start_code)
VmLib(KB) 被映像到任务的虚拟内存空间的库的大小 (exec_lib)
VmPTE 该进程的所有页表的大小，单位：kb
Threads 共享使用该信号描述符的任务的个数，在 POSIX 多线程程序应用程序中，线程组中的所有线程使用同一个信号描述符。



3.编写程序，连续申请分配六个 128MB 空间（记为 1~6 号），然后释放第 2、3、5 号的 128MB 空间。然后再分配 1024MB，记录该进程的虚存空间变化（/proc/pid/maps），每次操作前后检查/proc/pid/status 文件中关于内存的情况，简要说明虚拟内存变化情况。推测此时再分配 64M 内存将出现在什么位置，实测后是否和你的预测一致？解释说明用户进程空间分配属于课本中的离散还是连续分配算法？首次适应还是最佳适应算法？用户空间存在碎片问题吗？（40 分）

（0）代码分析
首先用 getpid（）获取进程 ID

```
1.  printf("进程：%d 开启\n",getpid() );
```

```
    getchar();
```

然后定义 6 个指针，用 malloc 分配内存，用 pow（2，25）表示 128MM，指针不为空，则说明分配成功，输出提示。

```
    int *a,*b,*c,*d,*e,*f,*g,*h;
    a=(int *)malloc(pow(2,25)*sizeof(int));
    b=(int *)malloc(pow(2,25)*sizeof(int));
    c=(int *)malloc(pow(2,25)*sizeof(int));
    d=(int *)malloc(pow(2,25)*sizeof(int));
    e=(int *)malloc(pow(2,25)*sizeof(int));
    f=(int *)malloc(pow(2,25)*sizeof(int));
    if(a!=NULL&&b!=NULL&&c!=NULL&&d!=NULL&&e!=NULL&&f!=NULL)
        printf("分配了 6 个 128M 的内存空间\n");
    else{
        printf("分配 6 个 128M 的内存空间失败\n");
    }
    getchar();
```

依次释放 2 号、3 号、5 号内存空间

```
    free(b);
    printf("释放了 2 号内存空间\n");
    getchar();
    free(c);
    printf("释放了 3 号内存空间\n");
    getchar();
    free(e);
    printf("释放了 5 号内存空间\n");
    getchar();
```

再用 malloc 分配 1024M 和 64M 内存空间

```
g=(int *)malloc(pow(2,28)*sizeof(int));
    if(g!=NULL){
        printf("分配了 1024M 的内存空间\n");
    }
    else{
        printf("分配 1024M 内存空间失败\n");
    }
    getchar();

    h=(int *)malloc(pow(2,24)*sizeof(int));
    if(h!=NULL){
        printf("分配了 64M 的内存空间\n");
```

```
    }
    else{
        printf("分配 64M 内存空间失败\n");
    }
    getchar();
```

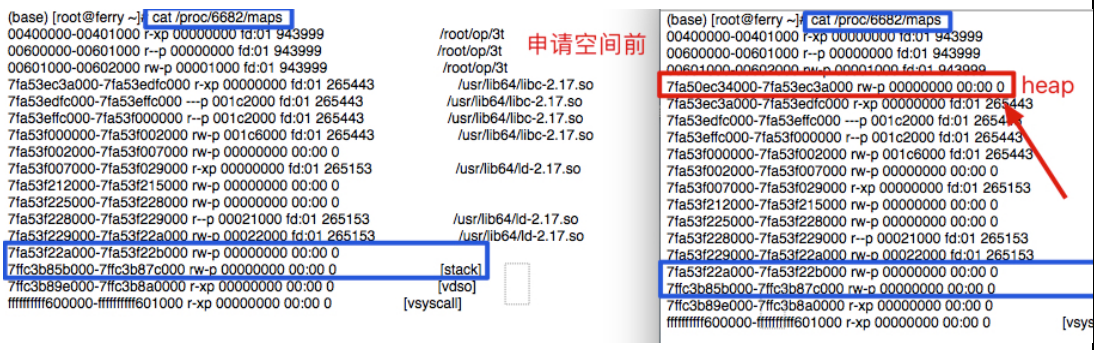
最后把仍存在的内存用 free 释放

```
free(a);
    free(d);
    free(f);
    free(g);
    free(h);
```

（1）连续申请分配六个 128MB 空间（记为 1~6 号）前后 maps 文件信息对比如下图所示。
首先我们需要弄清楚 heap（堆）和 stack（栈）区别 ：stack 的空间由操作系统自动分配和释放，heap 的空间是手动申请和释放的，heap 一般是由 malloc 申请的内存，使用 free 释放，但其实如果没有主动释放，在进程运行结束时也会被释放。按照常理，在堆 heap 后面应该有个 [heap]的提示，但可能由于系统原因，这里并没有 M 显示，不过不影响分析，地址空间就是这样的。

这里可以看到，在申请分配 6 个 128M 文件之前，maps 文件信息里面只有 stack 地址，没有 heap 地址，但当申请分配空间之后，malloc 分配的内存空间虚拟地址存在于 heap 中，这里说明分配的虚拟内存存于（7fa50ec34000-7fa53ec3a000）地址中，即为 768M = 6*128M，与实际相符合。即

----------连续分配 6 个 128MB 空间（总 768M）--------
1-6 号：7fa50ec34000-7fa53ec3a000(768M) ——> 存在



（2）连续申请分配六个 128MB 空间（记为 1~6 号）前后 status 文件信息对比如下图所示。
从 status 文件信息中，可以看到：
VmPeak 进程地址空间增加（790672-4280）= 786392KB = 6*128MB - 40KB，

VmSize 进程虚拟地址空间的大小增加了（790672-4216）= 786456KB = 6*128MB + 24KB，
VmHWM 程序得到分配到物理内存的峰值，文件内存映射和匿名内存映射的大小无变化，
VmRSS 应用程序正在使用的物理内存的大小无变化，
VmData 程序数据段的大小增加了（786512-56）KB = 786456KB = 6*128MB + 24KB，
VmPTE 该进程的页表的大小增加了 24KB，
voluntary_ctxt_swicthes 进程主动切换的次数增加了 1 次。

分配内存前

```
(base) [root@ferry ~]# cat /proc/6682/status
Name:   3t
Umask:  0022
State:  S (sleeping)
Tgid:   6682
Ngid:   0
Pid:    6682
PPid:   5270
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:     4280 kB
VmSize:     4216 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       352 kB
VmRSS:       352 kB
RssAnon:             76 kB
RssFile:            276 kB
RssShmem:             0 kB
VmData:       56 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        28 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     1
```

分配内存后

```
(base) [root@ferry ~]# cat /proc/6682/status
Name:   3t
Umask:  0022
State:  S (sleeping)
Tgid:   6682
Ngid:   0
Pid:    6682
PPid:   5270
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:   790672 kB
VmSize:   790672 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       352 kB
VmRSS:       352 kB
RssAnon:             76 kB
RssFile:            276 kB
RssShmem:             0 kB
VmData:   786512 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        52 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        3
nonvoluntary_ctxt_switches:     1
```

（3）释放 2 号内存空间后 maps 文件信息的变化,如下图所示。
可以看到原地址段（7fa50ec34000-7fa53ec3a000）(768M)一分为二，
其中一段为（7fa50ec34000-7fa52ec38000）(512M，为 3-6 号内存地址)，另一段是（7fa536c39000-7fa53ec3a000）(128M，为 1 号内存地址)，
所以可以推测 2 号内存的地址段为（7fa52ec38000-7fa536c39000）(128M)，

