

深圳大学实验报告

课程名称： 操作系统

实验项目名称： 实验3 内存管理：内存分配与回收

学 院： 计算机与软件学院

专 业： 计算机科学与技术

指导教师： 阮元

报告人： 刘睿辰 学号： 2018152051 班级： 数计班

实 验 时 间： 2021.5.25-2021.6.5

实验报告提交时间： 2021.6.5

一、实验目的与要求：

1. 加深对内存分配与使用操作的直观认识；
2. 掌握 Linux 操作系统的内存分配与使用的编程接口；
3. 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。

二、实验内容

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

三、实验环境

1. 硬件：个人电脑；
2. 操作系统：Linux 操作系统；
3. 实验平台：Ubuntu 18.04。

四、实验步骤及说明：

1. 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；
Linux 进程的内存分配一般调用的是 malloc() 函数，内存释放一般调用的是 free() 函数。
malloc() 函数：在堆区分配内存。堆(heap)指的是动态内存。根据计算机系统理论，用户空间由低地址到高地址依次是只读段、数据段、堆、文件映射区域和栈。
malloc() 函数的函数原型如图 1(1) 所示。

```
1  #include <malloc.h>
2  void *malloc(unsigned int num_bytes);
```

图 1(1). malloc() 函数的函数原型

该函数将分配长度为 num_byte 字节的内存块，如果分配成功则返回指向被分配内存的指针，否则返回空指针 NULL。

同样地，当内存不再使用的时候，我们调用 free() 函数来进行释放。

类似地，Linux 系统分配内存的函数还有 calloc() 函数。相比于 malloc() 函数，它指定了两个参数 n 和 size，表示分配 n 个长度为 size 的连续空间，函数返回一个指向分配起始地址的指针。同样地，该函数分配的内存不再使用的时候，也通过 free() 函数来进行释放。

此外，C 语言函数 realloc() 可以堆动态内存进行分配。其函数原型如图 1(2) 所示。

```
1  #include <stdlib.h>
2  #include <alloc.h>
3  void *realloc(void *mem_address, unsigned int newsize);
```

图 1(2). realloc() 函数的函数原型

该函数的作用是，更新 mem_address 所指内存区域的大小为 newsize 长度。同样地，该函数分配的内存不再使用的时候，也通过 free() 函数来进行释放。

C++ 中，new 函数作为一种更加方便的内存分配函数而被广泛使用。其用法为：

type *pstr = new type [size]

表示动态分配了size个type类型的内存给了pstr，来构成一个数组。也就是pstr作为首地址，连续分配size个type类型的内存。

new 函数分配的内存空间需要使用 delete 函数来进行释放。

2. 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、smmap 等几个文件内部信息的解读）；

/proc 文件系统是一个虚拟文件系统，通过它可以使用一种新的方法在 Linux 内核空间和用户空间之间进行通信。在 /proc 文件系统中，我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段，但是与普通文件不同的是，这些虚拟文件的内容都是动态创建的。

在这个目录下，所有进程都被一个文件夹表示，该文件夹名称就是该进程的进程号。首先我们编写一个死循环的代码，然后通过 top 命令得知进程号为 6343，此时我们运行 cd /proc 指令，如图 2 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/exp3$ cd /proc
ruichen@ruichen-virtual-machine:/proc$ ls
1      1667 181 210 330 5041 5209 6355 992      meminfo
10     1668 182 211 34 5046 5211 6343 997      misc
1067   1669 183 212 340 5049 5214 6356 acpi     modules
1094   167 184 213 35 5066 5216 6365 asound   mounts
11     1672 185 214 356 5072 5217 674  buddyinfo mpt
1115   1673 186 215 36 5077 5236 675  bus      mtrr
1128   1674 187 216 363 5088 5267 687  cgroups  net
1131   1675 188 217 365 5097 5268 695  cmdline  pagetypeinfo
1153   1679 189 218 368 5101 5272 7  consoles partitions
1155   168 19 219 370 5104 5297 700  cpuinfo  sched_debug
116    1680 190 22 372 5106 5305 772  crypto   schedstat
1168   1682 191 220 374 5116 5307 776  devices  scsi
1186   1685 192 221 376 5127 5334 777  diskstats self
1188   1687 193 222 378 5128 5351 778  dma      slabinfo
12     1688 1931 223 380 5135 5356 78  driver   softirqs
13     169 194 224 382 5139 5373 782  execdomains stat
1338   1691 195 225 384 5150 5380 784  fb        swaps
1349   1694 196 226 386 5152 5396 785  filesystems sys
14     1697 197 227 388 5160 5405 786  fs        sysrq-trigger
1442   17 198 228 391 5164 5407 79  interrupts sysvipc
15     170 199 229 393 5168 5441 8  iomem     thread-self
16     171 2 23 395 5169 5740 80  ioports  timer_list
1610   1713 20 230 4 517 5856 807  irq      tty
1615   172 200 24 452 5171 5867 81  kallsyms  uptime
1617   173 201 25 453 5172 5975 817  kcore     version
1620   174 202 254 4910 5173 5977 82  keys      version_signature
1622   1747 203 26 4915 5176 6 83  key-users vmallocinfo
1635   175 204 265 4916 5179 6121 879  kmsg      vmstat
1638   176 205 27 4929 5181 6133 89  kpagecgroup zoneinfo
1641   177 206 28 4933 519 6231 9  kpagecount
1645   178 207 289 4935 5192 6256 900  kpageflags
1652   179 208 29 4940 5202 6257 91  loadavg
1659   18 209 290 4944 5204 6261 917  locks
1660   180 21 30 5039 5206 6326 99  mdstat
```

图 2. /proc 的内容，我们可以看到进程 6343

我们看到很多进程号，分别代表不同的进程。我们在图 2 中发现了进程 6343，这是我们运行死循环程序的进程。我们进入到该进程，使用 ls 指令查看，可以看到该进程的内容，如图 3 所示。我们可以看到很多内容，比如 exe 文件（进程相关的可执行文件）、root（根目录）、maps（进程映像）、status（进程状态）、smmap（maps 扩展，显示虚存段使用物理页帧的情况）等。

接下来我们主要来看一下 maps, status 以及 smmap 的内容，观察它们分别的作用都是什么。

```

ruichen@ruichen-virtual-machine:/proc$ cd 6343
ruichen@ruichen-virtual-machine:/proc/6343$ ls
attr          exe           mounts        projid_map    status
autogroup     fd            mountstats    root          syscall
auxv          fdinfo        net           sched         task
cgroup        gid_map       ns            schedstat     timers
clear_refs    io            numa_maps     sessionid     timerslack_ns
cmdline        limits        oom_adj       setgroups     uid_map
comm          loginuid      oom_score     smaps         wchan
coredump_filter map_files     oom_score_adj smaps_rollup
cpuset         maps          pagemap       stack
cwd            mem           patch_state   stat
environ        mountinfo     personality    statm

```

图 3. 进程内部的内容，我们可以看到不同种类的文件

- 1) maps 输出的是进程的内存布局信息。我们输入 `cat /proc/6343/maps` 指令来查看 maps 信息，如图 4 所示。

