



# 本科生实验报告

实验课程 中山大学 2021 学年春季操作系统课程

实验名称 Course Projects

专业名称 计算机科学与技术（超算）

学生姓名 黄玟瑜

学生学号 19335074

任课教师 陈鹏飞

实验地点

实验成绩

二〇二一年七月十三日

# 目录

<b>1 Project 1: 虚拟内存的完善</b>	<b>1</b>
1.1 通过 IO 端口读取硬盘 . . . . .	1
1.1.1 实现过程 . . . . .	1
1.1.2 验证测试 . . . . .	6
1.2 缺页中断 . . . . .	9
1.2.1 异常与错误码 . . . . .	10
1.3 页换入换出机制 . . . . .	17
1.3.1 页面换入换出 . . . . .	17
1.3.2 缺页异常处理 . . . . .	20
<b>2 Project 2: malloc/free 的实现</b>	<b>23</b>
2.1 malloc 的实现 . . . . .	23
2.2 free 的实现 . . . . .	34
2.3 系统调用 malloc 和 free . . . . .	38
2.4 同步和互斥的实现 . . . . .	42
2.4.1 Race Condition 的分析 . . . . .	42
2.4.2 解决方法 . . . . .	45
<b>3 总结</b>	<b>48</b>

## Project 1: 虚拟内存的完善

通过分页机制，我们实现了内存的虚拟化，从而使得程序可以看到比实际的物理内存更大的虚拟内存。虚拟内存另外一个重要的组成部分是页换入换出机制，页换入换出的解释如下。

当内存紧张的时候，我们不得不将一些物理页换出内存，存放到磁盘上，这被称为页的换出。当程序需要使用被换出的页时，程序会产生一个缺页中断。此时，缺页中断处理函数根据页面置换算法将其他某些页换出，然后将程序需要的页从磁盘加载到内存，并修改页表，使得页表重新指向被重新加载到内存的页。这个过程被称为页的换入。

为了简便起见，我们并没有实现页换入换出机制。现在，同学们需要在自己本学期的实验基础上，实现页换入换出机制。在实现了页换入换出机制后，同学们需要自行提供测例来测试页的换出和页的换入。根据测试方法和输出结果来解释自己程序的正确性。最后将结果截图并说说你是怎么做的。

本次 project 的代码放置在 src/PROJECT1 下。

### 通过 IO 端口读取硬盘

#### 实现过程

首先，我们需要实现两个向特定 IO 端口读写一个字的汇编函数。如下。

(src/PROJECT1/src/utils/asm\_utils.asm)

```
1  global asm_inw_port
2  global asm_outw_port
3
4  ; void asm_inw_port(int port, void *value);
5  asm_inw_port:
6      push ebp
7      mov ebp, esp
8
9      push edx
10     push ebx
11
12     xor eax, eax
13     mov edx, dword[ebp + 4 * 2]
14     mov ebx, dword[ebp + 4 * 3]
15     in ax, dx
16     mov word[ebx], ax
17
18     pop ebx
19     pop edx
```

```

20     pop ebp
21     ret
22
23 ; void asm_outw_port(int port, int value);
24 asm_outw_port:
25     push ebp
26     mov ebp, esp
27
28     push edx
29
30     mov edx, dword[ebp + 4 * 2]
31     mov eax, dword[ebp + 4 * 3]
32     out dx, ax
33
34     pop edx
35     pop ebp
36     ret

```

上面的函数的参数的含义和 `asm_in_port`、`asm_out_port` 相同，主要区别在 16 行和 32 行，读写的长度由 8 位改编为 16 位。

然后将这两个函数的声明放置在中。

(src/PROJECT1/include/asm\_utils.h)

```

1 extern "C" void asm_inw_port(int port, void *value);
2 extern "C" void asm_outw_port(int port, int value);

```

随后在 include 目录中创建头文件 `disk.h`，放入磁盘类 `Disk` 的定义。

(src/PROJECT2/include/disk.h)

```

1 #ifndef DISK_H
2 #define DISK_H
3
4 #include "asm_utils.h"
5 #include "os_type.h"
6 #include "stdio.h"
7
8 #define SECTOR_SIZE 512
9
10 class Disk
11 {
12 private:
13     Disk();
14
15 public:
16     // 以扇区为单位写入，每次写入一个扇区
17     // 参数 start: 起始逻辑扇区号
18     // 参数 buf: 待写入的数据的起始地址

```

```

19     static void write(int start, void *buf)
20     {
21         byte *buffer = (byte *)buf;
22         int temp = 0;
23         int high, low;
24
25         // 请求硬盘写入一个扇区, 等待硬盘就绪
26         bool flag = waitForDisk(start, 1, 0x30);
27         if (!flag)
28         {
29             return;
30         }
31
32         for (int i = 0; i < SECTOR_SIZE; i += 2)
33         {
34             high = buffer[i+1];
35             high = high & 0xff;
36             high = high << 8;
37
38             low = buffer[i];
39             low = low & 0xff;
40
41             temp = high | low;
42
43             // 每次需要向0x1f0写入一个字 (2个字节)
44             asm_outw_port(0x1f0, temp);
45             // 硬盘的状态可以从0x1f7读入
46             // 最低位是err位
47             asm_in_port(0x1f7, (uint8 *)&temp);
48
49             if (temp & 0x1)
50             {
51                 asm_in_port(0x1f1, (uint8 *)&temp);
52                 printf("disk error, error code: %x\n", (temp & 0xff));
53                 return;
54             }
55         }
56
57         busyWait();
58     }
59
60     // 以扇区为单位读出, 每次读取一个扇区
61     // 参数 start: 起始逻辑扇区号
62     // 参数 buf: 读出的数据写入的起始地址
63     static void read(int start, void *buf)

```

```

64     {
65         byte *buffer = (byte *)buf;
66         int temp;
67
68         // 请求硬盘读出一个扇区，等待硬盘就绪
69         bool flag = waitForDisk(start, 1, 0x20);
70         if (!flag)
71         {
72             return;
73         }
74
75         for (int i = 0; i < SECTOR_SIZE; i += 2)
76         {
77             // 从0x1f0读入一个字
78             asm_inw_port(0x1f0, buffer + i);
79             // 硬盘的状态可以从0x1F7读入
80             // 最低位是err位
81             asm_in_port(0x1f7, (uint8 *)&temp);
82             if (temp & 0x1)
83             {
84                 asm_in_port(0x1f1, (uint8 *)&temp);
85                 printf("disk error, error code: %x\n", (temp & 0xff));
86                 return;
87             }
88         }
89
90         busyWait();
91     }
92
93 private:
94     // 请求硬盘读取或写入数据，等待硬盘就绪
95     // 参数 start: 待读取或写入的起始扇区的地址
96     // 参数 amount: 读取或写入的扇区数量
97     // 参数 type: 读取或写入的标志，读取=0x20，写入=0x30
98     static bool waitForDisk(int start, int amount, int type)
99     {
100         int temp;
101
102         temp = start;
103
104         // 将要读取的扇区数量写入0x1F2端口
105         asm_out_port(0x1f2, amount);
106
107         // LBA地址7~0
108         asm_out_port(0x1f3, temp & 0xff);

```

```

109
110     // LBA 地址 15~8
111     temp = temp >> 8;
112     asm_out_port(0x1f4, temp & 0xff);
113
114     // LBA 地址 23~16
115     temp = temp >> 8;
116     asm_out_port(0x1f5, temp & 0xff);
117
118     // LBA 地址 27~24
119     temp = temp >> 8;
120     asm_out_port(0x1f6, (temp & 0xf) | 0xe0);
121
122     // 向 0x1f7 端口写入操作类型, 读取=0x20, 写入=0x30
123     asm_out_port(0x1f7, type);
124
125     asm_in_port(0x1f7, (uint8 *)&temp);
126     while ((temp & 0x88) != 0x8)
127     {
128         // 读入硬盘状态
129         if (temp & 0x1)
130         {
131             // 错误码
132             asm_in_port(0x1f1, (uint8 *)&temp);
133             printf("disk error, error code: %x\n", (temp & 0xff));
134             return false;
135         }
136         asm_in_port(0x1f7, (uint8 *)&temp);
137     }
138     return true;
139 }
140
141 static void busyWait() {
142     uint temp = 0xfffff;
143     while(temp) --temp;
144 }
145 };
146
147 #endif

```

先是定义对外的接口读操作和写操作函数。

磁盘写操作 Disk::write, 26 行先等待磁盘就绪。随后 34~41 行, 将缓冲区的两个字节合并后暂存到变量 temp 中; 44 行, 通过上面的接口函数 asm\_outw\_port 将这个字写入磁盘; 47~54 行, 读入磁盘状态, 若写操作出错则输出错误信息。