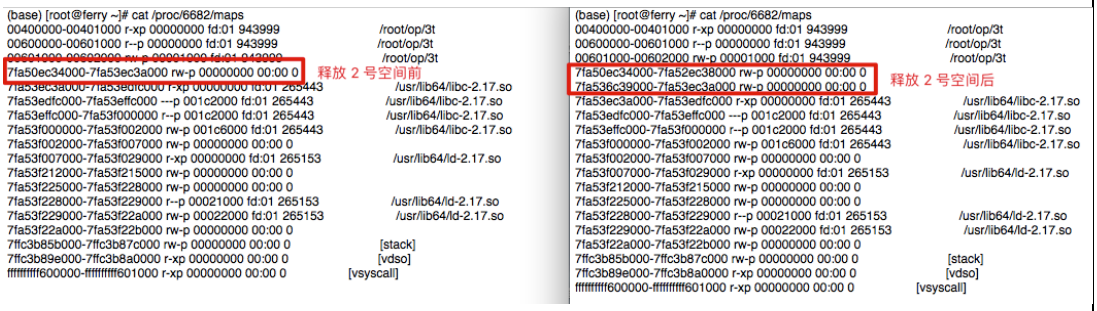
而 1 号内存地址段为（7fa536c39000-7fa53ec3a000）(128M)，这两个的地址段大小都是 128M。

----------释放 2 号内存空间（总 640M）----------

3-6 号：7fa50ec34000-7fa52ec38000 (512M) ——> 存在

2 号：7fa52ec38000-7fa536c39000（128M）——> 本次释放

1 号：7fa536c39000-7fa53ec3a000）(128M) ——> 存在



（4）释放 3 号内存空间后 maps 文件信息的变化,如下图所示。

可以看到地址段变成了（7fa50ec34000-7fa526c37000）(384M，为 4-6 号内存地址)和（7fa536c39000-7fa53ec3a000）(128M，为 1 号内存地址)，所以可以推测 3 号内存地址段为：（7fa526c37000-7fa52ec38000）（128M）

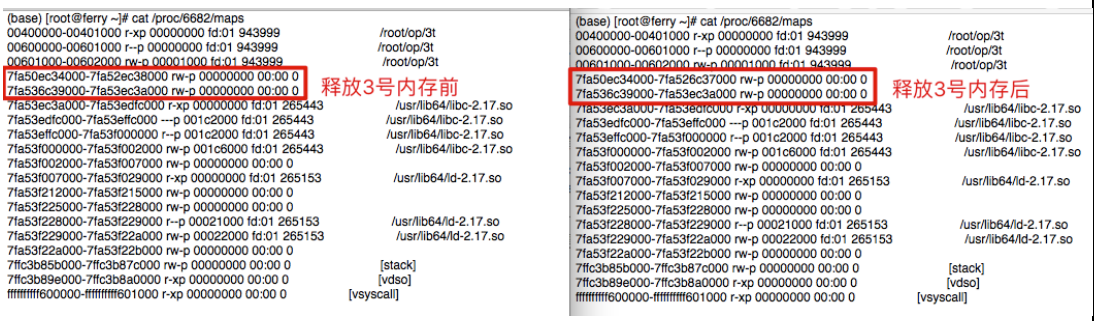
----------释放 3 号内存空间（总 512M）----------

4-6 号：7fa50ec34000-7fa526c37000 (384M) ——> 存在

3 号：7fa526c37000-7fa52ec38000（128M）——> 本次释放

2 号：7fa52ec38000-7fa536c39000（128M）——> 已释放

1 号：7fa536c39000-7fa53ec3a000）(128M) ——> 存在



（5）释放 5 号内存空间后 maps 文件信息的变化,如下图所示。

可以看到两段地址变成 3 段地址，其中 7fa50ec34000-7fa516c35000（128M，为 6 号内存地址）、7fa51ec36000-7fa526c37000 （128M，为 4 号内存地址）、7fa536c39000-7fa53ec3a000 （128M，为 1 号内存地址），所以可以推测本次释放的 5 号内存地址空间为 7fa516c35000-7fa51ec36000（128M）

----------释放 5 号内存空间（总 384M）----------

6 号：7fa50ec34000-7fa516c35000（128M） ——> 存在

5 号：7fa516c35000-7fa51ec36000（128M）——> 本次释放

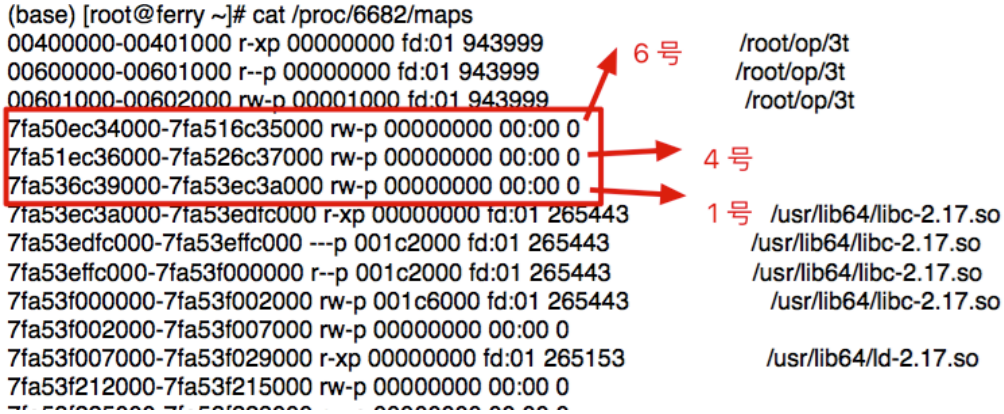
4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在

3 号：7fa526c37000-7fa52ec38000（128M）——> 已释放

2 号：7fa52ec38000-7fa536c39000（128M）——> 已释放

1 号：7fa536c39000-7fa53ec3a000）(128M) ——> 存在

（6） 可以看到最后剩下三段内存段（1,4,6 号内存空间）：



（7）再申请 1024M 内存空间，如下图 maps 文件信息，可以看到 6 号地址段变成了（7fa4cec33000-7fa516c35000），大小为 1152M，即（1024M+128M）：其他 1 号和 4 号两个内存地址段没有变化。

----------再分配 1024 内存空间(总 1536M)----------

6 号：7fa4cec33000-7fa516c35000(1152M=128M+1024M) ——> 存在（本次分配）

4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在

1 号：7fa536c39000-7fa53ec3a000 (128M) ——> 存在

（8）再申请 64M 内存空间，如下图 maps 文件信息，可以看到 1 号地址段变成了

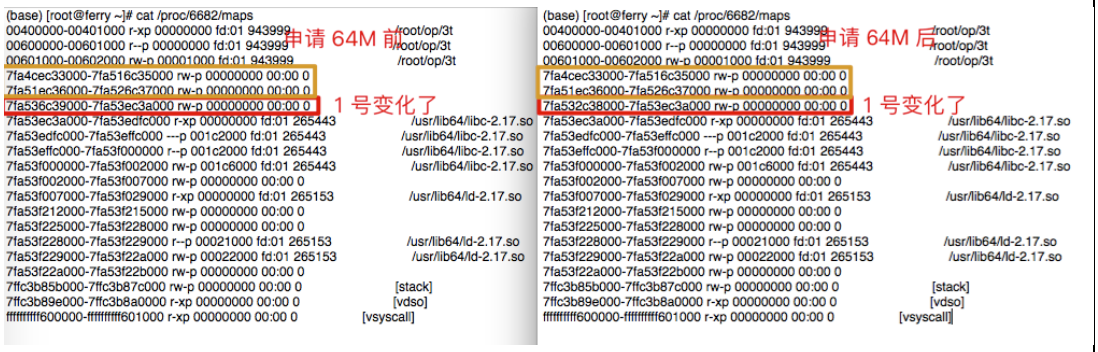
（7fa532c38000-7fa53ec3a000），大小为 192，即（64M+128M）