```

ruichen@ruichen-virtual-machine:/proc/6343$ cat /proc/6343/maps
559859668000-559859669000 r-xp 00000000 08:01 1180702
/home/ruichen/桌面/oslab/exp3/HelloWorld
559859868000-559859869000 r--p 00000000 08:01 1180702
/home/ruichen/桌面/oslab/exp3/HelloWorld
559859869000-55985986a000 rw-p 00001000 08:01 1180702
/home/ruichen/桌面/oslab/exp3/HelloWorld
7f5f4d5ca000-7f5f4d7b1000 r-xp 00000000 08:01 394867
/lib/x86_64-linux-gnu/libc-2.27.so
7f5f4d7b1000-7f5f4d9b1000 ---p 001e7000 08:01 394867
/lib/x86_64-linux-gnu/libc-2.27.so
7f5f4d9b1000-7f5f4d9b5000 r--p 001e7000 08:01 394867
/lib/x86_64-linux-gnu/libc-2.27.so
7f5f4d9b5000-7f5f4d9b7000 rw-p 001eb000 08:01 394867
/lib/x86_64-linux-gnu/libc-2.27.so
7f5f4d9b7000-7f5f4d9bb000 rw-p 00000000 00:00 0
7f5f4d9bb000-7f5f4d9e2000 r-xp 00000000 08:01 394863
/lib/x86_64-linux-gnu/ld-2.27.so
7f5f4dbcc000-7f5f4dbce000 rw-p 00000000 00:00 0
7f5f4dbe2000-7f5f4dbe3000 r--p 00027000 08:01 394863
/lib/x86_64-linux-gnu/ld-2.27.so
7f5f4dbe3000-7f5f4dbe4000 rw-p 00028000 08:01 394863
/lib/x86_64-linux-gnu/ld-2.27.so
7f5f4dbe4000-7f5f4dbe5000 rw-p 00000000 00:00 0
7ffc11cc4000-7ffc11ce5000 rw-p 00000000 00:00 0
[stack]
7ffc11ded000-7ffc11df0000 r--p 00000000 00:00 0
[vvar]
7ffc11df0000-7ffc11df2000 r-xp 00000000 00:00 0
[vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0
[vsyscall]

```

图 4. 进程内部的 maps 的内容

每两行都代表一个具有特定属性的连续的内存区，开头是该区间的地址范围。第一行第二列的权限值分别代表：r 为可读、w 为可写、x 为可执行、s 为共享以及 p 为私有。第一行第三列的数字用于表示文件影射内存对应于文件中的起始位置（如果不是文件影射区则为 0）。

以图 5 中标记位置为例，该内存区占据了 559859668000-559859669000 的地址范围，属性 r-xp 表示只读、可执行和私有的一个内存区间。该内存区间的内容是从 /home/ruichen/桌面/oslab/exp3/HelloWorld 文件（索引节点号为 1180702）中

00000000 偏移位置拷贝 4K 字节（代码）而来的，这是该进程的代码区。
后面还有很多内存区间对应于 /lib/x86_64 - linux - gnu/ld - 2.27.so，是动态链接的 C 语言库的内容。

- 2) status 显示程序整体上的虚存空间统计数值变化。简单来说，该文件可以显示进程状态，占用虚拟内存情况，占用物理内存情况等。

使用 cat /proc/6343/status 指令来查看 status 信息，如图 5 所示。

```
ruichen@ruichen-virtual-machine:/proc/6343$ cat /proc/6343/status
Name: HelloWorld
Umask: 0022
State: R (running)
Tgid: 6343
Ngid: 0
Pid: 6343
PPid: 6335
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 256
Groups: 4 24 27 30 46 116 126 1000
NSTgid: 6343
NSpid: 6343
NSpgid: 6343
NSSid: 6335
VmPeak: 4456 kB
VmSize: 4376 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 748 kB
VmRSS: 748 kB
RssAnon: 64 kB
RssFile: 684 kB
RssShmem: 0 kB
VmData: 44 kB
VmStk: 132 kB
VmExe: 4 kB
VmLib: 2112 kB
VmPTE: 52 kB
VmSwap: 0 kB
HugetlbPages: 0 kB
CoreDumping: 0
Threads: 1
SigQ: 0/7700
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
NoNewPrivs: 0
Seccomp: 0
Speculation_Store_Bypass: thread vulnerable
Cpus_allowed: ffffffff,ffffffff,ffffffff,ffffffff
Cpus_allowed_list: 0-127
Mems_allowed: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
00,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,000000
00,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
Mems_allowed_list: 0
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 128337
```

图 5. 进程内部的 status 的内容

这部分的内容显示了进程名、进程状态、线程组 ID、进程号、父进程的进程号一系列详细的信息。后面的一些信息，例如 VmPeak 代表当前进程运行过程中占用内存的峰值，当前为 4456KB；VmSize 代表进程现在正在占用的内存，当前为 4376KB；VmHWM 是程序得到分配到物理内存的峰值，当前为 748KB；VmRSS

是程序现在使用的物理内存，当前为 748KB。VmData 表示进程数据段的大小；VmStk 表示进程堆栈段的大小；VmExe 表示进程代码的大小；VmLib 表示进程所使用 LIB 库的大小等等。

在该部分的最后，voluntary_ctxt_switches 表示进程主动切换的次数；nonvoluntary_ctxt_switches 进程被动切换的次数，体现了进程优先级的差别。

3) smaps

这部分显示了进程各个虚存段使用物理页帧的情况。我们使用 cat /proc/6343/smaps 指令来观察结果。由于之前在图 4 中一共出现了 17 个内存区，所以这次的输出结果就是 17 组数据。我们依然选取第一组，也就是占据了 559859668000-559859669000 的地址范围的内存区作为例子，如图 6 所示。

```
559859668000-559859669000 r-xp 00000000 08:01 1180702
/home/ruichen/桌面/oslab/exp3/HelloWorld
Size: 4 kB
KernelPageSize: 4 kB
MMUPageSize: 4 kB
Rss: 4 kB
Pss: 4 kB
Shared_Clean: 0 kB
Shared_Dirty: 0 kB
Private_Clean: 4 kB
Private_Dirty: 0 kB
Referenced: 4 kB
Anonymous: 0 kB
LazyFree: 0 kB
AnonHugePages: 0 kB
ShmemPmdMapped: 0 kB
Shared_Hugetlb: 0 kB
Private_Hugetlb: 0 kB
Swap: 0 kB
SwapPss: 0 kB
Locked: 0 kB
VmFlags: rd ex mr mw me dw sd
```

图 6. 进程内部的smaps 的内容（第一个内存区）

其中 Size 表示该虚存区间的大小，这里为 4KB。Rss (Resident Set Size)表示该段虚存空间当前正在使用了多少物理内存（包含共享库占用的物理内存），其计算方式为：

$$Rss = Shared_Clean + Shared_Dirty + Private_Clean + Private_Dirty$$

Pss (Private RSS)表示实际使用的物理内存（按比例分摊计算共享库占用的内存）。Shared_Clean、Shared_Dirty、Private_Clean 和 Private_Dirty 这四个统计项的 shared/private 表示该虚存区间使用的物理页帧是共享还是私有，而 clean/dirty 分表表示没有修改过和被修改过的物理页帧。

Referenced 统计的是被访问过的物理页帧。Anonymous 统计的是“匿名”页帧（即不与文件相关联的）。Swap 统计那些对应于匿名内存空间但已经换出到交换设备（或交换文件）的页帧。KernelPageSize 是内核物理页帧的大小，MMUPageSize 是内存控制器的页帧大小。

从图 6 中我们可以看到虚存区间 559859668000-559859669000 共 4KB

(Size=4K), 该虚存区间已经映射到了一个物理页帧 (Rss = 4K), 该区间映射的页帧中私有的且未改写过 (Private_Clean = 4K) 的页帧一个, 该区间映射的页帧里被访问过的 (Referenced = 4K) 有一个。

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?

根据题目要求, 我们编写的代码如下所示。我们用指针数组来存储所有待分配内存的指针。然后按照题目要求, 先分配六个 128MB 空间, 如第 13 行-第 19 行所示。然后释放 2, 3, 5 号内存空间, 如 24 行-34 行所示。接下来分配 1024M, 如 39 行-40 行所示。然后再分配 64M 内存, 如第 42 行-第 43 行所示。每一个 getchar 指令都是提示用户来点击回车来执行下一段代码, 方便我们实时查看虚存信息。