磁盘读操作 `Disk::read`，69 行等待磁盘就绪。81~87 行，读入磁盘状态，若读操作出错则输出错误信息；90 行进入忙等，磁盘读取完成。

其次是等待硬盘就绪的 `waitForDisk` 函数。

105~123 行，依次写入要读取的扇区数量、磁盘的 LBA 地址、操作类型（读或写）；125~137 行，不断的读取磁盘状态信息，直到磁盘空闲就绪。

最后是忙等函数 `busyWait`。

在内核中使用，在 `setup.cpp` 中加入该头文件。

(src/PROJECT2/src/kernel/setup.cpp)

```
1 #include "disk.h"
```

## 验证测试

测试函数如下。

(src/PROJECT2/src/kernel/setup.cpp)

```
1 extern "C" void setup_kernel()
2 {
3     ...
4     // 创建第一个线程
5     int pid = programManager.executeThread(first_thread, nullptr, "first
        thread", 1);
6     if (pid == -1)
7     {
8         printf("can not execute thread\n");
9         asm_halt();
10    }
11
12    char buffer[SECTOR_SIZE];
13
14    // disk测试代码
15    Disk::read(1, buffer);
16
17    for ( int i = 0; i < SECTOR_SIZE; ++i ) {
18        if(!(buffer[i] & 0xf0)){
19            printf("0");
20        }
21        printf("%x ", buffer[i] & 0xff);
22        if (i % 16 == 15) {
23            printf("\n");
24        }
25    }
26
27    for (int i = 0; i < SECTOR_SIZE; ++i ) {
```



```

28     buffer[i] = i;
29 }
30
31 Disk::write(1, buffer);
32
33 char test_buffer[SECTOR_SIZE];
34 Disk::read(1, test_buffer);
35 for (int i = 0; i < SECTOR_SIZE; ++i ) {
36     if(buffer[i] != test_buffer[i]) {
37         printf("error!\n");
38     }
39 }
40
41 asm_halt();
42
43 ListItem *item = programManager.readyPrograms.front();
44 PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
45 firstThread->status = ProgramStatus::RUNNING;
46 programManager.readyPrograms.pop_front();
47 programManager.running = firstThread;
48 asm_switch_thread(0, firstThread);
49
50 asm_halt();
51 }

```

通过 IO 端口读取硬盘，读出的信息如图 1.1。

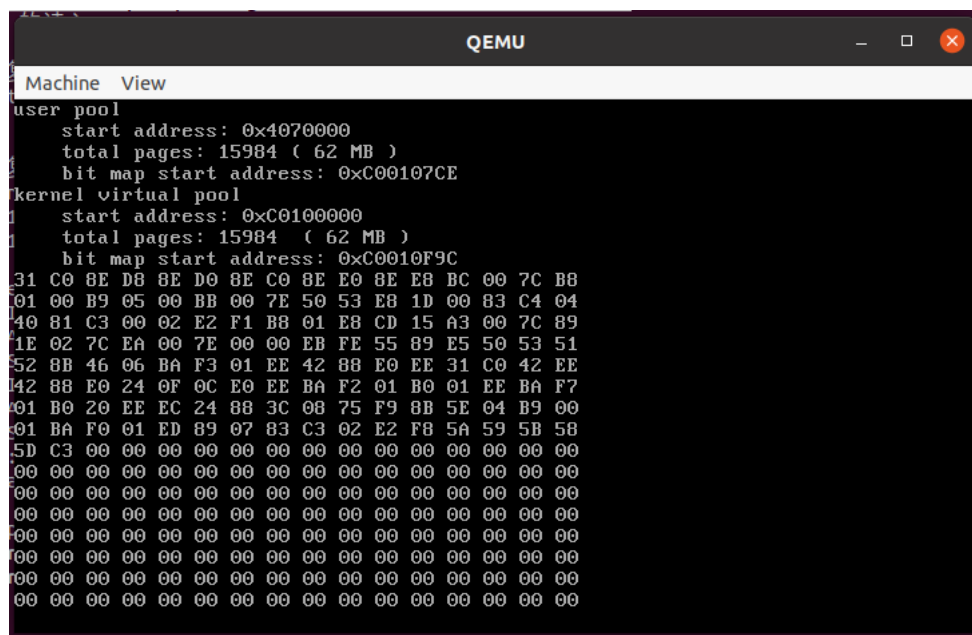


图 1.1: 通过 IO 端口读取 0 号扇区

使用 xxd 查看磁盘信息，如图 1.2。

```
wenny@owo:~/lab9/1/run$ xxd -u -g 1 -l 256 hd.img
00000000: 31 C0 8E D8 8E D0 8E C0 8E E0 8E E8 BC 00 7C B8
00000010: 01 00 B9 05 00 BB 00 7E 50 53 E8 1D 00 83 C4 04
00000020: 40 81 C3 00 02 E2 F1 B8 01 E8 CD 15 A3 00 7C 89
00000030: 1E 02 7C EA 00 7E 00 00 EB FE 55 89 E5 50 53 51
00000040: 52 8B 46 06 BA F3 01 EE 42 88 E0 EE 31 C0 42 EE
00000050: 42 88 E0 24 0F 0C E0 EE BA F2 01 B0 01 EE BA F7
00000060: 01 B0 20 EE EC 24 88 3C 08 75 F9 8B 5E 04 B9 00
00000070: 01 BA F0 01 ED 89 07 83 C3 02 E2 F8 5A 59 5B 58
00000080: 5D C3 00 00 00 00 00 00 00 00 00 00 00 00 00
00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

图 1.2: 磁盘 0 号扇区

对比两张图的信息，可以看到输出的磁盘信息是一样的。

尝试对 bootloader.asm 进行修改，将代码段后填充为 0x99（图 1.3），通过 IO 端口读取 1 号硬盘，读出的信息如图 1.4。

```
147 | | ret
148 |
149 | pgdt dw 0
150 | | dd GDT_START_ADDRESS
151 |
152 | times 512 - ($ - $$) db 0x99
153 | times 512 db 0x88
154 | times 512 db 0x77
155 | times 512 db 0x66
```

图 1.3: bootloader.asm

后面尝试查看 2、3、4 号磁盘的信息，输出信息为全为 0x88、全为 0x77、全为 0x66，符合预期。

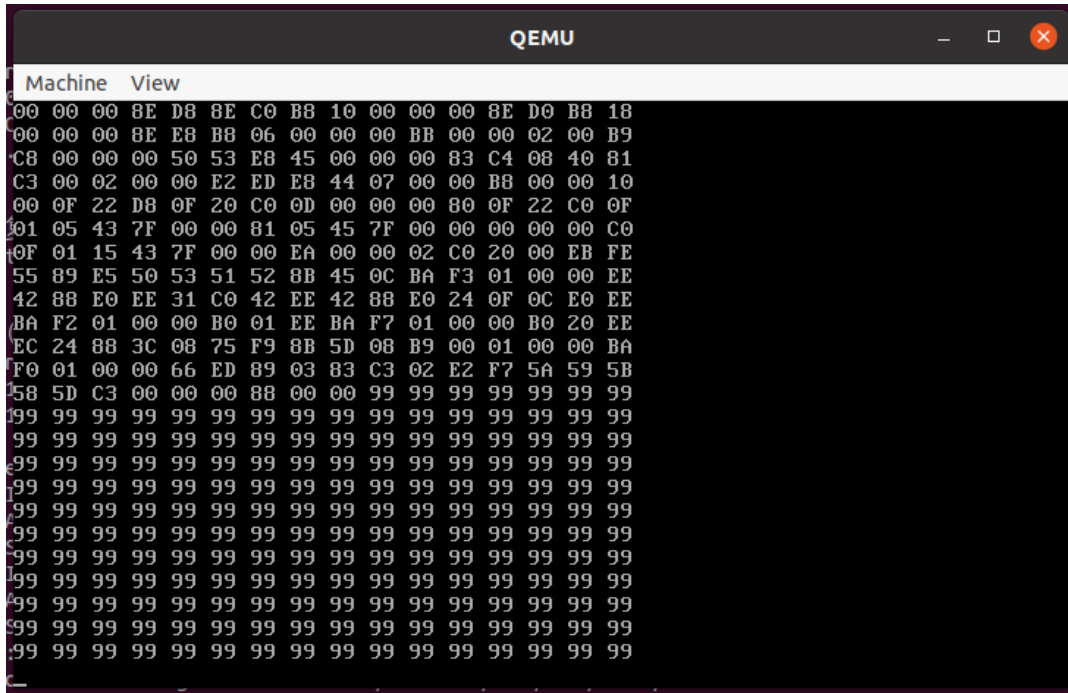


图 1.4: 通过 IO 端口读取 1 号扇区

## 缺页中断

在之前的实验中，我们实现了中断处理机制的基本框架，并根据需要实现了一些中断：80 号系统调用中断、基于 8259A 的时钟中断（中断号 0x20）。其他的中断并未做具体处理，而是统一的 trap 在 `asm_unhandled_interrupt` 中。

中断可分别外部中断和内部中断，内部中断又可分为软中断和异常。软中断，就是由软件主动发起的中断，因为它来自于软件，所以称之为软中断。由于该中断是软件运行中主动发起的，所以它是主观上的，并不是客观上的某种内部错误，可以通过 INT 等指令发起，如我们在实现系统调用时通过发起 80 号中断进入系统调用的中断处理程序 `asm_system_call_handler`。

异常是另一种内部中断，是指令执行期间 CPU 内部产生的错误引起的。按照严重程度，可以分为以下三种：

(1) Fault，也称为故障。这种错误是可以被修复的一种类型，属于最轻的一种异常，它给软件一次“改过自新”的机会。当发生此类异常时 CPU 将机器状态恢复到异常之前的状态，之后调用中断处理程序时，CPU 将返回地址依然指向导致 fault 异常的那条指令。通常中断处理程序中会将此问题修复，待中断处理程序返回后便能重试。最典型的例子就是操作系统课程中所说的缺页异常 page fault，话说 Linux 的虚拟内存就是基于 page fault 的，这充分说明这种异常是极易被修复的，甚至是有益的。

(2) Trap，也称为陷阱，这一名称很形象地说明软件掉进了 CPU 设下的陷阱，导致停了下来。此异常通常用在调试中，比如 `int3` 指令便引发此类异常，为了让中断处理程序返回后能够继续向下执行，CPU 将中断处理程序的返回地址指向导致异常指令

的下一个指令地址。

(3) Abort, 也称为终止, 从名字上看, 这是最严重的异常类型, 一旦出现, 由于错误无法修复, 程序将无法继续运行, 操作系统为了自保, 只能将此程序从进程表中去掉。导致此异常的错误通常是硬件错误, 或者某些系统数据结构出错。

某些异常会有单独的错误码, 即 error code, 进入中断时 CPU 会把它们压在栈中, 这是在压入 eip 之后做的, 如下表所示。

表 1: 异常与中断

Vector No.	Mnemonic	Description	Source	Type	Error code(Y N)
0	#DE	Divide Error	DIV and IDIV instructions.	Fault	N
...	...	...	...	...	...
14	#PF	Page Fault	Any memory reference.	Fault	Y
...	...	...	...	...	...

Error code 为 N 为无错误码, Y 为有错误码。

总得来说, Fault 类型的中断在处理完毕后返回地址为当前指令, Trap 的返回地址为下一条指令, Abort 则直接将进程杀死。异常的中断向量号范围为 0~19, 有些会异常有错误码, 在触发时 CPU 会自动压入, 因此需要手动处理。

## 异常与错误码

在实现缺页中断之前, 通过除零异常体验一下 CPU 的这种异常处理机制。

首先, 定义 C++ 编写的中断处理函数, 其实只是简单的输出错误信息。

(src/PROJECT2/src/kernel/interrupt.cpp)