----------再分配 64M 内存空间(总 1600M)----------

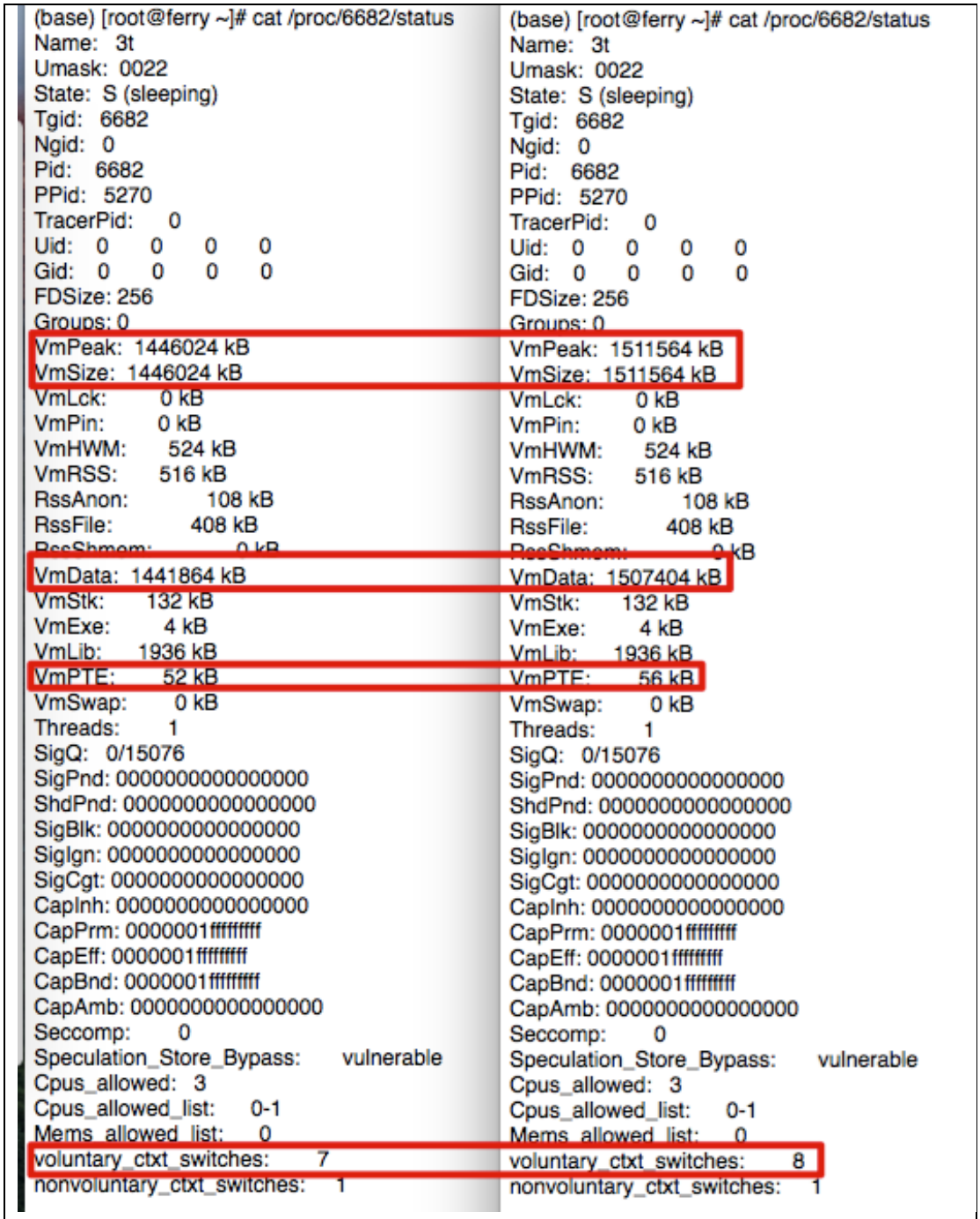
6 号：7fa4cec33000-7fa516c35000 (1152M) ——> 存在

4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在

1 号：7fa532c38000-7fa53ec3a000 (192M=128M+64M) ——> 存在（本次分配）



再次查看分配 64M 内存空间前后 status 文件信息的变化，可以看到结论和前面的分析结果是一致的。

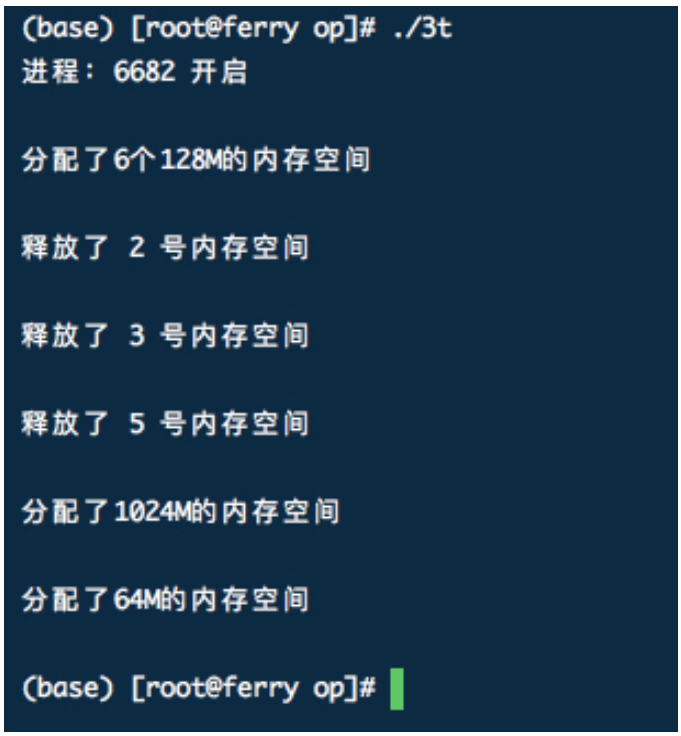


```
(base) [root@ferry ~]# cat /proc/6682/status
Name:   3t
Umask:  0022
State:  S (sleeping)
Tgid:   6682
Ngid:   0
Pid:    6682
PPid:   5270
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:  1446024 kB
VmSize:  1446024 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       524 kB
VmRSS:       516 kB
RssAnon:             108 kB
RssFile:             408 kB
RssShmem:              0 kB
VmData:  1441864 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        52 kB
VmSwap:        0 kB
Threads:       1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        7
nonvoluntary_ctxt_switches:     1
```

```
(base) [root@ferry ~]# cat /proc/6682/status
Name:   3t
Umask:  0022
State:  S (sleeping)
Tgid:   6682
Ngid:   0
Pid:    6682
PPid:   5270
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:  1511564 kB
VmSize:  1511564 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       524 kB
VmRSS:       516 kB
RssAnon:             108 kB
RssFile:             408 kB
RssShmem:              0 kB
VmData:  1507404 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        56 kB
VmSwap:        0 kB
Threads:       1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        8
nonvoluntary_ctxt_switches:     1
```