```
1. #include <stdio.h>
2. #include <malloc.h>
3. #include <unistd.h>
4.
5. int main()
6. {
7.     printf("This is process %d\n", getpid());
8.
9.     printf("Before malloc()s.\n");  getchar();
10.
11.     int *bufs[8];
12.
13.     bufs[0] = (int *)malloc(1024*1024*32*sizeof(int)); //1
14.     //(2^10)*(2^10)*(2^5)*(2^2)=2^27=(2^7)*(2^20)=128M
15.     bufs[1] = (int *)malloc(1024*1024*32*sizeof(int)); //2
16.     bufs[2] = (int *)malloc(1024*1024*32*sizeof(int)); //3
17.     bufs[3] = (int *)malloc(1024*1024*32*sizeof(int)); //4
18.     bufs[4] = (int *)malloc(1024*1024*32*sizeof(int)); //5
19.     bufs[5] = (int *)malloc(1024*1024*32*sizeof(int)); //6
20.
21.     printf("After 6 malloc()s.\n");
22.     getchar();
23.
24.     free(bufs[1]); //free 2
25.     printf("2 is freed\n");
26.     getchar();
27.
28.     free(bufs[2]); //free 3
29.     printf("3 is freed\n");
```

```

30.  getchar();
31.
32.  free(bufs[4]); //free 5
33.  printf("5 is freed\n");
34.  getchar();
35.
36.  printf("After 3 free()s.\n");
37.  getchar();
38.
39.  bufs[6] = (int *)malloc(1024*1024*256*sizeof(int));
40.  printf("1024M allocated\n"); getchar();
41.
42.  bufs[7] = (int *)malloc(1024*1024*16*sizeof(int));
43.  printf("64M allocated\n");
44.  getchar();
45.
46.  free(bufs[0]); //free 1
47.  free(bufs[3]); //free 4
48.  free(bufs[5]); //free 6
49.  free(bufs[6]); //free 7
50.  free(bufs[7]); //free 8
51.  return 0;
52. }

```

运行该程序，在分配内存前后，我们来看一下 maps 的内容，如图 7 所示。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab/exp3$ cat /proc/2975/maps
55daf4304000-55daf4305000 r-xp 00000000 08:01 1180704 /home/ruichen/桌面/oslab/exp3/ans3
55daf4504000-55daf4505000 r--p 00000000 08:01 1180704 /home/ruichen/桌面/oslab/exp3/ans3
55daf4505000-55daf4506000 rw-p 00001000 08:01 1180704 /home/ruichen/桌面/oslab/exp3/ans3
55daf50be000-55daf50df000 rw-p 00000000 00:00 0 [heap]
7f89848b7000-7f8984a9e000 r-xp 00000000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984a9e000-7f8984c9e000 ---p 001e7000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984c9e000-7f8984ca2000 r--p 001e7000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984ca2000-7f8984ca4000 rw-p 001eb000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984ca4000-7f8984ca8000 rw-p 00000000 00:00 0
7f8984ca8000-7f8984ccf000 r-xp 00000000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7f8984eb9000-7f8984ebb000 rw-p 00000000 00:00 0
7f8984ecf000-7f8984ed0000 r--p 00027000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7f8984ed0000-7f8984ed1000 rw-p 00028000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7f8984ed1000-7f8984ed2000 rw-p 00000000 00:00 0
7ffd28069000-7ffd2808a000 rw-p 00000000 00:00 0 [stack]
7ffd28134000-7ffd28137000 r--p 00000000 00:00 0 [vvar]
7ffd28137000-7ffd28139000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

图 7(a). 分配内存之前 proc/2975/maps 的内容

```

ruichen@ruichen-virtual-machine:~/桌面/oslab/exp3$ cat /proc/2975/maps
55daf4304000-55daf4305000 r-xp 00000000 08:01 1180704 /home/ruichen/桌面/oslab/exp3/ans3
55daf4504000-55daf4505000 r--p 00000000 08:01 1180704 /home/ruichen/桌面/oslab/exp3/ans3
55daf4505000-55daf4506000 rw-p 00001000 08:01 1180704 /home/ruichen/桌面/oslab/exp3/ans3
55daf50be000-55daf50df000 rw-p 00000000 00:00 0 [heap]
7f89548b1000-7f89848b7000 rw-p 00000000 00:00 0
7f89848b7000-7f8984a9e000 r-xp 00000000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984a9e000-7f8984c9e000 ---p 001e7000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984c9e000-7f8984ca2000 r--p 001e7000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984ca2000-7f8984ca4000 rw-p 001eb000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7f8984ca4000-7f8984ca8000 rw-p 00000000 00:00 0
7f8984ca8000-7f8984ccf000 r-xp 00000000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7f8984eb9000-7f8984ebb000 rw-p 00000000 00:00 0
7f8984ecf000-7f8984ed0000 r--p 00027000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7f8984ed0000-7f8984ed1000 rw-p 00028000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7f8984ed1000-7f8984ed2000 rw-p 00000000 00:00 0
7ffd28069000-7ffd2808a000 rw-p 00000000 00:00 0 [stack]
7ffd28134000-7ffd28137000 r--p 00000000 00:00 0 [vvar]
7ffd28137000-7ffd28139000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

图 7(b). 分配内存之后 proc/2975/maps 的内容

1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hc	1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hc
2.	55daf4504000-55daf4505000	r-p	00000000	08:01	1180704	/hc	2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704	/hc
3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc	3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc
4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0	[he	4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0	[he
5.							5.	7f89548b1000-7f89848b7000	rw-p	00000000	00:00	0	
6.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867	/li	6.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867	/li
7.	7f8984a9e000-7f8984c9e000	--p	001e7000	08:01	394867	/li	7.	7f8984a9e000-7f8984c9e000	--p	001e7000	08:01	394867	/li
8.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867	/li	8.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867	/li
9.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867	/li	9.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867	/li
10.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0		10.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0	
11.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863	/li	11.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863	/li
12.	7f8984ccf000-7f8984ceb000	rw-p	00000000	00:00	0		12.	7f8984ccf000-7f8984ceb000	rw-p	00000000	00:00	0	
13.	7f8984ceb000-7f8984ed0000	r--p	00027000	08:01	394863	/li	13.	7f8984ceb000-7f8984ed0000	r--p	00027000	08:01	394863	/li
14.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863	/li	14.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863	/li
15.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0		15.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0	
16.	7ff4d28069000-7ff4d2808a000	rw-p	00000000	00:00	0	[st	16.	7ff4d28069000-7ff4d2808a000	rw-p	00000000	00:00	0	[st
17.	7ff4d28134000-7ff4d28137000	r--p	00000000	00:00	0	[v	17.	7ff4d28134000-7ff4d28137000	r--p	00000000	00:00	0	[v
18.	7ff4d28137000-7ff4d28139000	r-xp	00000000	00:00	0	[vc	18.	7ff4d28137000-7ff4d28139000	r-xp	00000000	00:00	0	[vc
19.	ffffffff600000-ffffffff601000	r-xp	00000000	00:00	0	[vs	19.	ffffffff600000-ffffffff601000	r-xp	00000000	00:00	0	[vs
20.							20.						