```
1 // 中断处理函数
2 extern "C" void c_DE_handler()
3 {
4     printf("Devide Error!\n");
5 }
```

随后在汇编中编写入口函数 asm\_DE 保护现场。

(src/PROJECT2/src/utlis/asm\_utils.asm)

```
1 ...
2 extern c_DE_handler
3 ...
4 global asm_DE
5 ...
```

```

6  asm_DE:
7      cli
8      pushad
9      push ds
10     push es
11     push fs
12     push gs
13
14     call c_DE_handler
15
16     pop gs
17     pop fs
18     pop es
19     pop ds
20     popad
21
22     sti
23     iret
24     . . .

```

第 7 行关中断，第 8~12 行将各个寄存器入栈，随后转到中断处理函数，处理完毕后恢复现场，再返回到出现异常的指令。除零异常 Divide Error 是没有错误码的，因此在返回时只需按照正常方式返回，具体表现为在 20 行 popad 恢复各个寄存器的值后不，不再对 esp 进行操作，随后通过 iret 返回。在后面缺页中断的处理会体现这一区别，因为缺页中断是有错误码的。

在头文件 asm\_utils.h 中加入外部函数声明，并且将 asm\_DE 设为 0 号中断处理函数。

(src/PROJECT2/include/asm\_utils.h)

```

1  extern "C" void asm_DE();

```

(src/PROJECT2/src/kernel/interrupt.cpp)

```

1  void InterruptManager::initialize()
2  {
3      . . .
4      for (uint i = 0; i < 256; ++i)
5      {
6          setInterruptDescriptor(i, (uint32)asm_unhandled_interrupt, 0);
7      }
8      // 将asm_DE设为0号中断处理函数
9      setInterruptDescriptor(0, (uint32)asm_DE, 0);
10     . . .
11 }

```

最后在第一个线程中触发除零异常。

(src/PROJECT2/src/kernel/setup.cpp)

```
1 void first_thread(void *arg)
2 {
3     printf("Start first thread.\n");
4     // 触发除零异常
5     int a = 1 / 0;
6     asm_halt();
7 }
```

运行结果如图 1.5。



图 1.5: 除零异常

除零异常属于 Fault 类型的异常，在异常处理完毕后会返回到触发异常的指令，再将它执行一遍，因此虽然在第一个线程中只使用了一条语句来触发，但其每次返回到同样的地址，由于我们只是单纯的输出错误信息，没对导致异常的地方做处理，因此会不断执行除零异常的中断处理函数，表现为不断地重复输出“Devide Error!”。

下面按如上模式添加缺页中断处理函数，仍只是输出错误信息不做处理。

(src/PROJECT2/src/kernel/interrupt.cpp)

```
1 // 中断处理函数
2 extern "C" void c_PF_handler()
3 {
4     printf("Page Fault! \n");
5
6 }
```

最后尝试在第一个线程中触发它。缺页异常该如何触发？

当启动分页机制以后，如果一条指令或数据的虚拟地址所对应的物理页框不在内存中或者访问的类型有错误（比如写一个只读页或用户态程序访问内核态的数据等），就会发生页访问异常。产生页访问异常的原因主要有：

- 目标页帧不存在（页表项全为 0，即该线性地址与物理地址尚未建立映射或者已经撤销）；
- 相应的物理页帧不在内存中（页表项非空，但 Present 标志位 =0，比如在 swap 分区或磁盘文件上），这在本次实验中会出现，我们将在下面介绍换页机制实现时进一步讲解如何处理；
- 不满足访问权限（此时页表项 P 标志 =1，但低权限的程序试图访问高权限的地址空间，或者有程序试图写只读页面）。

在新创建一个进程后，它的页表是全为空的（实际上最后一项指向了页表本身），因此随便访问一个地址都会产生缺页中断，故可编写测试方法如下：

(src/PROJECT2/src/kernel/setup.cpp)

```
1 void first_process()
2 {
3     printf("Start first process.\n");
4     // 设置地址
5     int* a = (int*)0x4000;
6     // 访问该地址
7     int b = *a;
8     printf("%d\n", b);
9
10    asm_halt();
11
12 }
13 void first_thread(void *arg)
14 {
15     printf("Start first thread.\n");
16     programManager.executeProcess((const char *)first_process, 1);
17
18     asm_halt();
19 }
```

运行结果如图 1.6。

看到运行结果中出现了“Page Fault!”，说明缺页中断着实被触发了，但只输出了一条，并不像除零异常那样被重复执行。原因就在于被 CPU 自动压入的错误码。某些异常会有错误码，此错误码用于报告异常是在哪个段上发生的，也就是异常发生的位置，所以错误码中包含选择子等信息。错误码会紧跟在 EIP 之后入栈，记作 ERROR\_CODE。

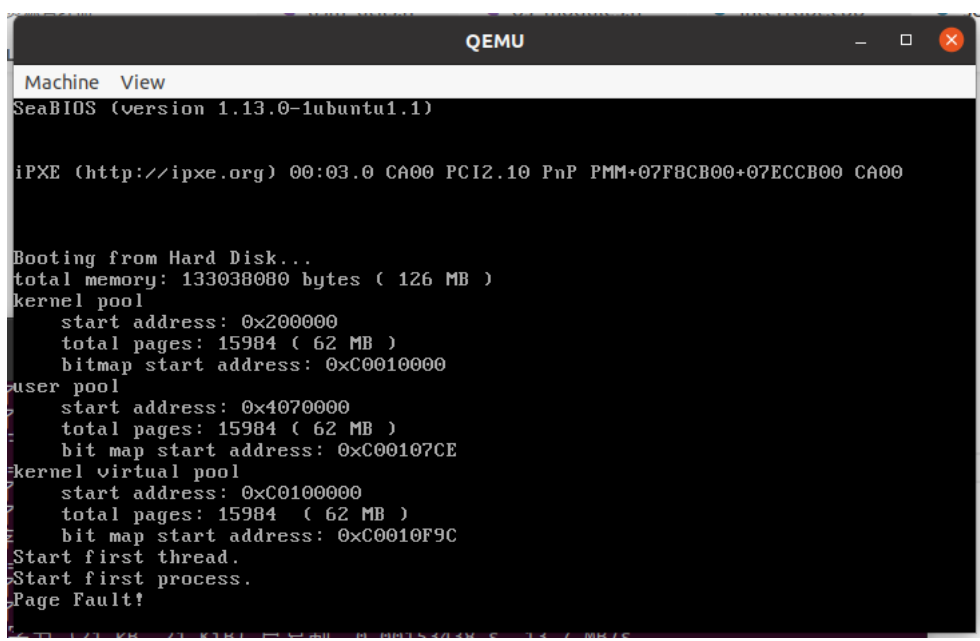


图 1.6: 缺页异常

如图 1.7，在中断发生后，若特权级发生变化，处理器会依次压入旧栈 SS 和 ESP 的值，以便日后依此在 TSS 中找到栈信息恢复 (A)，再压入 EFLAGS 寄存器 (B)，没有错误码时 (C)，有错误码时 (D)。对比 C、D，按原来的方式从中断处理程序退出，返回的是将 ERROR\_CODE 当作地址解析的地方，从而会出现意料之外的结果。

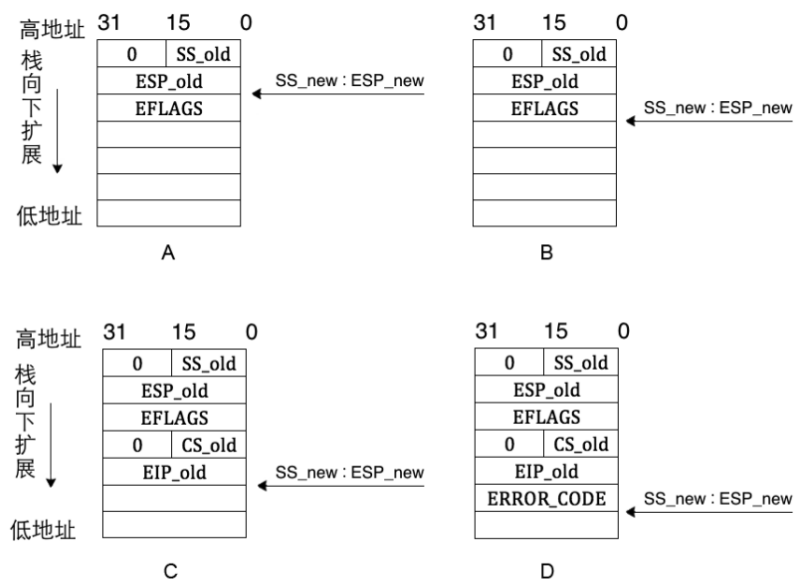


图 1.7: 中断发生，特权级变化时的新栈

而真正的地址应该为 ESP+4。故修改中断入口程序如下：