（9）总结/结论：

a) **虚拟内存变化**情况如上面的分析,总结如下；



----------连续分配 6 个 128MB 空间（总 768M）--------
1-6 号：7fa50ec34000-7fa53ec3a000(768M) ——> 存在
----------释放 2 号内存空间（总 640M）----------
3-6 号：7fa50ec34000-7fa52ec38000 (512M) ——> 存在
2 号：7fa52ec38000-7fa536c39000（128M）——> 本次释放
1 号：7fa536c39000-7fa53ec3a000 (128M) ——> 存在
----------释放 3 号内存空间（总 512M）----------
4-6 号：7fa50ec34000-7fa526c37000 (384M) ——> 存在
3 号：7fa526c37000-7fa52ec38000（128M）——> 本次释放
2 号：7fa52ec38000-7fa536c39000（128M）——> 已释放
1 号：7fa536c39000-7fa53ec3a000 (128M) ——> 存在
----------释放 5 号内存空间（总 384M）----------
6 号：7fa50ec34000-7fa516c35000（128M） ——> 存在
5 号：7fa516c35000-7fa51ec36000（128M）——> 本次释放
4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在
3 号：7fa526c37000-7fa52ec38000（128M）——> 已释放
2 号：7fa52ec38000-7fa536c39000（128M）——> 已释放
1 号：7fa536c39000-7fa53ec3a000(128M) ——> 存在
----------再分配 1024 内存空间(总 1536M)----------
6 号：7fa4cec33000-7fa516c35000(1152M=128M+1024M) ——> 存在（本次分配）
4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在
1 号：7fa536c39000-7fa53ec3a000 (128M) ——> 存在
----------再分配 64M 内存空间(总 1600M)----------
6 号：7fa4cec33000-7fa516c35000 (1152M) ——> 存在
4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在
1 号：7fa532c38000-7fa53ec3a000 (192M=128M+64M) ——> 存在（本次分配）

b) 用户进程空间分配属于**连续分配算法**

可以从实验结果看出，程序中代码或数据的逻辑地址是相邻连续的，即为用户程序分配的是一个连续的内存空间

c) **首次适应算法**

1024M 大于前面的两个 256M（被释放的 2 号和 3 号内存区）和 128M（被释放的 5 号内存区），实验结果得到的是该内存段最后被分配在 6 号内存区之后；64M 小于前面的两段 256M 和 128M，而且 256M 在 128M 前面，分配 64M 内存时选择了在 256M 内存碎片地址段而不是更小碎片的 128M，综上所述，选择的是首次适应算法。

d) **存在碎片问题**

连续分配算法，在释放中间的内存段时会将一条连续内存段切割成两条的独的内存段，被释放的的内存段形成碎片。

4.设计一个程序测试出你的系统单个进程所能分配到的最大虚拟内存空间为多大。(20 分)

（1）编写代码：

```
#include<unistd.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int main(){
    int i;
    char *p[100000];
    printf("进程：%d 开启\n",getpid() );
    getchar();
    for (i=0;i<100000;i++){
        p[i]=(char *)malloc(pow(2,30)*sizeof(char));//每次分配 1G 内存
        if(p[i]==NULL){
            printf("虚拟内存已消耗完\n");
            getchar();
            break;
        }
    }
    return 0;
}
```

（2）运行结果：

```
(base) [root@ferry op]# ./4t
进程： 31176 开启

虚拟内存已消耗完

(base) [root@ferry op]#
```

用 status 查看耗光内存前后的详细内存使用情况：如下图比较图，可以看出此时 VmPeak 进程地址空间大小增加了（16201745532-4880）KB = 15GB，同样的 VmSize 进程虚拟地址空间的大小增加（16201679996-4880）=15G 左右，那么说明当前的进程能分配的最大的虚拟空间是 15G。

```
(base) [root@ferry ~]# cat /proc/31176/status
Name:   4t
Umask:  0022
State:  S (sleeping)
Tgid:   31176
Ngid:   0
Pid:    31176
PPid:   27208
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:     4880 kB
VmSize:     4880 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       352 kB
VmRSS:       352 kB
RssAnon:              76 kB
RssFile:             276 kB
RssShmem:              0 kB
VmData:       56 kB
VmStk:       796 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        24 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:   00000000,00000000,00000000,00000000
0000,00000000,00000000,00000000,00000000,00000000,
00,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        1
nonvoluntary_ctxt_switches:     1
```

耗光内存前

```
(base) [root@ferry ~]# cat /proc/31176/status
Name:   4t
Umask:  0022
State:  S (sleeping)
Tgid:   31176
Ngid:   0
Pid:    31176
PPid:   27208
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak: 16201745532 kB
VmSize: 16201679996 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:     62444 kB
VmRSS:     62444 kB
RssAnon:             62028 kB
RssFile:               416 kB
RssShmem:              0 kB
VmData: 16201675172 kB
VmStk:       796 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:     61832 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:   00000000,00000000,00000000,00000000
0000,00000000,00000000,00000000,00000000,00000000,
00,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     39
(base) [root@ferry ~]#
```

耗光内存后

5.编写一个程序，分配 256MB 内存空间（或其他足够大的空间），检查分配前后/proc/pid/status 文件中关于虚拟内存和物理内存的使用情况，然后每隔 4KB 间隔将相应地址进行写操作，再次检查/proc/pid/status 文件中关于内存的情况，对比前后两次内存情况，说明所分配物理内存（物理内存块）的变化。然后重复上面操作，不过此时为读操作，再观察其变化。(20 分)

（1）编写代码：

```
#include<unistd.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int main(){

    int *p;
    printf("进程：%d 开启\n",getpid() );
    getchar();

    int i,sum=pow(2,26);
    p=(int *)malloc(sum*sizeof(int));//256M
    printf("申请分配 256M 内存空间\n");

    //相隔 4KB 进行读写  就是相隔 1024 个 int 进行读写
    printf("回车开始进行读操作\n");
    getchar();
    for (i = 0; i < sum; i+=1024)
    {
        p[i];
    }
    printf("读操作结束\n");
    getchar();

    printf("回车开始写操作\n");
    getchar();
    for (i = 0; i < sum; i+=1024)
    {
        p[i]=i;
    }
    printf("写操作结束\n");
    getchar();

    return 0;

}
```

(2)运行结果：

```
(base) [root@ferry op]# ./5t
进程：30739 开启

申请分配256M内存空间
回车开始进行读操作

读操作结束

回车开始写操作

写操作结束

(base) [root@ferry op]#
```

（3）在申请 256M 文件前后，通过 status 查看此时的内存情况，具体情况如下图所示，可以看出 VMSize（虚拟内存）增加了（266364-4216）KB = 256MB，而 VMRSS（程序正在使用的物理内存）不变，voluntary_ctxt_switches（进程主动切换次数）增加了 1 次：

```
(base) [root@ferry ~]# cat /proc/30739/status
Name:   5t
Umask:  0022
State:  S (sleeping)
Tgid:   30739
Ngid:   0
Pid:    30739
PPid:   29284
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:     4280 kB
VmSize:     4216 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       348 kB
VmRSS:       348 kB
RssAnon:              72 kB
RssFile:             276 kB
RssShmem:              0 kB
VmData:       56 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        24 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        4
nonvoluntary_ctxt_switches:     2
```

申请 256M 内存前

```
(base) [root@ferry ~]# cat /proc/30739/status
Name:   5t
Umask:  0022
State:  S (sleeping)
Tgid:   30739
Ngid:   0
Pid:    30739
PPid:   29284
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:   266364 kB
VmSize:   266364 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       348 kB
VmRSS:       348 kB
RssAnon:              72 kB
RssFile:             276 kB
RssShmem:              0 kB
VmData:   262204 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        28 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        5
nonvoluntary_ctxt_switches:     2
```