图 8. 分配内存前后 proc/2975/maps 的变化

我们根据文本对比之后发现内存分配之后出现了一个新的内存区，地址范围为 7f89548b1000-7f89848b7000，如图 8 所示。按照理论情况，这个内存区域应该包含了 6 个 128M 的大小。我们来进行计算，有

$$(7f89848b7000 - 7f89548b1000)_{16} = (3000\ 6000)_{16} = (805, 330, 944)_{10}$$

十进制 805330944，经过计算，我们有

$$805330944 \div 1024 \div 1024 \div 128 \approx 6.00018$$

忽略页表大小，也就是说我们分配了 6 个 128M 大小的空间。这个值比理论值要大一些。接下来我们来比对分配前后 status 的区别，如图 9 所示。

17.	VmPeak:	4508 kB	17.	VmPeak:	790964 kB
18.	VmSize:	4508 kB	18.	VmSize:	790964 kB
19.	VmLck:	0 kB	19.	VmLck:	0 kB
20.	VmPin:	0 kB	20.	VmPin:	0 kB
21.	VmHWM:	756 kB	21.	VmHWM:	756 kB
22.	VmRSS:	756 kB	22.	VmRSS:	756 kB
23.	RssAnon:	64 kB	23.	RssAnon:	64 kB
24.	RssFile:	692 kB	24.	RssFile:	692 kB
25.	RssShmem:	0 kB	25.	RssShmem:	0 kB
26.	VmData:	176 kB	26.	VmData:	786632 kB
27.	VmStk:	132 kB	27.	VmStk:	132 kB
28.	VmExe:	4 kB	28.	VmExe:	4 kB
29.	VmLib:	2112 kB	29.	VmLib:	2112 kB
30.	VmPTE:	52 kB	30.	VmPTE:	80 kB

图 9. 分配内存前后 proc/2975/status 的变化

由于 VmSize 代表进程现在正在占用的内存，当前分配造成的差异为 786456KB，此外由于 $786456 \div 1024 \div 128 \approx 6.00018$ ，和我们之前的计算结果是吻合的。此外，VmData 和 VmPTE 也相应地发生了变化。

接下来释放 2，3，5 号内存区。首先释放 2 号内存区，比对释放前后的变化。

1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hc	1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hc
2.	55daf4504000-55daf4505000	r-p	00000000	08:01	1180704	/hc	2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704	/hc
3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc	3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc
4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0	[he	4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0	[he
5.	7f89548b1000-7f89848b7000	rw-p	00000000	00:00	0		5.	7f89548b1000-7f89848b7000	rw-p	00000000	00:00	0	
6.							6.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0	
7.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867	/li	7.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867	/li
8.	7f8984a9e000-7f8984c9e000	--p	001e7000	08:01	394867	/li	8.	7f8984a9e000-7f8984c9e000	--p	001e7000	08:01	394867	/li
9.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867	/li	9.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867	/li
10.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867	/li	10.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867	/li
11.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0		11.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0	
12.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863	/li	12.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863	/li
13.	7f8984ccf000-7f8984ceb000	rw-p	00000000	00:00	0		13.	7f8984ccf000-7f8984ceb000	rw-p	00000000	00:00	0	
14.	7f8984ceb000-7f8984ed0000	r--p	00027000	08:01	394863	/li	14.	7f8984ceb000-7f8984ed0000	r--p	00027000	08:01	394863	/li
15.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863	/li	15.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863	/li
16.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0		16.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0	
17.	7ff4d28069000-7ff4d2808a000	rw-p	00000000	00:00	0	[st	17.	7ff4d28069000-7ff4d2808a000	rw-p	00000000	00:00	0	[st
18.	7ff4d28134000-7ff4d28137000	r--p	00000000	00:00	0	[v	18.	7ff4d28134000-7ff4d28137000	r--p	00000000	00:00	0	[v
19.	7ff4d28137000-7ff4d28139000	r-xp	00000000	00:00	0	[vc	19.	7ff4d28137000-7ff4d28139000	r-xp	00000000	00:00	0	[vc
20.	ffffffff600000-ffffffff601000	r-xp	00000000	00:00	0	[vs	20.	ffffffff600000-ffffffff601000	r-xp	00000000	00:00	0	[vs

图 10. 释放内存区 2 前后 proc/2975/maps 的变化

我们从图 10 中可以看到，原来的内存区 548b1000-848b7000 被切割成两部分，一部分是 7c8b6000-848b7000(buf1)，另一区域是 548b1000-748b5000(buf3-buf6)，经过计算这两个区域分别对应 1 个 128M 和 4 个 128M，如图 11 所示。（计算时方便起见，忽略地址前 4 位，因为都是 7f89 不影响计算，下同）

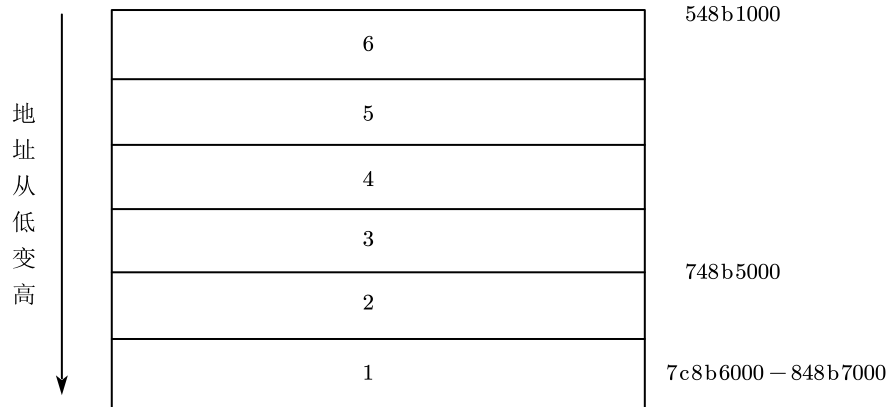


图 11. 释放内存区 2 时虚存的地址

所以释放掉第二个内存区的时候会被切割开来。那么接下来释放内存区 3，maps 输出结果发生了如图 12 所示的变化。

1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hc	1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704
2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704	/hc	2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704
3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc	3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704
4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0	/hc	4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0
5.	7f89548b1000-7f89748b5000	rw-p	00000000	00:00	0		5.	7f89548b1000-7f896c8b4000	rw-p	00000000	00:00	0
6.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0		6.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0
7.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867	/lj	7.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867
8.	7f8984a9e000-7f8984c9e000	---p	001e7000	08:01	394867	/lj	8.	7f8984a9e000-7f8984c9e000	---p	001e7000	08:01	394867
9.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867	/lj	9.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867
10.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867	/lj	10.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867
11.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0		11.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0
12.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863	/lj	12.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863
13.	7f8984eb9000-7f8984ebb000	rw-p	00000000	00:00	0		13.	7f8984eb9000-7f8984ebb000	rw-p	00000000	00:00	0
14.	7f8984ecf000-7f8984ed0000	r--p	00027000	08:01	394863	/lj	14.	7f8984ecf000-7f8984ed0000	r--p	00027000	08:01	394863
15.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863	/lj	15.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863
16.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0		16.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0
17.	7ffd28069000-7ffd2808a000	rw-p	00000000	00:00	0	[st	17.	7ffd28069000-7ffd2808a000	rw-p	00000000	00:00	0
18.	7ffd28134000-7ffd28137000	r--p	00000000	00:00	0	[va	18.	7ffd28134000-7ffd28137000	r--p	00000000	00:00	0
19.	7ffd28137000-7ffd28139000	r-xp	00000000	00:00	0	[vc	19.	7ffd28137000-7ffd28139000	r-xp	00000000	00:00	0
20.	ffffffff600000-ffffffff601000	r-xp	00000000	00:00	0	[vs	20.	ffffffff600000-ffffffff601000	r-xp	00000000	00:00	0
21.							21.					