```

1  ...
2  extern c_PF_handler
3  ...
4  global asm_PF
5  ...
6
7  interrupt_exit:
8  ; 以下是恢复上下文环境
9      add esp, 4          ; 跳过中断号
10
11     pop gs
12     pop fs
13     pop es
14     pop ds
15     popad
16
17     add esp, 4          ; 跳过error_code
18     iretd
19
20  asm_PF:
21     cli
22     pushad
23     push ds
24     push es
25     push fs
26     push gs
27
28     ; 如果是从片上进入的中断,除了往从片上发送EOI外,还要往主片上发送EOI
29     mov al,0x20          ; 中断结束命令EOI
30     out 0xa0,al          ; 向从片发送
31     out 0x20,al          ; 向主片发送
32
33     push 0xe ;压入中断向量号,先不管用不用
34     call c_PF_handler
35
36     jmp interrupt_exit
37     iret
38  ...

```

在中断入口程序 `asm_PF` 中压入了中断向量号（第 33 行），根据前面的所学知识，依照函数调用规则，可以知道，压入的中断向量号即 C 函数第一个参数，尝试在处理函数中输出它的值。如下代码所示。

(src/PROJECT2/src/kernel/interrupt.cpp)

```
1 // 中断处理函数
2 extern "C" void c_PF_handler(uint32 index)
3 {
4     printf("Page Fault! NO.%.d\n", index);
5
6 }
```

继续看中断处理程序和出口函数。因此在退出中断处理程序时，要先跳过中断向量号（第 9 行），随后恢复现场（第 11~15 行），然后再跳过错码（第 17 行），这也是和无错误码的异常的主要区别，最后再通过 `iretd` 返回。

第 18 行，返回指令为 `iretd`。同类的指令还有 `iretw`，16 位模式下用 `iretw`，32 位模式下用 `iretd`。`iret` 是 `iretw` 和 `iretd` 的简写，无论是在 16 位模式，还是在 32 位模式下编码，都可以只用 `iret` 指令，它是被编译成 `iretw`，还是 `iretd`，取决于伪指令 `BITS` 所指明的字长。故在 32 位模式下它相当于 `iret`。

最后运行，结果如图 1.8。

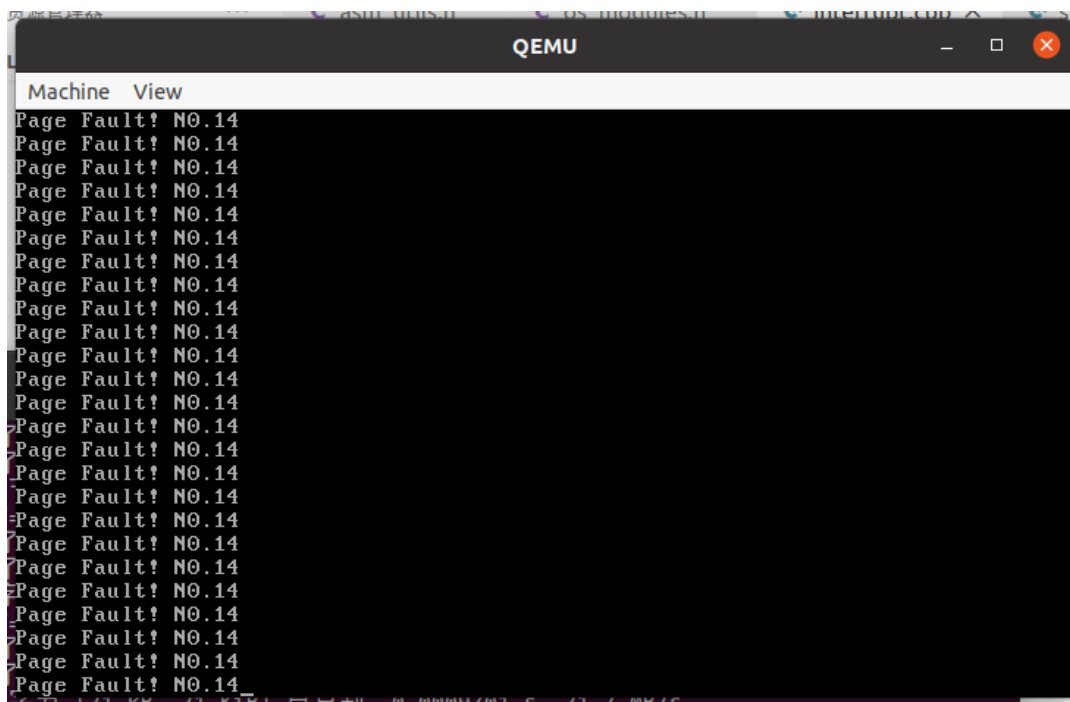


图 1.8: 缺页异常

看到预期的结果，该处理函数被不停重复执行，输出“Page Fault!”和缺页中断的中断向量号“No.14”，该语句将被执行到不再发生缺页异常。这样页换入换出机制的入口部分就处理好了。

## 页换入换出机制

当内存紧张的时候，我们不得不将一些物理页换出内存，存放到磁盘上，这被称为页的换出。当程序需要使用被换出的页时，程序会产生一个缺页中断。此时，缺页中断处理函数根据页面置换算法将其他某些页换出，然后将程序需要的页从磁盘加载到内存，并修改页表，使得页表重新指向被重新加载到内存的页。这个过程被称为页的换入。

在本次实验中采取固定分配局部置换的驻留集管理方案，在创建进程时，分配给进程的页框数是固定的。发生一次缺页中断时，操作系统必须从该进程的驻留页中选择一页用于置换。

### 页面换入换出

为了实现页面的换入和换出，需要一个页面置换管理器 SwapManager，它的定义如下：

(src/PROJECT2/include/swap.h)

```
1  #ifndef SWAP_H
2  #define SWAP_H
3
4  #include "os_type.h"
5  #include "address_pool.h"
6
7  class SwapManager
8  {
9
10     AddressPool hda;
11     int StartSector;
12
13 public:
14     SwapManager();
15
16     void initialize();
17
18     bool createPool(int SectorNum);
19     // 将虚拟地址对应的页换出到磁盘中
20     int swapOut(uint32 virtualAddress);
21     // 将虚拟地址对应的页换入到对应的物理地址的页
22     int swapIn(uint32 virtualAddress, uint32 physicalAddress);
23     // 对缺页异常进行处理
24     void doPageFault();
25 };
26
27 #endif
```

定义了一个主硬盘地址池，用于管理硬盘的扇区。我们的硬盘的大小为 10MB，将前 2MB 用于存储硬盘，将后 8MB 用于存储换出的页，因此可用于存储换出的页的磁盘起始扇区的 LBA 地址为第  $2 * 1024 * 2$ （每个扇区大小为 0.5KB），页面置换管理器的初始化过程如下：

(src/PROJECT2/src/kernel/swap.cpp)

```
1  ...
2  SwapManager::SwapManager(){
3
4  }
5
6  void SwapManager::initialize(){
7      // 前2MB用于存储硬盘，后8MB用于存储换出的页
8      StartSector = 2 * 1024 * 2;
9      bool flag = createPool(8 * 1024 * 2);
10     if(!flag)
11         return ;
12 }
13 ...
```

此外，主硬盘地址池的初始化过程 createPool 过程如下：

(src/PROJECT2/src/kernel/swap.cpp)

```
1  ...
2  bool SwapManager::createPool(int SectorNum)
3  {
4      int sourcesCount = SectorNum / 8;
5      int bitmapLength = ceil(sourcesCount, 8);
6
7      // 计算位图所占的页数
8      int pageCount = ceil(bitmapLength, PAGE_SIZE);
9
10     int start = memoryManager.allocatePages(AddressPoolType::KERNEL,
11                                             pageCount);
12     //printf("%x %d\n", start, pageCount);
13
14     if (!start)
15     {
16         return false;
17     }
18
19     memset((char *)start, 0, PAGE_SIZE * pageCount);
20     hda.initialize((char *)start, sourcesCount, 0);
21
22     return true;
```