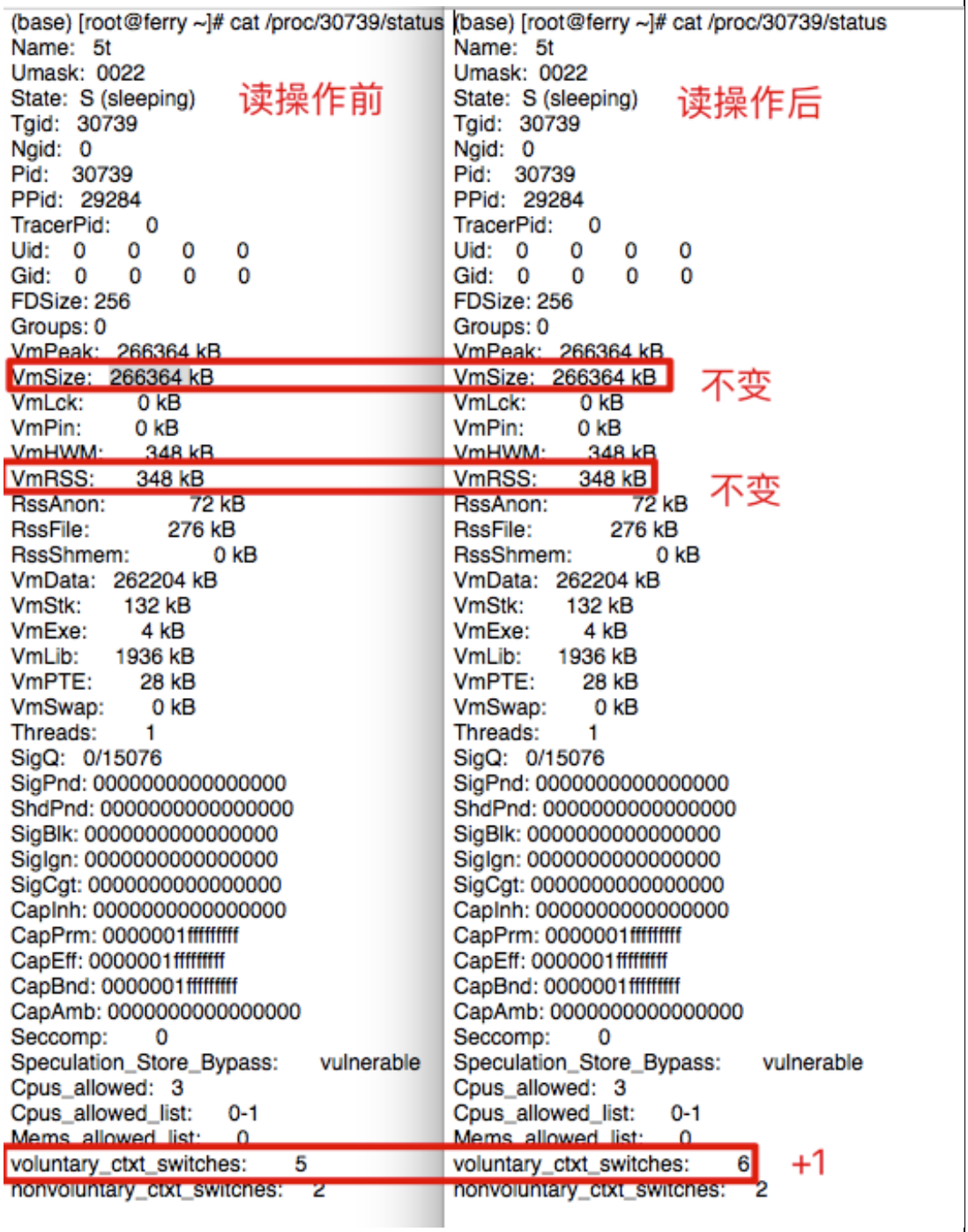
申请 256M 内存后

+256M

不变

+1

（4）读操作结束前后，调用 status 查看两者的内存情况，具体情况如下图所示，可以看出 VMSize（虚拟内存）不变，VMRSS（程序正在使用的物理内存）同样不变，voluntary_ctxt_switches（进程主动切换次数）增加了 1 次；



（5）写操作结束前后，调用 status 查看两者的内存情况，具体情况如下图所示，可以看出 VMSize（虚拟内存）不变，VMRSS（程序正在使用的物理内存）增加了（262496 -348）KB，约等于 256M，voluntary_ctxt_switches（进程主动切换次数）增加了 2 次，nonvoluntary_ctxt_switches（进程被动切换次数）增加了 128 次；

（6）结论： 上述分析结果说明了读操作不需要物理内存，写操作需要使用物理内存。

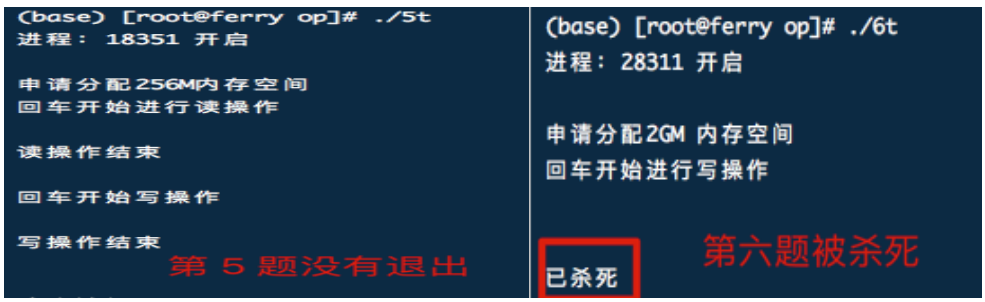
6.编写并运行（在第 5 步的程序未退出前）另一进程，分配等于或大于物理内存的空间，然后每隔 4KB 间隔将相应地址的字节数值增 1，此时再查看前一个程序的物理内存变化，观察两个进程竞争物理内存的现象。

(1)代码

```
1. #include<unistd.h>
2. #include<stdio.h>
3. #include<math.h>
4. #include<stdlib.h>
5. int main(){
6.
```

```
7.      int *p;
8.      printf("进程：%d 开启\n",getpid() );
9.      getchar();
10.
11.     int i,sum=pow(2,28);
12.     p=(int *)malloc(sum*sizeof(int));
13.     int *q=(int *)malloc(sum*sizeof(int));
14.     if(p!=NULL&&q!=NULL){
15.        printf("申请分配 1G 内存空间\n");
16.
17.        //相隔 4KB 进行读写  就是相隔 1024 个 int 进行读写
18.        printf("回车开始进行写操作\n");
19.        getchar();
20.        for (i = 0; i < sum; i+=1024)
21.        {
22.           p[i]=i;
23.        }
24.        for (i = 0; i < sum; i+=1024)
25.        {
26.           q[i]=i;
27.        }
28.        printf("写操作结束\n");
29.
30.        free(p);
31.        free(q);
32.
33.     }else{
34.        printf("分配内存失败\n");
35.     }
36.     getchar();
37.     return 0;
38. }
```

(2)如果先运行 5 题程序不退出，再运行 6 题程序，这里本来在第 6 题分配的物理空间是 2G，但是会由于读操作太频繁进程会被杀死。



所以修改代码，将读写操作只分配了 1GB（pow(2,28)*sizeof(int)）的内存

，并且运行程序：

```
(base) [root@ferry op]# ./5t
进程： 2732 开启

申请分配256M内存空间
回车开始进行读操作

读操作结束
```

```
(base) [root@ferry op]# ./6t
进程： 6398 开启

申请分配2GM 内存空间
回车开始进行写操作

写操作结束
```

(3) 本题(第 6 题)程序的 status 前后对比如下，可以看到 nonvoluntary_ctxt_switches 进程被动切换的次数增加了 148 次