图 12. 释放内存区 3 前后 proc/2975/maps 的变化

根据图 12 的显示，虚存地址发生了如图 13 所示的变化。

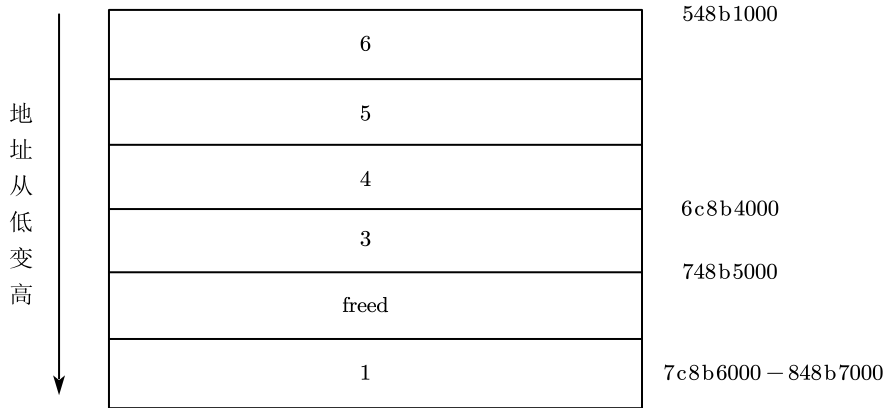


图 13. 释放内存区 3 时虚存的地址

所以现在内存区依然是 2 个区域，分别是 7c8b6000-848b7000 和 548b1000-6c8b4000，

分别代表 buf1 和 buf4-buf6。

接下来我们释放 buf5，maps 变化如图 14 所示。

1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hc	1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704
2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704	/hc	2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704
3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc	3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704
4.	55daf50b0000-55daf50df000	rw-p	00000000	00:00	0	/he	4.	55daf50b0000-55daf50df000	rw-p	00000000	00:00	0
5.	7f89548b1000-7f896c8b4000	rw-p	00000000	00:00	0		5.	7f89548b1000-7f896c8b2000	rw-p	00000000	00:00	0
6.							6.	7f89648b3000-7f896c8b4000	rw-p	00000000	00:00	0
7.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0		7.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0
8.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867	/lj	8.	7f89848b7000-7f8984a9e000	r-xp	00000000	08:01	394867
9.	7f8984a9e000-7f8984c9e000	---p	001e7000	08:01	394867	/lj	9.	7f8984a9e000-7f8984c9e000	---p	001e7000	08:01	394867
10.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867	/lj	10.	7f8984c9e000-7f8984ca2000	r--p	001e7000	08:01	394867
11.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867	/lj	11.	7f8984ca2000-7f8984ca4000	rw-p	001eb000	08:01	394867
12.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0		12.	7f8984ca4000-7f8984ca8000	rw-p	00000000	00:00	0
13.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863	/lj	13.	7f8984ca8000-7f8984ccf000	r-xp	00000000	08:01	394863
14.	7f8984ccf000-7f8984ebb000	rw-p	00000000	00:00	0		14.	7f8984ccf000-7f8984ebb000	rw-p	00000000	00:00	0
15.	7f8984ecf000-7f8984ed0000	r--p	00027000	08:01	394863	/lj	15.	7f8984ecf000-7f8984ed0000	r--p	00027000	08:01	394863
16.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863	/lj	16.	7f8984ed0000-7f8984ed1000	rw-p	00028000	08:01	394863
17.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0		17.	7f8984ed1000-7f8984ed2000	rw-p	00000000	00:00	0
18.	7ffcd28069000-7ffcd2808a000	rw-p	00000000	00:00	0	[st	18.	7ffcd28069000-7ffcd2808a000	rw-p	00000000	00:00	0
19.	7ffcd28134000-7ffcd28137000	r--p	00000000	00:00	0	[v	19.	7ffcd28134000-7ffcd28137000	r--p	00000000	00:00	0
20.	7ffcd28137000-7ffcd28139000	r-xp	00000000	00:00	0	[vc	20.	7ffcd28137000-7ffcd28139000	r-xp	00000000	00:00	0
21.	fffffffff600000-fffffffff601000	r-xp	00000000	00:00	0	[vs	21.	fffffffff600000-fffffffff601000	r-xp	00000000	00:00	0

图 14. 释放内存区 5 前后 proc/2975/maps 的变化

根据图 14 的显示，虚存地址发生了如图 15 所示的变化。

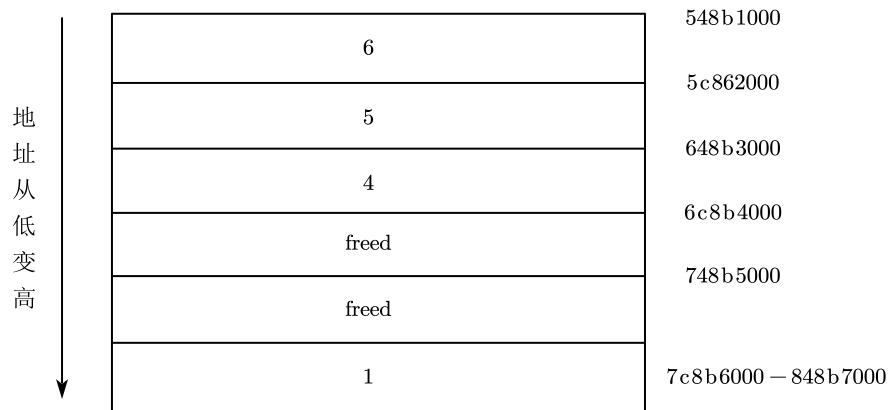


图 15. 释放内存区 5 时虚存的地址

由此我们也可以得到结论，6 个内存区的地址范围为：

buf1: 7f897c8b6000 — 7f89848b7000

buf2: 7f89748b5000 — 7f897c8b6000

buf3: 7f896c8b4000 — 7f89748b5000

buf4: 7f89648b3000 — 7f896c8b4000

buf5: 7f895c862000 — 7f89648b3000

buf6: 7f89548b1000 — 7f895c862000

再来看一下 status 的信息。在没有进行内存分配的时候查看一部分 status 的信息，如图 16 所示。VmPeak 代表当前进程运行过程中占用内存的峰值，当前为 4508KB；VmSize 代表进程现在正在占用的内存，当前为 4508KB；VmHWM 是程序得到分配到物理内存的峰值，当前为 708KB；VmRSS 是程序现在使用的物理内存，当前为 708KB。

接下来我们分别查看分配空间以及释放了三次空间之后 status 的信息，如图 17 所示。我们可以看到，如图 17(a)所示，当一次性分配 6 个 128M 大小的空间之后，VmPeak 有很大的提升，提升到了 790964KB，同时 VmSize 也有了很大的提升，代表此时进程占用内存量变大，当前达到了最大值，790964KB。

```

VmPeak:    4508 kB
VmSize:    4508 kB
VmLck:      0 kB
VmPin:      0 kB
VmHWM:     708 kB
VmRSS:     708 kB
RssAnon:           64 kB
RssFile:          644 kB
RssShmem:           0 kB
VmData:     176 kB
VmStk:      132 kB
VmExe:        4 kB
VmLib:     2112 kB
VmPTE:       52 kB
VmSwap:      0 kB

```

图 16. 开始时期 status 信息

<pre> VmPeak: 790964 kB VmSize: 790964 kB VmLck: 0 kB VmPin: 0 kB VmHWM: 708 kB VmRSS: 708 kB RssAnon: 64 kB RssFile: 644 kB RssShmem: 0 kB VmData: 786632 kB VmStk: 132 kB VmExe: 4 kB VmLib: 2112 kB VmPTE: 80 kB VmSwap: 0 kB </pre>	<pre> VmPeak: 790964 kB VmSize: 659888 kB VmLck: 0 kB VmPin: 0 kB VmHWM: 1436 kB VmRSS: 1436 kB RssAnon: 100 kB RssFile: 1336 kB RssShmem: 0 kB VmData: 655556 kB VmStk: 132 kB VmExe: 4 kB VmLib: 2112 kB VmPTE: 80 kB VmSwap: 0 kB </pre>
(a) Before free the three buffers	(b) After free buf2
<pre> VmPeak: 790964 kB VmSize: 528812 kB VmLck: 0 kB VmPin: 0 kB VmHWM: 1436 kB VmRSS: 1432 kB RssAnon: 96 kB RssFile: 1336 kB RssShmem: 0 kB VmData: 524480 kB VmStk: 132 kB VmExe: 4 kB VmLib: 2112 kB VmPTE: 76 kB VmSwap: 0 kB </pre>	<pre> VmPeak: 790964 kB VmSize: 397736 kB VmLck: 0 kB VmPin: 0 kB VmHWM: 1436 kB VmRSS: 1428 kB RssAnon: 92 kB RssFile: 1336 kB RssShmem: 0 kB VmData: 393404 kB VmStk: 132 kB VmExe: 4 kB VmLib: 2112 kB VmPTE: 76 kB VmSwap: 0 kB </pre>
(c) After free buf3	(d) After free buf5