```
22 }  
23 ...
```

第4行，主硬盘地址池的一个资源即一个页框的大小4KB，4KB的内容需要8个扇区来存储，因此资源数量为扇区数除以8。

随后，计算位图所需的内存页数，再从内核内存中申请页用于存储位图，将申请到的页初始化，最后对主硬盘地址池进行初始化。

下面是将页换出的函数。

将页面换出，即将该页的内容存到磁盘上，并将占用的物理页释放，从而为其他程序腾出内存空间，将物理页释放的关键在于将P位置0。但是，在进程看来，这个过程是透明的，该虚拟页仍未释放，在进程再次访问该页时，缺页异常处理函数找到该页表项，将LBA地址指向的扇区内容读入物理页框中，再将对应的页表项P位置1，实现页面换入。

(src/PROJECT2/src/kernel/swap.cpp)

```
1  ...  
2  int SwapManager::swapOut(uint32 virtualAddress){  
3      // 关中断  
4      bool status = interruptManager.getInterruptStatus();  
5      interruptManager.disableInterrupt();  
6  
7      // LBA地址为起始扇区地址+资源的下标*8  
8      int lba = StartSector + (hda.allocate(1) * 8);  
9  
10     char *pageSector = (char *)virtualAddress;  
11  
12     // 每次写入一个扇区，即512字节，共写8个扇区  
13     for(int i = 0; i < 8; i++){  
14         Disk::write(lba + i, (void *)pageSector);  
15         pageSector += 512;  
16     }  
17  
18     // 将虚拟地址对应的物理页释放  
19     int pAddr = memoryManager.vaddr2paddr(virtualAddress);  
20     memoryManager.releasePhysicalPages(AddressPoolType::USER, pAddr, 1);  
21  
22     // 修改页表项，将LBA地址存到高20位，并将P位置0  
23     int *pte = (int *)memoryManager.toPTE(virtualAddress);  
24     *pte = *pte & 0xffe; // 将页表项的高20位和P位都清零  
25     *pte = *pte | (lba << 12); // 将lba地址放到高20位  
26  
27     // 恢复中断状态  
28     interruptManager.setInterruptStatus(status);  
29
```

```

30     return pAddr;
31 }
32
33 int SwapManager::swapIn(uint32 virtualAddress, uint32 physicalAddress){
34     // 关中断
35     bool status = interruptManager.getInterruptStatus();
36     interruptManager.disableInterrupt();
37
38     int *pte = (int *)memoryManager.toPTE(virtualAddress);
39     int lba = *pte >> 12;
40     // 将物理页P位置1, 只有这样才能MMU才能寻址到对应的物理页框, 否则仍认为该物理页不存在内存中
41     *pte = (physicalAddress & 0xffff000) | 0x7;    // 将物理页P位置位
42
43     char *pageSector = (char *)virtualAddress;
44
45     // 每次读入一个扇区, 即512字节, 共读8个扇区
46     for(int i = 0; i < 8; i++){
47         Disk::read(lba + i, (void *)pageSector);
48         pageSector += 512;
49     }
50
51     // 将对应的扇区资源释放
52     hda.release((lba - StartSector) / 8, 1);
53     // 恢复中断状态
54     interruptManager.setInterruptStatus(status);
55 }
56 ...

```

第 24 行, 修改页表项时, 将页表项的高 20 位和最后一位 P 位都置 0, 即与上 0xffe, 高 20 位用于存储 LBA 地址。第 38 行, 将页面换入时, 找到该虚拟地址对应的页表项, 随后取出 LBA 地址 (39 行), 需要先将该物理页 P 位置位, 否则 MMU 仍无法找到该物理页框, 随后将磁盘扇区的内容写入该物理页。52 行, 将 LBA 地址所对应的扇区资源释放。

## 缺页异常处理

最后是缺页中断的处理函数, 此函数将采取页面置换算法, 通过调用页面换入换出函数 (swapOut 和 swapIn), 对缺页异常进行处理。

(src/PROJECT2/src/kernel/swap.cpp)

```

1 void SwapManager::doPageFault(){
2     PCB *process = programManager.running;
3     uint32 virtualAddress = asm_get_cr2();

```

```

4     printf("Miss %x!\n", virtualAddress);
5
6     // 若发生缺页时驻留集未满,则说明是读写权限的问题
7     if(process->user_resident_set_size < process->
        user_resident_set_maxsize){
8         printf("Access Error!\n");
9         return ;
10    }
11 }

```

CPU 会把产生异常的虚拟地址存储在 CR2 中，并且把表示页访问异常类型的值（简称页访问异常错误码，errorCode）保存在中断栈中。获取 CR2 的值需要用到汇编语言工具 asm\_get\_cr2。它将返回当前 CR2 寄存器的值。

(src/PROJECT2/src/include/asm\_utils.h)

```

1 extern "C" uint32 asm_get_cr2();

```

(src/PROJECT2/src/utlis/asm\_utils.asm)

```

1 global asm_get_cr2
2
3 asm_get_cr2:
4     mov eax, cr2
5     ret

```

最后对上述函数进行测试，测试方法如下：

(src/PROJECT2/src/kernel/setup.cpp)

```

1 ...
2 void first_process()
3 {
4     printf("Start first process.\n");
5     int* a = (int*)0x4000;
6     int b = *a;
7     printf("%d\n", b);
8
9     asm_halt();
10
11 }
12 void first_thread(void *arg)
13 {
14     printf("Start first thread.\n");
15     programManager.executeProcess((const char *)first_process, 1);
16     asm_halt();
17 }
18 ...

```

运行结果如图 1.9。

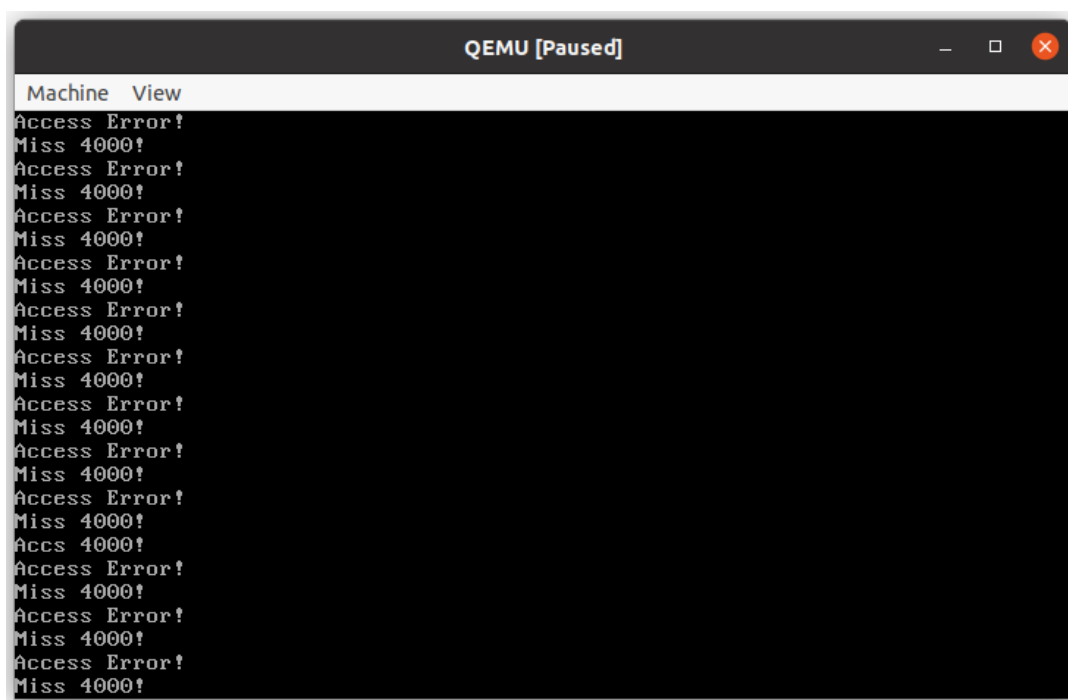


图 1.9: 缺页异常

由于时间问题（个人能力有限 QAQ）没有继续实现页面置换算法。



## Project 2: malloc/free 的实现

我们已经实现了以页为粒度的动态内存分配和释放。但是，我们在程序中使用的往往是以字节为粒度的动态内存管理机制，即我们可以分配和释放任意字节长度的内存。

在本项目中，同学们需要实现系统调用 malloc 和 free。malloc 用于分配任意字节的内存，free 用于释放任意字节的内存。在实现了 malloc 和 free 后，同学们需要自行提供测例来测试 malloc 和 free。根据测试方法和输出结果来解释自己程序的正确性。最后将结果截图并说说你是怎么做的。

本次 project 的代码放置在 src/PROJECT2 下。

注：该 Project 是在 lab9 实验指导更新前完成的，参考了郑刚《操作系统真象还原》中 malloc 和 free 的实现方法。

### malloc 的实现

引用一个新的名词：“arena”，该单词的意思是“舞台”。arena 是很多开源项目中都会用到的内存管理概念，将一大块内存划分成多个小内存块，每个小内存块之间互不干涉，可以分别管理，这样众多的小内存块就称为 arena。

我们将在原有内存管理系统的基础上实现 arena，原有系统只能分配 4KB 粒度的内存页框，根据请求的内存量的大小，arena 的大小也许是 1 个页框，也许是多个页框（一定是页框的整数倍），随后再将它们平均拆分成多个小内存块。按内存块的大小，可以划分出多种不同规格的 arena，比如一种 arena 中全是 16 字节大小的内存块，故它只响应 16 字节以内的内存分配，另一种 arena 中全是 32 字节的内存块，故它只响应 32 字节以内的内存分配。

平时调用 malloc 申请内存时，操作系统返回的地址其实就是某个内存块的起始地址，操作系统会根据 malloc 申请的内存大小来选择不规格内存块。因此，为支持多种容量内存块的分配，我们要提前建立好多种不同容量内存块的 arena。

arena 是个提供内存分配的数据结构（见图 2.1），它分为两部分，一部分是元信息，用来描述自己内存池中空闲内存块数量，这其中包括内存块描述符指针（后面介绍），通过它可以间接获知本 arena 所包含内存块的规格大小，此部分占用的空间是固定的，约为 12 字节。另一部分就是内存池区域，这里面有无数的内存块，此部分占用 arena 大量的空间。我们把每个内存块命名为 mem\_block，它们是内存分配粒度更细的资源，最终为用户分配的就是这其中的一个内存块。

为某一类型内存块提供内存的 arena 只有 1 个，当此 arena 中的全部内存块都被分配完时，系统将再创建一个同规格的 arena 继续提供该规格的内存块，当此 arena 又被分配完时，再继续创建出同规格的 arena，arena 规模逐渐增大，逐步形成 arena 集群。

既然同一类内存块可以由多个 arena 提供，为了跟踪每个 arena 中的空闲内存块，

## 内存块规格为64字节的arena

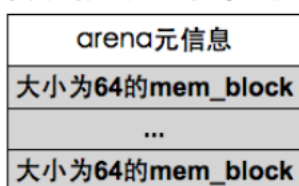


图 2.1: arena 结构

## mem\_block\_desc

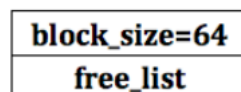


图 2.2: 规格为 64 字节的内存块描述符

分别为每一种规格的内存块建立一个内存块描述符，即 `mem_block_desc`（见图 2.2），在其中记录内存块规格大小，以及位于所有同类 arena 中的空闲内存块链表。（如果申请的内存量较大，超过 1024 字节，单独的一个小内存块无法满足需求时，直接将整块大内存分配出去，此类 arena 没有对应的内存块描述符，元信息中的内存块描述符指针为空。见图 2.3。）

## 申请的内存大于1024字节时



图 2.3: 规格大于 1024 字节的 arena

总结，在内存管理系统中，arena 为任意大小内存的分配提供了统一的接口，它既支持 1024 字节以下的小块内存的分配，又支持大于 1024 字节以上的大块内存，malloc 函数实际上就是通过 arena 申请这些内存块。

据此定义内存块、arena、内存块描述符的结构。

(src/PROJECT2/include/memory.h)