前

```
(base) [root@ferry ~]# cat /proc/6398/status
Name:   6t
Umask:  0022
State:  S (sleeping)
Tgid:   6398
Ngid:   0
Pid:    6398
PPid:   1751
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:     4280 kB
VmSize:     4216 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       348 kB
VmRSS:       348 kB
RssAnon:             72 kB
RssFile:            276 kB
RssShmem:             0 kB
VmData:       56 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        28 kB
VmSwap:        0 kB
Threads:        1
SigQ:   1/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     1
```

后

```
(base) [root@ferry ~]# cat /proc/6398/status
Name:   6t
Umask:  0022
State:  S (sleeping)
Tgid:   6398
Ngid:   0
Pid:    6398
PPid:   1751
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:  2101376 kB
VmSize:     4216 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:    2097496 kB
VmRSS:       500 kB
RssAnon:             92 kB
RssFile:            408 kB
RssShmem:             0 kB
VmData:       56 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        28 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        4
nonvoluntary_ctxt_switches:     245
```

（4）对比第五题程序(没有进程竞争)和本题程序(有进程竞争)的两个 status 文件信息。从

图中可以得到结论：nonvoluntary_ctxt_switches: 进程被动切换次数多了 206 次，说明它们之间存在竞争关系，这跟它们占用的物理内存大小有关。

```
(base) [root@ferry ~]# cat /proc/2732/status
Name:   5t
Umask:  0022
State:  S (sleeping)
Tgid:   2732
Ngid:   0
Pid:    2732
PPid:   1239
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:   266364 kB
VmSize:   266364 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:    262576 kB
VmRSS:    262576 kB
RssAnon:         262168 kB
RssFile:            408 kB
RssShmem:             0 kB
VmData:   262204 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:       540 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        6
nonvoluntary_ctxt_switches:     39
```

5

```
(base) [root@ferry ~]# cat /proc/6398/status
Name:   6t
Umask:  0022
State:  S (sleeping)
Tgid:   6398
Ngid:   0
Pid:    6398
PPid:   1751
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmPeak:  2101376 kB
VmSize:     4216 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:    2097496 kB
VmRSS:     500 kB
RssAnon:           92 kB
RssFile:          408 kB
RssShmem:             0 kB
VmData:     56 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1936 kB
VmPTE:        28 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15076
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed_list:      0
voluntary_ctxt_switches:        4
nonvoluntary_ctxt_switches:     245
```

6

7.分配足够大的内存空间，其容量超过系统现有的空闲物理内存的大小，1）按 4KB 的间隔逐个单元进行写操作，重复访问数遍（使得程序运行时间可测量）；2）与前面访问总量和次数不变，但是将访问分成 16 个连续页为一组，逐组完成访问，记录运行时间。观察系统的状态，比较两者运行时间，给出判断和解释。

(此部分学习并借鉴了网上教程，得以完成)

（1）按 4KB 的间隔逐个单元进行写操作，重复访问数遍（使得程序运行时间可测量）；

```cpp
#include<unistd.h>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/time.h>
using namespace std;

int main(){
    timeval starttime,endtime;
    int *p;
    char s[8];
    int pid = getpid();
    sprintf(s,"%d",pid);
    int sum = 2.74 * 1024 * 1024 * 1024 / 4;
    p = new int[sum];
    //相隔 4KB 进行读写就是相隔 1024 个 int 进行读写
    int i,j;
    gettimeofday(&starttime,0);
    for(i=0;i<sum;i += 1024){
        p[i]++;
    }
    gettimeofday(&endtime,0);
    double timeuse = 1000000*(endtime.tv_sec - starttime.tv_sec) + endtime.tv_usec - starttime.tv_usec;
    timeuse /=2000;//ms
    cout<<timeuse<<"ms"<<" = "<<timeuse/1000<<"s"<<endl;
    delete[] p;
    return 0;
}
```

（2）将访问分成 16 个连续页为一组，逐组完成访问，记录运行时间

```cpp
#include<unistd.h>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
using namespace std;
```

```
9.
10. int sum = 2.74 * 1024 * 1024 * 1024 / 4;
11. int *p  = new int[sum];;
12. int a = 0;
13. void* visit(void* arg){
14.     int i = *(int *)arg;
15.     int end = i + sum/16;
16.     for(;i < end;i += 1024){
17.         ++p[i];
18.     }
19.     ++a;
20. }
21.
22.
23. int main(){
24.     timeval starttime,endtime;
25.     pthread_t id[16];
26.     int i,ret;
27.     int pc[16];
28.     int x = sum /16;
29.     for(i=-1;++i<16;){
30.         pc[i] = i * x;
31.     }
32.     char s[8];
33.     int pid = getpid();
34.     sprintf(s,"%d",pid);
35.     cout<<"进入计时状态"<<endl;
36.     gettimeofday(&starttime,0);
37.     for(i=-1;++i<16;){
38.         ret=pthread_create(&id[i],NULL,visit,&pc[i]);
39.     }
40.     while(a!=16){}
41.     gettimeofday(&endtime,0);
42.     cout<<"结束计时状态"<<endl;
43.     double timeuse = 1000000*(endtime.tv_sec - starttime.tv_sec) + endtime.tv_usec - s
    tarttime.tv_usec;
44.     timeuse /=1000;//ms
45.     cout<<timeuse<<"ms"<<" = "<<timeuse/1000<<"s"<<endl;
46.     delete[] p;
47.     return 0;
48. }
```

（3） 运行结果如下图所示，可以看到解法一比解法二慢。解法一是顺序访问数组，解法二是将数组切割成十六份，每份由一个线程负责访问，总共十六个线程。

```
(base) [root@ferry op]# g++ 71.c -o 71
(base) [root@ferry op]# ./71
775.268ms = 0.775267s
(base) [root@ferry op]# ./71
765.153ms = 0.765153s
(base) [root@ferry op]# ./71
663.528ms = 0.663528s
(base) [root@ferry op]#
```

解法一

```
(base) [root@ferry op]# g++ 72.c -o 72 -lpthread
(base) [root@ferry op]# ./72
进入计时状态
结束计时状态
593.253ms = 0.593253s
(base) [root@ferry op]# ./72
进入计时状态
结束计时状态
590.682ms = 0.590682s
(base) [root@ferry op]#
```

解法二

四、实验结论：（提供运行结果，对结果进行探讨、分析、评价，并提出结论性意见和改进想法）

1.学习使用 Linux 进程的内存分配、释放函数；
C 和 C++程序的内存分配函数及对应释放函数：realloc() / malloc() / calloc() / free()

2.学习 Linux proc 文件系统中关于内存影射的部分内容（了解/proc/pid/目录下的 maps、status、smap 等几个文件内部信息的解读）
/proc/pid/maps 最主要是通过 heap 判断内存地址

```
(base) [root@ferry 23263]# cat /proc/974/maps
5636f411a000-5636f4356000 r-xp 00000000 fd:01 292760                     /usr/bin/Xorg
5636f4555000-5636f4559000 r--p 0023b000 fd:01 292760                     /usr/bin/Xorg
5636f4559000-5636f4564000 rw-p 0023f000 fd:01 292760                     /usr/bin/Xorg
5636f4564000-5636f4584000 rw-p 00000000 00:00 0
5636f5e42000-5636fa03f000 rw-p 00000000 00:00 0                          [heap]
7f0ff8000000-7f0ff8021000 rw-p 00000000 00:00 0
7f0ff8021000-7f0ffc000000 ---p 00000000 00:00 0
7f0ffdae5000-7f0ffdb45000 rw-s 00000000 00:04 202997777                  /SYSV00000000 (deleted)
7f0ffdbc5000-7f0ffdc45000 rw-s 00000000 00:04 193560591                  /SYSV00000000 (deleted)
7f0ffdd85000-7f0ffdd91000 r-xp 00000000 fd:01 265462                     /usr/lib64/libnss_files-2.17.so
7f0ffdd91000-7f0ffdf90000 ---p 0000c000 fd:01 265462                     /usr/lib64/libnss_files-2.17.so
7f0ffdf90000-7f0ffdf91000 r--p 0000b000 fd:01 265462                     /usr/lib64/libnss_files-2.17.so
7f0ffdf91000-7f0ffdf92000 rw-p 0000c000 fd:01 265462                     /usr/lib64/libnss_files-2.17.so
7f0ffdf92000-7f0ffdf98000 rw-p 00000000 00:00 0
7f0ffdfd9000-7f0ffe039000 rw-s 00000000 00:04 1409034                    /SYSV00000000 (deleted)
```