图 17. 分配以及释放时 status 的变化

经过内存区 2 释放之后, VmPeak 保持不变, 同时 VmSize 降低到 65988KB, 如图 17(b) 所示。经过内存区 3 释放, VmSize 降低到 528812KB, 如图 17(c)所示; 而内存 5 被释放之后, VmSize 降低到了 397736KB, 如图 17(d)所示。而结合物理内存使用情况, 物理内存分配始终要低一些。VmData 表示进程数据段, 这个值和 VmSize 保持接近, 但是要略小于进程正在占用的内存。

再分配 1024MB 的时候, 我们对比 maps 的结果如图 18 所示。

根据图 18 反应的情况, 虚存再次向低地址前近。前进大小为

$$548b\ 1000 - 148b\ 0000 = 4000\ 1000$$

该值转换为十进制并除以 2^{20} 之后等于 1024.00390625, 也就是这部分的多出来的大小

为约为 1024MB。这个过程并不难理解，如图 19 所示。

1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hon	1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704
2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704	/hon	2.	55daf4504000-55daf4505000	r--p	00000000	08:01	1180704
3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc	3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704
4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0	[hes	4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0
5.	7f89548b1000-7f895c8b2000	rw-p	00000000	00:00	0		5.	7f89148b0000-7f895c8b2000	rw-p	00000000	00:00	0
6.	7f89648b3000-7f896c8b4000	rw-p	00000000	00:00	0		6.	7f89648b3000-7f896c8b4000	rw-p	00000000	00:00	0
7.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0		7.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0

图 18. 新分配 1024M 前后 maps 的变化

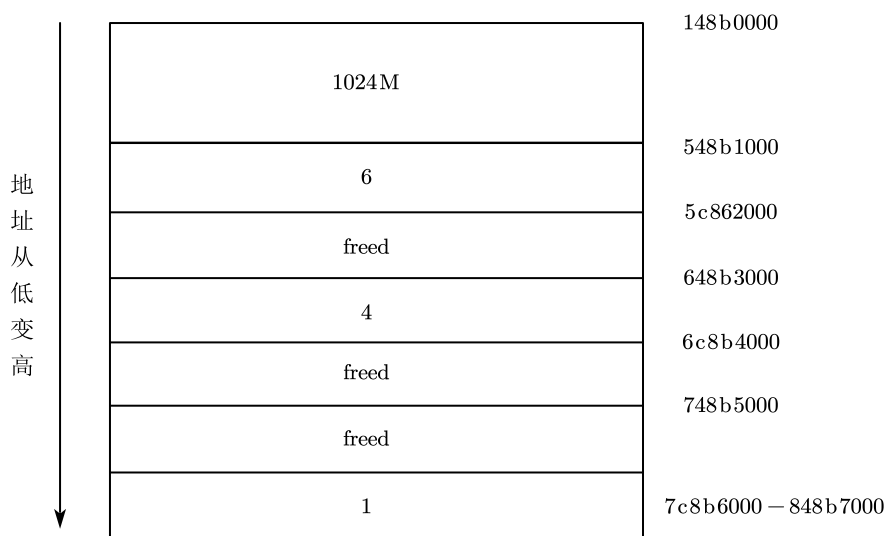


图 19. 新分配 1024M 之后虚存的变化

再来看 status 的变化。如图 20 所示。VmPeak 发生了明显的提升，达到了 1446316KB，同时 VmSize 也提升到 1446316KB。VmData 表示进程数据段，这个值依然和 VmSize 保持接近同时略小于进程正在占用的内存。物理内存的使用前后区别不大，因为我们并没有提高 CPU 利用率，只是分配了空间而已。

1.	VmPeak:	790964 kB	1.	VmPeak:	1446316 kB
2.	VmSize:	397736 kB	2.	VmSize:	1446316 kB
3.	VmLck:	0 kB	3.	VmLck:	0 kB
4.	VmPin:	0 kB	4.	VmPin:	0 kB
5.	VmHWM:	1436 kB	5.	VmHWM:	1436 kB
6.	VmRSS:	1428 kB	6.	VmRSS:	1428 kB
7.	RssAnon:	92 kB	7.	RssAnon:	92 kB
8.	RssFile:	1336 kB	8.	RssFile:	1336 kB
9.	RssShmem:	0 kB	9.	RssShmem:	0 kB
10.	VmData:	393404 kB	10.	VmData:	1441984 kB
11.	VmStk:	132 kB	11.	VmStk:	132 kB
12.	VmExe:	4 kB	12.	VmExe:	4 kB
13.	VmLib:	2112 kB	13.	VmLib:	2112 kB
14.	VmPTE:	76 kB	14.	VmPTE:	84 kB
15.	VmSwap:	0 kB	15.	VmSwap:	0 kB
16.			16.		

图 20. 新分配 1024M 之后 status 的变化

如果再分配 64M，根据我们之前的分析，有

$$64 \times 1024 \times 1024 = 67108864 = (4000000)_{16}$$

查资料分析，推测系统使用的是首次适应算法，即将从空闲分区表的第一个表目起查找该表，把最先能够满足要求的空闲区分配给作业，这种方法目的在于减少查找时间。

如果是这样的话，系统会从第一个块开始检索。由于我们这次只需要 64M 大小的内存，从图 19 中显示来看，748b5000 - 7c8b6000 这 128M 完全可以容纳这 64M，所以我们将如图 21 所示的位置分配给这 64M，随即第一个内存区的大小也发生了变化，

从 128M 扩充到 192M.

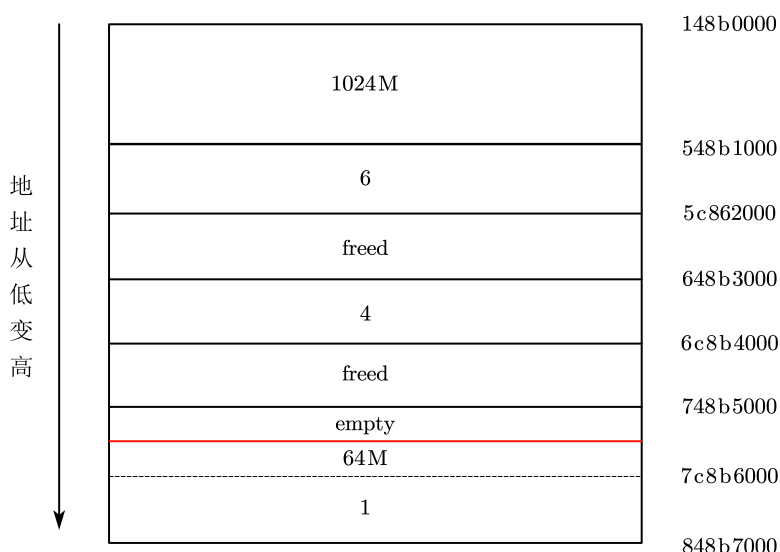


图 21. 在第一个内存区扩充 64M

所以我们要算的就是红线位置的内存地址是什么。之前计算的 64M 对应了 $(4000000)_{16}$ 大小，所以有 $7c8b6000 - 4000000 = 788b6000$ ，这就是我们计算的结果。当然会有一定的误差，因为这是我们的理论值。

实际运行分配 64M 之后，可以看到如图 22 所示的结果

1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704	/hon	1.	55daf4304000-55daf4305000	r-xp	00000000	08:01	1180704
2.	55daf4504000-55daf4505000	r-p	00000000	08:01	1180704	/hon	2.	55daf4504000-55daf4505000	r-p	00000000	08:01	1180704
3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704	/hc	3.	55daf4505000-55daf4506000	rw-p	00001000	08:01	1180704
4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0	[hez	4.	55daf50be000-55daf50df000	rw-p	00000000	00:00	0
5.	7f89548b1000-7f895c8b2000	rw-p	00000000	00:00	0		5.	7f89548b1000-7f895c8b2000	rw-p	00000000	00:00	0
6.	7f89648b3000-7f896c8b4000	rw-p	00000000	00:00	0		6.	7f89648b3000-7f896c8b4000	rw-p	00000000	00:00	0
7.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0		7.	7f897c8b6000-7f89848b7000	rw-p	00000000	00:00	0
8.							8.					