```

1  /* 内存块 */
2  struct mem_block {
3      ListItem free_elem;
4  };
5
6  /* 内存块描述符 */
7  struct mem_block_desc {
8      int block_size;        // 内存块大小
9      int blocks_per_arena;  // 本arena中可容纳此mem_block的数量.
10     List free_list;        // 目前可用的mem_block链表
11 };
12
13 /* 内存仓库arena元信息 */

```

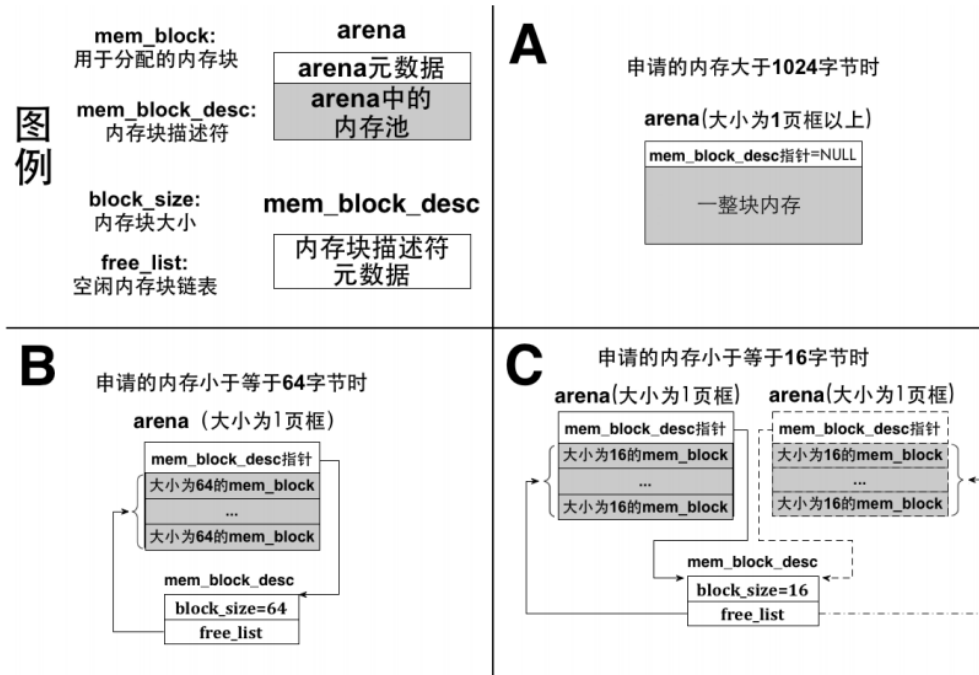


图 2.4: 内存块、arena、内存块描述符的结构

```

14 struct arena {
15     mem_block_desc* desc;    // 此arena关联的mem_block_desc
16     /* large为ture时,cnt表示的是页框数。
17      * 否则cnt表示空闲mem_block数量 */
18     int cnt;
19     bool large;
20 };

```

将实现的 malloc 的可供应的内存块规格有 7 种，以 16 字节为起始，向上依次是 32 字节、64 字节、128 字节、256 字节、512 字节、1024 字节，因此，内存块描述符也就这 7 种。因此设置内存块描述符个数为 7。

(src/PROJECT2/include/os\_constant.h)

```

1 #define DESC_CNT 7

```

下面来考虑如何实现上述内存块、arena、内存块描述符结构之间的关系。

每个线程/进程都有自己的堆,故每个 PCB 都应拥有自己的一套完整内存块、arena、内存块描述符构成的体系,处于这套体系中最顶层的是内存块描述符,线程/进程在 malloc 时最先访问的是它,因此每个 PCB 结构中都包含一个长度为 DESC\_CNT 的内存块描述符数组。而内存块和 arena 是在线程/进程申请时动态产生的,因此不需要放在 PCB 内。

只有一个内核虚拟地址池,故内核只有一个堆,把内核的内存块描述符数组放在内存管理器 MemoryManager 中。

(src/PROJECT2/include/memory.h)