/proc/pid/status 最主要是如下图所示：

VmPeak	进程所使用的虚拟内存的峰值
VmSize	进程当前使用的虚拟内存的大小
VmLck	已经锁住的物理内存的大小（锁住的物理内存不能交换到硬盘）
VmHWM	进程所使用的物理内存的峰值
VmRSS	进程当前使用的物理内存的大小
VmData	进程占用的数据段大小
VmStk	进程占用的栈大小
VmExe	进程占用的代码段大小（不包括库）
VmLib	进程所加载的动态库所占用的内存大小（可能与其它进程共享）
VmPTE	进程占用的页表大小（交换表项数量）
VmSwap	进程所使用的交换区的大小

3.编写程序，连续申请分配六个 128MB 空间（记为 1~6 号），然后释放第 2、3、5 号的 128MB 空间。然后再分配 1024MB，记录该进程的虚存空间变化（/proc/pid/maps）。
（1）虚拟内存变化情况如上面的分析,总结如下；

----------连续分配 6 个 128MB 空间（总 768M）--------
1-6 号：7fa50ec34000-7fa53ec3a000(768M)　　——> 存在
----------释放 2 号内存空间（总 640M）----------

3-6 号：7fa50ec34000-7fa52ec38000 (512M) ——> 存在
2 号：7fa52ec38000-7fa536c39000（128M）——> 本次释放
1 号：7fa536c39000-7fa53ec3a000 (128M) ——> 存在
----------释放 3 号内存空间（总 512M）----------
4-6 号：7fa50ec34000-7fa526c37000 (384M) ——> 存在
3 号：7fa526c37000-7fa52ec38000（128M）——> 本次释放
2 号：7fa52ec38000-7fa536c39000（128M）——> 已释放
1 号：7fa536c39000-7fa53ec3a000 (128M) ——> 存在
----------释放 5 号内存空间（总 384M）----------
6 号：7fa50ec34000-7fa516c35000（128M） ——> 存在
5 号：7fa516c35000-7fa51ec36000（128M）——> 本次释放
4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在
3 号：7fa526c37000-7fa52ec38000（128M）——> 已释放
2 号：7fa52ec38000-7fa536c39000（128M）——> 已释放
1 号：7fa536c39000-7fa53ec3a000(128M) ——> 存在
----------再分配 1024 内存空间(总 1536M)----------
6 号：7fa4cec33000-7fa516c35000(1152M=128M+1024M) ——> 存在（本次分配）
4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在
1 号：7fa536c39000-7fa53ec3a000 (128M) ——> 存在
----------再分配 64M 内存空间(总 1600M)----------
6 号：7fa4cec33000-7fa516c35000 (1152M) ——> 存在
4 号：7fa51ec36000-7fa526c37000 （128M）——> 存在
1 号：7fa532c38000-7fa53ec3a000 (192M=128M+64M) ——> 存在（本次分配）

（2）用户进程空间分配属于连续分配算法

可以从实验结果看出，程序中代码或数据的逻辑地址是相邻连续的，即为用户程序分配的是一个连续的内存空间

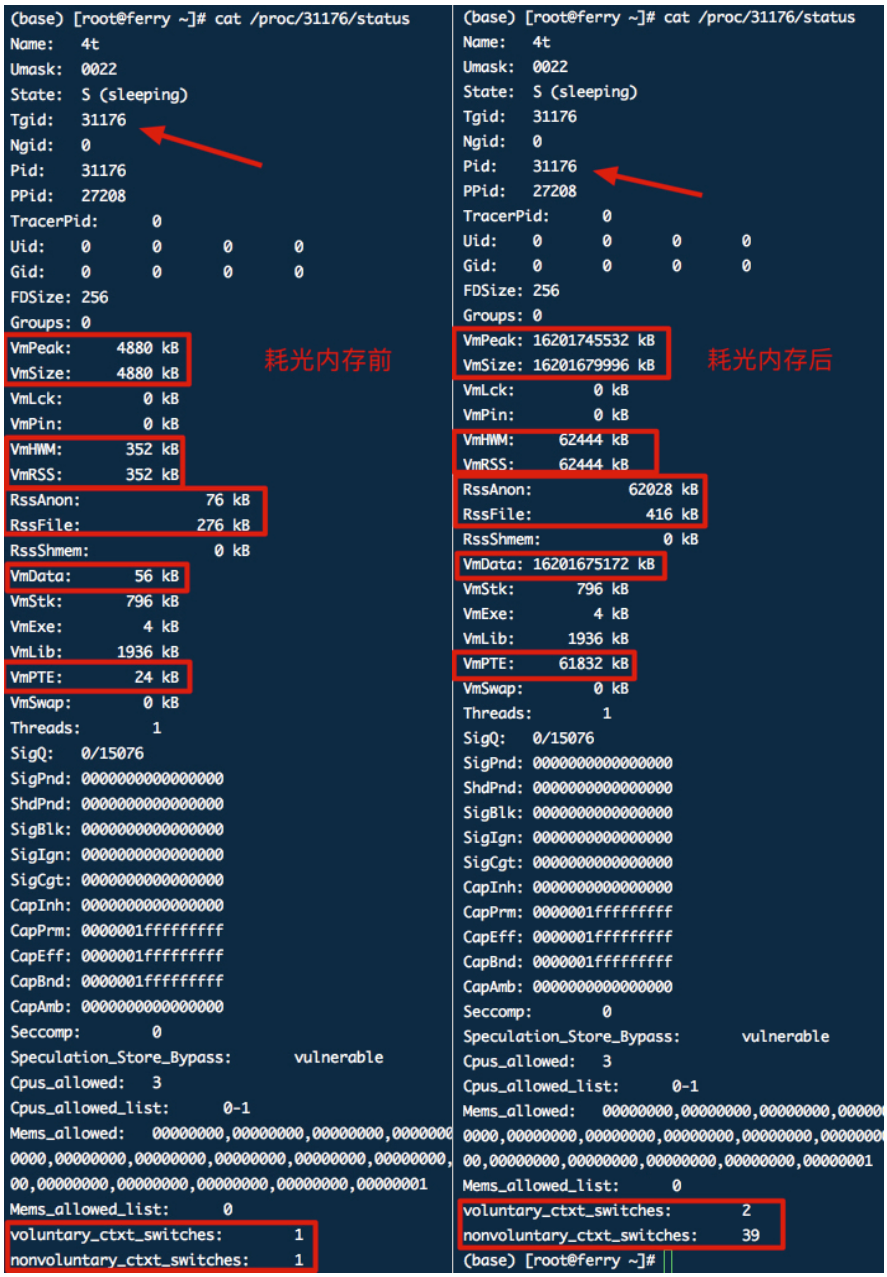
（3）首次适应算法

1024M 大于前面的两个 256M（被释放的 2 号和 3 号内存区）和 128M（被释放的 5 号内存区），实验结果得到的是该内存段最后被分配在 6 号内存区之后；64M 小于前面的两段 256M 和 128M，而且 256M 在 128M 前面，分配 64M 内存时选择了在 256M 内存碎片地址段而不是更小碎片的 128M，综上所述，选择的是首次适应算法。