图 22. 新分配 64M 之后的虚存变化

在图 22 中我们看到实际上的数值是 788b5000，也就是分配了多一些的内存空间。而多出来的这部分空间导致新分配的内存大小为

$$7c8b6000 - 788b5000 \approx 64.00390625 \text{ MB}$$

可以看出正确地分配了 64M 的空间。

所以综上所述，问题回答如下：

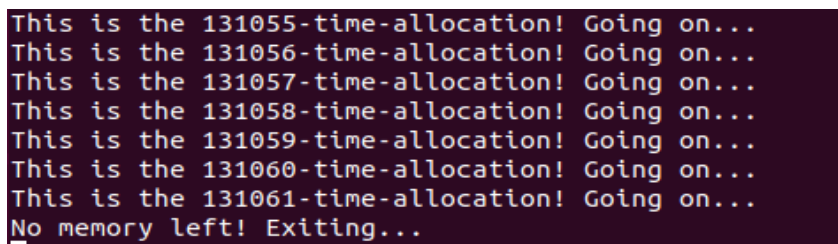
- 1) 内存分配属于连续分配。在每次的计算过程中，计算出来的都是带一些误差的。就以上面的计算为例，我们理论计算数值是 788b6000 而实际是 788b5000。我们经过计算之后发现多出来的 $0x1000$ 实际上是 4KB 的大小。这个大小实际上是页表的大小。再比如我们之前 $0x7f89548b1000 - 0x7f89848b7000$ 表示 6 个 128M 大小，实际上这些地址包括了 6 个页表大小，实际的分配区域为大小为 $0x848b7000 - 0x548b1000 - 6(0x1000) = 0x3000000 = 768 \text{ M} = 128 \text{ M} \times 6$ 所以对于包含页表的时候，分配都是连续的，因为所谓的误差都是由页表造成的。
- 2) 算法采用的是首次适应算法。第一次分配 1024M，从内存区域 1 开始寻找，直到最后一个才找到空间。第二次分配 64M，第一次就找到了空间。如果是最佳适应的话，由于它是想找出能满足作业要求的、且大小最小的空闲分区，这种方法能

使碎片尽量小，那应该从 buffer4 开始扩充，因为 buffer4 和 buffer6 之间只有一个 buffer5 的空间(128M)，而 buffer1 和 buffer4 却有原来 buffer2 和 buffer3 的空间(256M)。既然选择了 buffer1 进行扩充，那么它属于首次适应；

- 3) 碎片是存在的。如图 21 所示，empty 和 freed 的区域都是空的，这些属于碎片。此外，由于离散的分配，必然也会造成一些微小的碎片。
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；我们不知道虚拟内存大小的具体数量级，但是可以确定的是它必然是一个很大的数量级。所以我们每次分配 1G 的大小（不写入），直到不能分配为止。代码如下所示。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. int main ()
5. {
6.     printf("This is process %d\n",getpid());
7.     getchar();
8.
9.     char *buf;
10.    for(long i=0; ; i++)
11.    {
12.        buf = (char *)malloc(1024*1024*1024*sizeof(char)); //1 Gigabit
13.        if (!buf)
14.        {
15.            printf("No memory left! Exiting...\n");
16.            getchar();
17.            break;
18.        }
19.        else
20.        {
21.            printf("This is the %ld-time-allocation! Going on...\n",i);
22.        }
23.    }
24. }
```

我们运行该代码，结果如图 23 所示。



```
This is the 131055-time-allocation! Going on...
This is the 131056-time-allocation! Going on...
This is the 131057-time-allocation! Going on...
This is the 131058-time-allocation! Going on...
This is the 131059-time-allocation! Going on...
This is the 131060-time-allocation! Going on...
This is the 131061-time-allocation! Going on...
No memory left! Exiting...
```

图 23. 访问虚存大小

我们看到连续申请了 131061 次之后申请结束，此时我们查看一下 status 的情况，如图 24 所示。

```
VmPeak: 137437383796 kB
VmSize: 137437383796 kB
VmLck:      0 kB
VmPin:      0 kB
VmHWM: 1049688 kB
VmRSS: 1049688 kB
```

图 24. 申请得到的最大虚存大小

在图 24 中我们看到 VmPeak 的大小为 137437383796KB，约为 131071G。此外我们可以看一下当前的物理内存大小，VmRSS 为 1049688KB，约为 1G。

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。
这部分的代码比较容易，如下所示。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. int main ()
5. {
6.     printf("This is process %d\n",getpid());
7.     getchar();
8.
9.     printf("Before allocating...\n");
10.    getchar();
11.
12.    char *buf = (char *)malloc(1024*1024*256*sizeof(char)); //256 Megabit
13.    printf("After allocating...\n");
14.    getchar();
15.
16.    printf("Now we start to write!\n");
17.    getchar();
18.
19.    //WRITE action
20.    for(i=0; i<1024*1024*256; i+=(4*1024))
21.    {
22.        buf[i] = 1;
23.    }
24.    printf("Writing finish!\n");
25.    getchar();
26.
27.    printf("Now we start to read!\n");
```

```

28.     getchar();
29.
30.     //READ action
31.     for(i=0; i<1024*1024*256; i+=(4*1024))
32.     {
33.         buf[i];
34.     }
35.     printf("Reading finish!\n");
36.     getchar();
37. }

```

接下来我们执行代码。首先，对比分配 256MB 前后虚存和物理内存的变化，如图 25 所示。

1.	VmPeak:	4508 kB
2.	VmSize:	4508 kB
3.	VmLck:	0 kB
4.	VmPin:	0 kB
5.	VmHWM:	712 kB
6.	VmRSS:	712 kB
7.		

1.	VmPeak:	266656 kB
2.	VmSize:	266656 kB
3.	VmLck:	0 kB
4.	VmPin:	0 kB
5.	VmHWM:	712 kB
6.	VmRSS:	712 kB
7.		

图 25. 分配 256M 前后的虚存变化

可以看到物理内存并没有变化，因为我们仅仅是分配了这部分的内存而没有使用。但是虚存已经有了变化，变化大小为 $266656 - 4508 = 262148\text{KB} \approx 256\text{MB}$ ，印证了我们分配了 256M 的虚存。

接下来我们来比较没有写入和写入结束的区别，如图 26 所示。

1.	VmPeak:	266656 kB
2.	VmSize:	266656 kB
3.	VmLck:	0 kB
4.	VmPin:	0 kB
5.	VmHWM:	712 kB
6.	VmRSS:	712 kB
7.		

1.	VmPeak:	266656 kB
2.	VmSize:	266656 kB
3.	VmLck:	0 kB
4.	VmPin:	0 kB
5.	VmHWM:	263504 kB
6.	VmRSS:	263504 kB
7.		

图 26. 写入与未写入的区别

可以看到，物理内存增大了 $263504 - 712 = 262792\text{KB}$ ，这个值约为 256MB，可以看出物理内存变化了约 256M，约等于我们分配的整个空间大小。