```

1  class MemoryManager
2  {
3  public:
4      // 可管理的内存容量
5      int totalMemory;
6      // 内核物理地址池
7      AddressPool kernelPhysical;
8      // 用户物理地址池
9      AddressPool userPhysical;
10     // 内核虚拟地址池
11     AddressPool kernelVirtual;
12
13     // 内核内存块描述符数组
14     mem_block_desc kernel_block_descs[DESC_CNT];
15
16 public:
17     ...
18
19 };

```

梳理前面所述各个结构体之间的关系，如图 2.5、2.6。

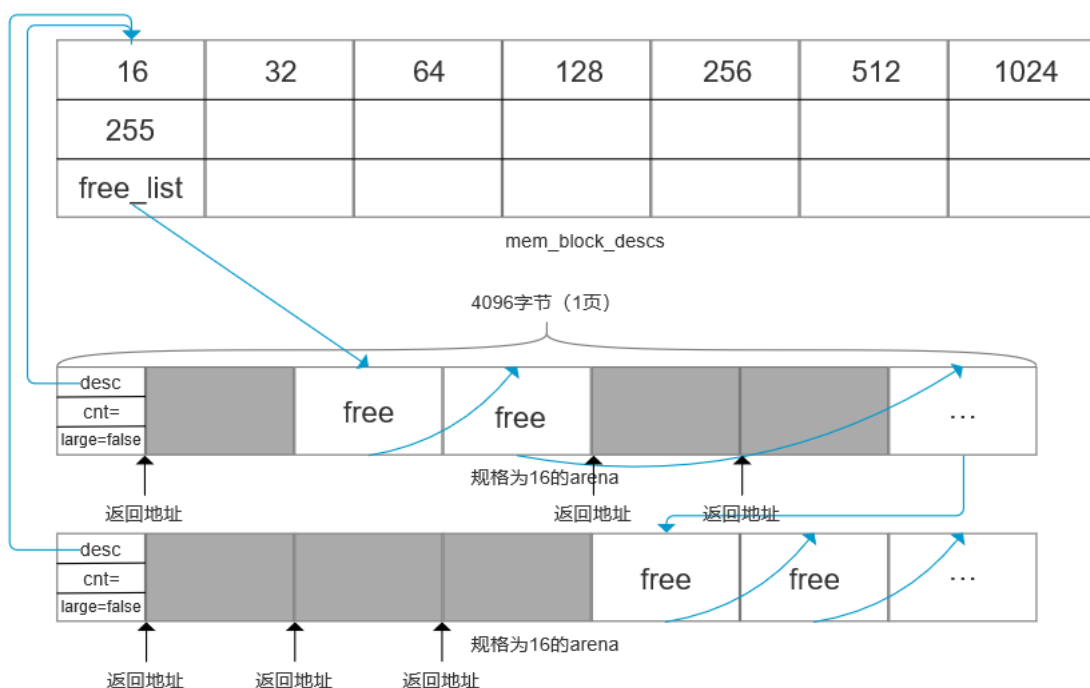


图 2.5: 内存块、arena、内存块描述符关系图 (size ≤ 1024)

内存块规格为 16 字节的内存块描述符内 `blocks_per_arena` 为  $\lfloor (4096 - 12) / 16 \rfloor = 255$ ，即（页框大小-arena 元信息大小）/内存块规格大小，再向下取整，其他规格以此类推。

arena 结构中第 1 个成员是 desc，它指向本 arena 中的内存块被关联到哪个内存块描述符，同一规格的 arena 只能关联到同一规格的内存块描述符，比如本 arena 中的内存块规格为 16 字节，desc 只能指向规格为 16 字节的内存块描述符。第 2 个成员是 cnt，它的意义要取决于第 3 个成员 large 的值。当 large 为 true 时，cnt 表示的是本 arena 占用的页框数，否则 large 为 false 时，cnt 表示本 arena 中还有多少空闲内存块可用，将来释放内存时要用到此项。

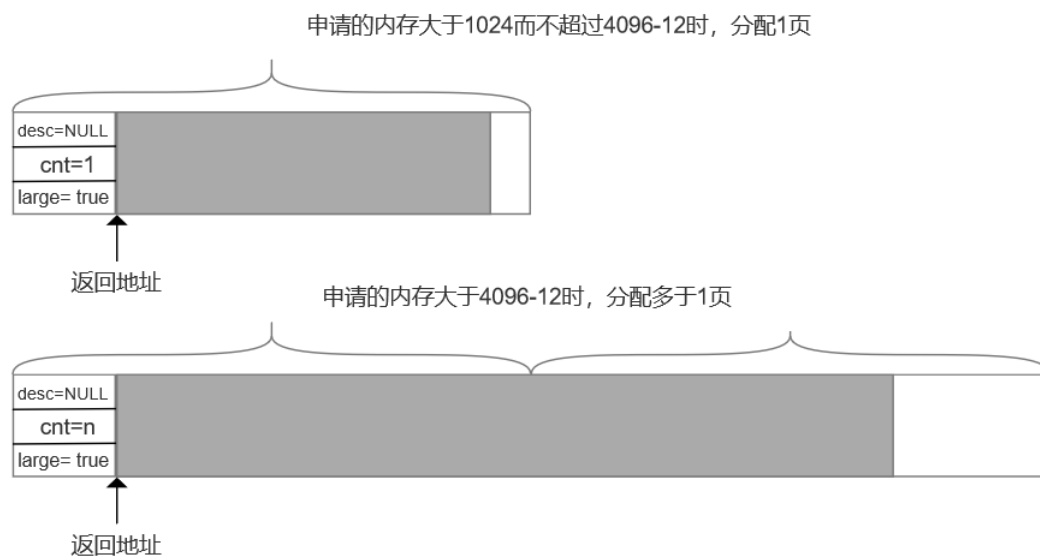


图 2.6: 内存块、arena、内存块描述符关系图 (size > 1024)

在使用 malloc 前需要对内存块描述符数组进行初始化，如下：

(src/PROJECT2/src/kernel/memory.cpp)

```
1 void MemoryManager::block_desc_init(mem_block_desc* desc_array) {
2     int block_size = 16;
3
4     /* 初始化每个mem_block_desc描述符 */
5     for (int desc_idx = 0; desc_idx < DESC_CNT; desc_idx++) {
6         desc_array[desc_idx].block_size = block_size;
7
8         /* 初始化arena中的内存块数量 */
9         desc_array[desc_idx].blocks_per_arena = (PAGE_SIZE - sizeof(arena)
10            ) / block_size;
11         desc_array[desc_idx].free_list.initialize();
12
13         block_size *= 2;          // 更新为下一个规格内存块
14     }
15 }
```

在内存管理器的初始化函数中调用上述函数初始化内核线程的内存块描述符数组。

(src/PROJECT2/src/kernel/memory.cpp)

```

1 void MemoryManager::initialize()
2 {
3     ...
4     /* 初始化kernel_block_descs */
5     block_desc_init(kernel_block_descs);
6 }

```

至此完成了 malloc 实现的基础部分的构建，下面来完成 malloc。

定义两个函数 arena2block 和 block2arena，用于内存块和它对应的 arena 元信息的地址相互索取，根据图 2.5、2.6，它们之间映射的逻辑不难理解。

(src/PROJECT2/src/kernel/memory.cpp)

```

1 mem_block* MemoryManager::arena2block(arena* a, int idx) {
2     return (mem_block*)((int)a + sizeof(arena) + idx * (a->desc->
3         block_size)); /* 返回arena中第idx个内存块的地址 */
4 }
5 arena* MemoryManager::block2arena(mem_block* b) {
6     return (arena*)((int)b & 0xfffff000); /* 返回内存块b所在的arena地
7         址 */
8 }

```

下面是 malloc 的定义。

(src/PROJECT2/src/kernel/memory.cpp)

```

1 int MemoryManager::malloc(const int size) {
2     // 关中断，防止malloc的过程被打断
3     bool status = interruptManager.getInterruptStatus();
4     interruptManager.disableInterrupt();
5
6     enum AddressPoolType type;
7     int pool_size;
8     mem_block_desc* desc;
9     PCB* cur = programManager.running;
10
11     /* 判断用哪个内存池 */
12     if (!(cur->pageDirectoryAddress)) { // 若为内核线程
13         type = AddressPoolType::KERNEL;
14         pool_size = this->kernelPhysical.resources.length;
15         desc = kernel_block_descs;
16     }
17     else { // 用户进程pcb中的pgdir会在为其分配页表时创建
18         type = AddressPoolType::USER;
19         pool_size = this->userPhysical.resources.length;
20         desc = cur->user_block_descs;
21     }
22

```

```

23  /* 若申请的内存不在内存池容量范围内则直接返回0 */
24  if (!(size > 0 && size < pool_size)) {
25      // 恢复中断
26      interruptManager.setInterruptStatus(status);
27      return 0;
28  }
29
30  arena* a;
31  mem_block* b;
32  /* 超过最大内存块1024, 就分配页框 */
33  if (size > 1024) {
34      int page_cnt = ceil(size + sizeof(arena), PAGE_SIZE);    // 向上
35      // 取整需要的页框数
36      a = (arena*)allocatePages(type, page_cnt);
37
38      if (!a) {
39          interruptManager.setInterruptStatus(status);
40          return 0;
41      }
42      else {
43          memset(a, 0, page_cnt * PAGE_SIZE);    // 将分配的内存清0
44          /* 对于分配的大块页框, 将desc置为0, cnt置为页框数, large置为
45             true */
46          a->desc = 0;
47          a->cnt = page_cnt;
48          a->large = true;
49
50          // printf("%x\n", a);
51          interruptManager.setInterruptStatus(status);
52          return (int)(a + 1);    // 跨过arena大小, 把剩下的内存返回
53      }
54  }
55  else
56  {    // 若申请的内存小于等于1024, 可在各种规格的mem_block_desc中去适配
57      int desc_idx;
58      /* 从内存块描述符中匹配合适的内存块规格 */
59      for (desc_idx = 0; desc_idx < DESC_CNT; desc_idx++) {
60          if (size <= desc_idx[desc_idx].block_size) {    // 从小往大后, 找到后退出
61              break;
62          }
63      }
64      // printf(" %d\n", desc_idx);

```

```

65  /* 若mem_block_desc的free_list中已经没有可用的mem_block,
66  * 就创建新的arena提供mem_block */
67      if(descs[desc_idx].free_list.empty()){
68          a = (arena*)allocatePages(type, 1);          // 分配1页框做为
              arena
69          if(!a){
70              interruptManager.setInterruptStatus(status);
71              return 0;
72          }
73          // printf("a: %x\n", a);
74
75          memset(a, 0, PAGE_SIZE);
76
77          /* 对于分配的小块内存,将desc置为相应内存块描述符,
78          * cnt置为此arena可用的内存块数,large置为false */
79          a->desc = &descs[desc_idx];
80          a->large = false;
81          a->cnt = descs[desc_idx].blocks_per_arena;
82
83          // printf("a->desc: %x\n a->cnt: %d\n", a->desc, a->cnt);
84          /* 开始将arena拆分成内存块,并添加到内存块描述符的free_list中
            */
85          for (int block_idx = 0; block_idx < descs[desc_idx].
              blocks_per_arena; block_idx++) {
86              b = arena2block(a, block_idx);
87              // 确保未被分配过
88              if(descs[desc_idx].free_list.find(&(b->free_elem)) !=
                  -1){
89                  releasePages(type, (int)a, 1);
90                  interruptManager.setInterruptStatus(status);
91                  return 0;
92              }
93              descs[desc_idx].free_list.push_back(&(b->free_elem));
94          }
95      }
96
97      /* 开始分配内存块 */
98      ListItem* item = descs[desc_idx].free_list.front();
99      b = ListItem2Block(item, free_elem);
100      descs[desc_idx].free_list.pop_front();
101
102      memset(b, 0, descs[desc_idx].block_size);
103
104      a = block2arena(b); // 获取内存块b所在的arena
105      a->cnt--;           // 将此arena中的空闲内存块数减1

```