（4）存在碎片问题

连续分配算法，在释放中间的内存段时会将一条连续内存段切割成两条的独的内存段，被释放的的内存段形成碎片。

4.设计一个程序测试出你的系统单个进程所能分配到的最大虚拟内存空间为多大。(20 分)
用 status 查看耗光内存前后的详细内存使用情况：可以看出此时 VmPeak 进程地址空间大小增加了（16201745532-4880）KB = 15GB，同样的 VmSize 进程虚拟地址空间的大小增加（16201679996-4880）=15G 左右，那么说明当前的进程能分配的最大的虚拟空间是 15G。

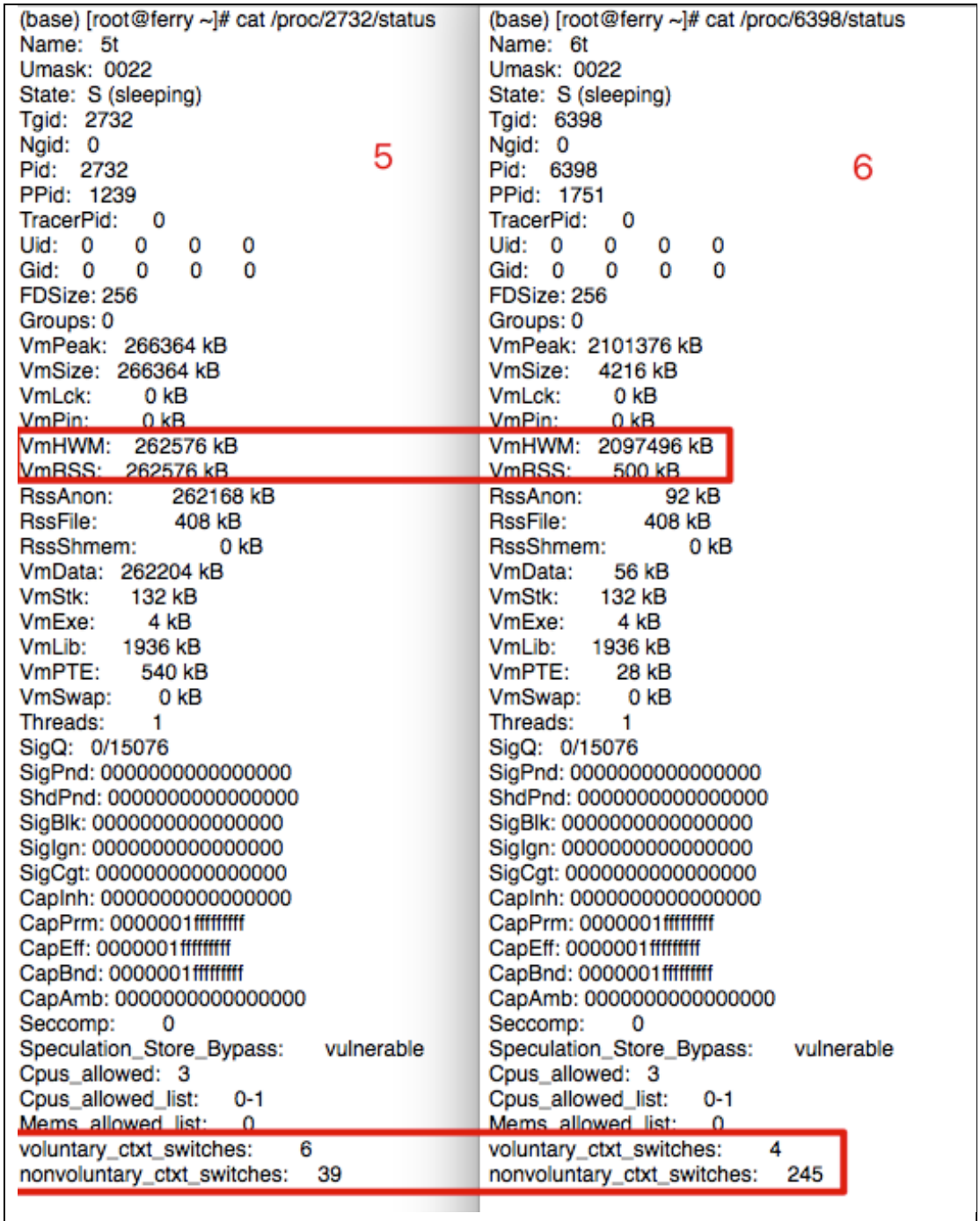


5.编写一个程序，分配 256MB 内存空间（或其他足够大的空间），检查分配前后/proc/pid/status 文件中关于虚拟内存和物理内存的使用情况，然后每隔 4KB 间隔将相应地址进行写操作，再次检查/proc/pid/status 文件中关于内存的情况，对比前后两次内存情况，说明所分配物理内存（物理内存块）的变化。然后重复上面操作，不过此时为读操作，再观察其变化。

分析结果说明了读操作不需要物理内存，写操作需要使用物理内存。

6.编写并运行（在第 5 步的程序未退出前）另一进程，分配等于或大于物理内存的空间，然后每隔 4KB 间隔将相应地址的字节数值增 1，此时再查看前一个程序的物理内存变化，观察两个进程竞争物理内存的现象。

对比第五题程序(没有进程竞争)和本题程序(有进程竞争)的两个 status 文件信息。从图中可以得到结论：nonvoluntary_ctxt_switches: 进程被动切换次数多了 206 次，说明它们之间存在竞争关系，这跟它们占用的物理内存大小有关。



7.分配足够大的内存空间，其容量超过系统现有的空闲物理内存的大小，1）按 4KB 的间隔逐个单元进行写操作，重复访问数遍（使得程序运行时间可测量）；2）与前面访问总量和次数不变，但是将访问分成 16 个连续页为一组，逐组完成访问，记录运行时间。观察系统的状态，比较两者运行时间，给出判断和解释。

可以看到解法一比解法二慢。解法一是顺序访问数组，解法二是将数组切割成十六份，每份由一个线程负责访问，总共十六个线程。

```
(base) [root@ferry op]# g++ 71.c -o 71
(base) [root@ferry op]# ./71
775.268ms = 0.775267s
(base) [root@ferry op]# ./71
765.153ms = 0.765153s
(base) [root@ferry op]# ./71
663.528ms = 0.663528s
(base) [root@ferry op]#
```

解法一

```
(base) [root@ferry op]# g++ 72.c -o 72 -lpthread
(base) [root@ferry op]# ./72
进入计时状态
结束计时状态
593.253ms = 0.593253s
(base) [root@ferry op]# ./72
进入计时状态
结束计时状态
590.682ms = 0.590682s
(base) [root@ferry op]#
```

解法二

五、实验体会：（根据自己情况填写）

通过这次实验，我懂得了内存管理分配和回收的知识。对于实验过程中遇到的问题，通过上网查找相关资料、研读课本上的理论知识以及请教助教和老师，都一一得到了解决。其中，可能由于系统的原因，在实验过程中遇到了一些和理论不太相符的情况，比如 heap 没有显示等等，但是最后都得到解决，多次实验下来最大的感受就是：实践出真知。只有把所学知识付诸实践才能真正地掌握知识。最后，特别感谢阮老师和陈助教！不然问题得不到解决，我还得原地绕圈呢。