下面，我们屏蔽掉写入的操作，仅仅读入数据，差别如图 27 所示。

1.	VmPeak:	266656 kB
2.	VmSize:	266656 kB
3.	VmLck:	0 kB
4.	VmPin:	0 kB
5.	VmHWM:	856 kB
6.	VmRSS:	856 kB
7.		

1.	VmPeak:	266656 kB
2.	VmSize:	266656 kB
3.	VmLck:	0 kB
4.	VmPin:	0 kB
5.	VmHWM:	856 kB
6.	VmRSS:	856 kB
7.		

图 27. 读与未读的区别

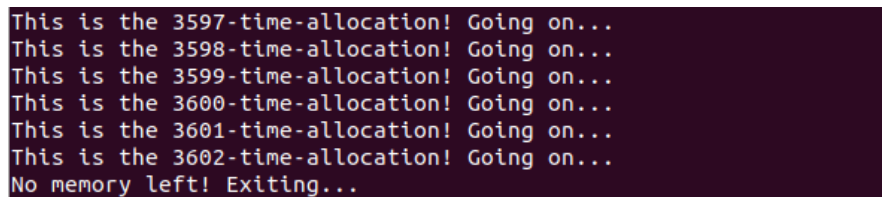
从图中我们可以看到，读操作并没有影响物理内存和虚存。

所以综上所述，我们认为 256M 大小未虚存大小，在写入操作的时候实际使用了物理内存，而读操作并没有影响物理内存。这也启发了我们对第 4 题的思考。如果想要知道电脑的实际物理内存，那么我们需要通过写入的方法占用物理内存，这样才能测得电脑的物理内存大小。

6. **附加题 1.** 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。
- 首先我们需要知道的是当前电脑空间物理内存有多大。为此，我们用之前第 5 题中给出的提示，用写入的方法来占用物理内存。我们每次都分配 1M，然后写入这 1M 的空间，直到不能再分配为止。代码如下所示。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. int main ()
5. {
6.     printf("This is process %d\n",getpid());
7.     getchar();
8.
9.     char *buf[4000];
10.    for(int i=0; ; i++)
11.    {
12.        buf[i] = (char *)malloc(1024*1024*sizeof(char)); //1 Megabit
13.        if (!buf[i])
14.        {
15.            printf("No memory left! Exiting...\n");
16.            getchar();
17.            break;
18.        }
19.        else
20.        {
21.            printf("This is the %d-time-allocation! Going on...\n",i+1);
22.            for(int j=0; j<1024*1024; j++) //写入
23.            {
24.                buf[i][j] = '1';
25.            }
26.        }
27.    }
28. }
```

我们运行该代码，结果如图 28 所示，可以看出空闲物理内存大小为 3602M，即 3.5G 左右。



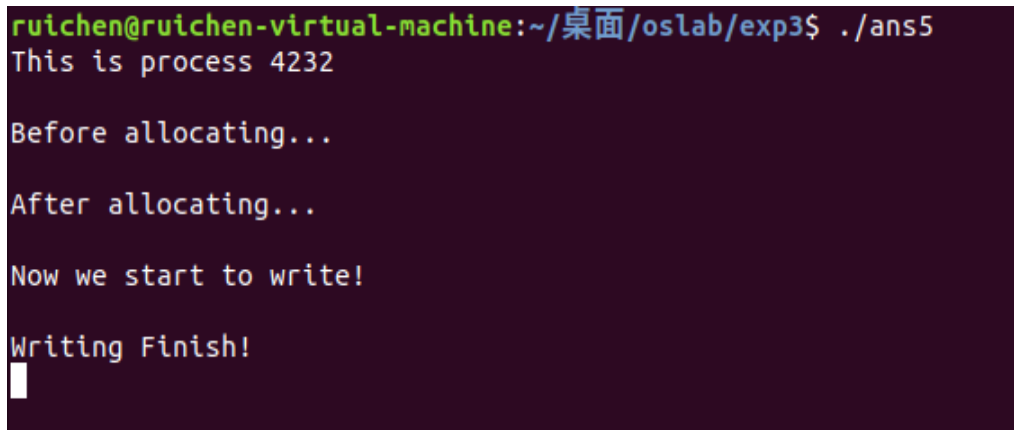
```
This is the 3597-time-allocation! Going on...
This is the 3598-time-allocation! Going on...
This is the 3599-time-allocation! Going on...
This is the 3600-time-allocation! Going on...
This is the 3601-time-allocation! Going on...
This is the 3602-time-allocation! Going on...
No memory left! Exiting...
```

图 28. 测定系统当前最大空闲物理内存

所以我们应该分配 4G 的大小。但是实际上分配过多内存的话程序非常卡顿，我们无法看到习题 5 的运行结果。所以我们仅仅分配 2500M，然后每隔 4KB 间隔将相应地址的字节数值增 1，代码如下所示。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4.
5. int main ()
6. {
7.     printf("This is process %d\n",getpid());
8.     getchar();
9.
10.    //4 Gigabit
11.    char *buf = (char *)malloc(2500LL*1024*1024*sizeof(char));
12.
13.    if(!buf)
14.    {
15.        printf("This is null!\n");
16.    }
17.
18.    printf("Initialization over and programme is going to run!\n");
19.    getchar();
20.
21.    for(long long i=0; i<2500LL*1024*1024; i+=(4*1024))
22.    {
23.        buf[i] = '1';
24.    }
25. }
```

接下来我们先运行第 5 题的代码，我们应该屏蔽掉读操作运行写操作来增大物理内存占用，因为这样才能体现物理内存的竞争。在如图 29 所示的位置暂停程序。



```
ruichen@ruichen-virtual-machine:~/桌面/oslab/exp3$ ./ans5
This is process 4232

Before allocating...

After allocating...

Now we start to write!

Writing Finish!
█
```

图 29. 第 5 题代码暂停位置

然后运行我们附加题一的代码。在写入数据结束之后我们查看习题 5 进程的 status 信息，两次进行对比如图 30 所示。

VmPeak:	266656 kB	VmPeak:	266656 kB
VmSize:	266656 kB	VmSize:	266656 kB
VmLck:	0 kB	VmLck:	0 kB
VmPin:	0 kB	VmPin:	0 kB
VmHWM:	263600 kB	VmHWM:	263600 kB
VmRSS:	263600 kB	VmRSS:	156592 kB

(a) Before preempt

(b) After preempt

图 30. 抢占前与抢占后的对比

我们对比两部分，观察两个进程竞争物理内存的现象，如图 31 所示。

1.	VmPeak:	266656 kB	1.	VmPeak:	266656 kB
2.	VmSize:	266656 kB	2.	VmSize:	266656 kB
3.	VmLck:	0 kB	3.	VmLck:	0 kB
4.	VmPin:	0 kB	4.	VmPin:	0 kB
5.	VmHWM:	263600 kB	5.	VmHWM:	263600 kB
6.	VmRSS:	263600 kB	6.	VmRSS:	156592 kB

图 31. 两个进程竞争物理内存的现象

由此我们看到了内存的抢占现象。原来的进程占据物理内存 263600KB，而现在变为 156592KB，证明有另一个进程抢占了原来进程的物理资源。

为此，我们来看一下新的进程的物理内存，如图 32 所示。

VmPeak:	2564512 kB
VmSize:	2564512 kB
VmLck:	0 kB
VmPin:	0 kB
VmHWM:	55292 kB
VmRSS:	55292 kB

图 32. 新进程的虚存与物理内存

新进程的虚存大小为 2564512KB，物理内存大小为 55292KB.

五、实验体会：

本次实验学会了使用 `proc` 文件系统来查看进程的状态以及映射地址，同时弄清楚了虚存与物理地址之间的区别。通过本次实验，在虚存方面，得知系统内存是连续分配的，而且本系统采用的是首次适应算法，同时测定本台系统的虚存大小约为 131071G，同时结合第 5 问，测定了系统空闲物理内存大概是 3.5G。查看虚拟机的设置，发现内存大小设置为 4G，表示有 0.5G 被系统占用。

内存的分配和回收非常重要。大一的时候学到过 `malloc` 函数的使用，但是当时只知道这是一个分配数组大小的函数，而且也没有释放空间意识。本次实验让我加深了对内存分配函数的理解，同时也明白了释放内存的重要性。总之本次实验从操作系统原理的角度出发让我理解了内存的分配和回收。

指导教师批阅意见：

成绩评定：

指导教师签字：

年 月 日

备注：