```

106
107         interruptManager.setInterruptStatus(status);
108         return (int)b;
109     }
110 }

```

malloc 只有一个参数 size, size 是申请的内存字节数。函数开头定义了一些变量: type、pool\_size 和 desc, 它们的值由内存申请者来决定, 这里的内存申请者包括内核线程和用户进程两种, 第 12~21 行针对这两种情况为它们赋值。

首先判断如果申请的内存量大于 1024 字节, 先计算内存量 size 需要的页框数, ceil 除法向上取整, 计算出的页框数存入变量 page\_cnt。把申请到的页框初始化为 0, 对大内存直接返回 arena 的内存区的首地址, 不需要再将其拆分成小内存块, 因此没有对应的内存块描述符, 故 “a->desc = 0”。a->cnt 此时的意义是此 arena 占用的页框数, 因此 a->cnt = page\_cnt。“a->large = true” 表示此 arena 用于处理大于 1024 字节以上的内存分配。arena 中可被用户使用的部分是内存池部分, 也就是要跨过 arena 前面的元信息部分, 故在第 50 行, 用 “(a+1)” 跨过 arena 元信息, 也就是跨过一个 arena 结构体的大小。最后通过 “return (int)(a + 1)” 把 arena 中的内存池起始地址返回, 此地址便是为用户分配的内存地址。

第 54 行处理内存小于 1024 字节的情况。第 58~61 我们有 7 种规格的内存块描述符, 把它们都遍历一次, 肯定能找到合适的内存块。找到后退出循环, desc\_idx 便是最合适的内存块索引。

在分配之前先要判断是否有可用的内存块, 这里是通过内存块描述符中的 free\_list 是否为空判断的, 若为空表示此规格大小的内存块已经没有了, 此时需要再创建新的 arena。申请新的一页并将其初始化为 0 作为新的 arena, 将它的 desc 指向对应的内存块描述符, a->large 置为 false, 表示此 arena 不用于处理大于 1024 字节的大内存。a->cnt 置为 desc[desc\_idx].blocks\_per\_arena, 表示此 arena 现在具有的空闲内存块数量。后面会看到, 随着以后的分配, 会将 a->cnt 减少。

在创建新的 arena 后, 下一步是将它拆分成内存块, 此部分是在第 411 行的 for 循环开始的, 循环次数是 desc[desc\_idx].blocks\_per\_arena, 这表示此 arena 将被拆分成内存块数量。拆分内存块是通过 arena2block 函数完成的, 它在 arena 中按照内存块的索引 block\_idx 拆分出相应的内存块。指针 b 指向每次新拆分出来的内存块, 然后将其添加到内存块描述符的 free\_list 中。

下面开始分配内存块, 内存块被汇总在内存块描述符的 free\_list 中, 前面的步骤保证了内存块描述符的 free\_list 必不为空。从 free\_list 取出队头元素, 它仅仅是内存块 mem\_block 中 list\_elem 的地址, 需要将其转化为 mem\_block 的起始地址, 用到宏 ListItem2Block, 它的原理和之前宏 ListItem2PCB 相似, 故不赘述。

(src/PROJECT2/include/memory.h)

```

1  #define ListItem2Block(ADDRESS, LIST_ITEM) \
2  ((mem_block *)((int)(ADDRESS) - (int)&((mem_block *)0)->LIST_ITEM))

```

第 104 行通过函数 `block2arena(b)` 获取内存块 `b` 所在的 `arena` 地址, 然后将 `a->cnt` 减 1, 表示空闲内存块少了一个。此项是供将来释放内存使用的, 释放内存时会参考 `cnt` 的值, 用来判断是将 `mem_block` 回收到内存块描述符的 `free_list` 中, 还是直接释放内存块所在的 `arena`。

第 108 行将内存块 `b` 的地址转换成 `int` 后返回, 此地址便是用户进程得到的内存地址。

至此完成了 `malloc`, 将上述函数的定义添加进内存管理器 `MemoryManager` 的定义, 并将其定义为系统调用。

(src/PROJECT2/include/memory.h)

```
1  class MemoryManager
2  {
3  public:
4  ...
5      // 内核内存块描述符数组
6      mem_block_desc kernel_block_descs[DESC_CNT];
7  public:
8  ...
9      // 为malloc做准备
10     void block_desc_init(mem_block_desc* desc_array);
11
12     // 返回arena中第idx个内存块的地址
13     mem_block* arena2block(arena* a, const int idx);
14
15     // 返回内存块b所在的arena地址
16     arena* block2arena(mem_block* b);
17
18     // 在堆中申请size字节内存
19     int malloc(const int size);
20 };
```

(src/PROJECT2/include/syscall.h)

```
1  int syscall_malloc(const int size);
```

(src/PROJECT2/src/kernel/syscall.cpp)

```
1  int syscall_malloc(const int size){
2      return memoryManager.malloc(size);
3  }
```

在启动内核时, 在内核线程中对齐进行测试, 测试方法如下:

(src/PROJECT2/src/kernel/setup.cpp)

```
1  void second_thread(void *arg){
2      printf("start second thread.\n");
3
4      int addr2 = syscall_malloc(63);
```

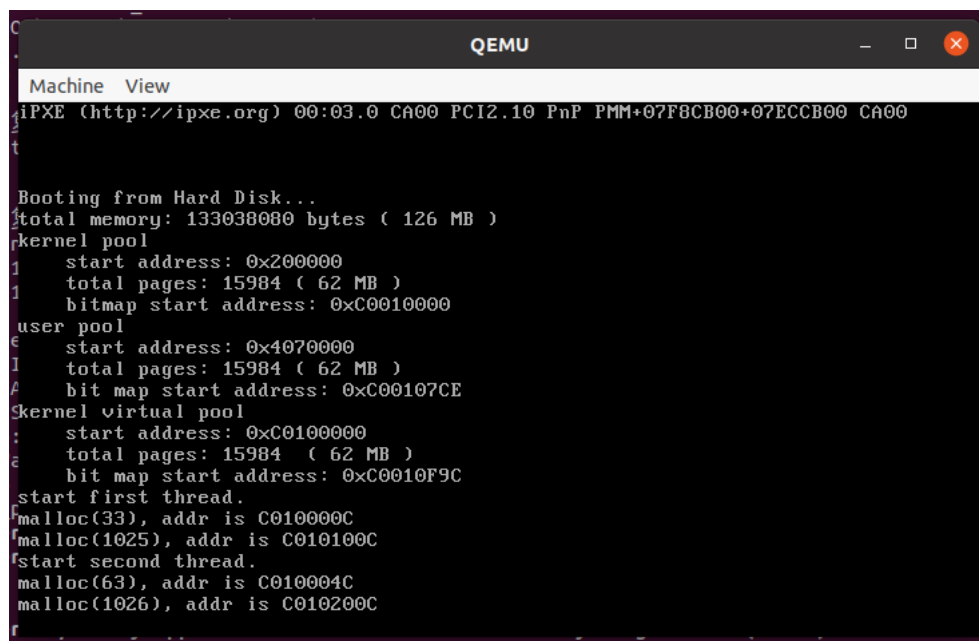
```

5     printf("malloc(63), addr is %x\n", addr2);
6
7     addr2 = syscall_malloc(1026);
8     printf("malloc(1026), addr is %x\n", addr2);
9
10    asm_halt();
11 }
12
13 void first_thread(void *arg)
14 {
15     printf("start first thread.\n");
16     programManager.executeThread(second_thread, nullptr, "second thread"
17         , 1);
18
19     int addr1 = syscall_malloc(33);
20     printf("malloc(33), addr is %x\n", addr1);
21
22     addr1 = syscall_malloc(1025);
23     printf("malloc(1025), addr is %x\n", addr1);
24
25     asm_halt();
26 }

```

按照我们的实现思路，这两个内核线程将会申请 2 个 64 字节大小的内存块和 2 个一整页除了 arena 元信息外的内存块。

运行结果如图 2.7。



The screenshot shows a QEMU window titled "QEMU" with a "Machine View" tab. The output text is as follows:

```

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start first thread.
malloc(33), addr is C010000C
malloc(1025), addr is C010100C
start second thread.
malloc(63), addr is C010004C
malloc(1026), addr is C010200C

```

图 2.7: 运行结果

内核的虚拟起始位置为 0xC0100000，因此第 1 个线程申请到的第一个页的起始地址为 0xC0100000，即第一个 arena 所在位置，返回了 64 字节的内存块的起始地址为 0xC010000C，它是第一个 arena 的内存池的起始地址，即该 arena 越过 arena 元信息的地址。后面第 2 个线程也申请了 64 字节的内存块，也由此 arena 提供，返回的是该 arena 的第二个内存块的地址，起始地址为第一个内存块地址 +64，即 0xC010004C。

第 1 个线程申请 1025 个字节，申请的内存大于 1024 时直接分配 1 页，也是内核线程请求的第二个虚拟页，因此起始地址为 0xC0101000 + 12 = 0xC010100C。第 2 个线程申请 1026 个字节时也再分配 1 页，同理，起始地址为 0xC0102000 + 12 = 0xC010200C。

在此处仅实现了内核态下的 malloc，下面先实现内核态的 free，最后再实现用户态的 malloc 和 free。用户态的 malloc 和 free 只需在内核态的基础上稍加修改。

## free 的实现

实现 free 的代码如下：

(src/PROJECT2/src/kernel/memory.cpp)

```
1 void MemoryManager::free(const int ptr) {
2     if (!ptr)
3         return ;
4
5     // 关中断，防止free的过程被打断
6     bool status = interruptManager.getInterruptStatus();
7     interruptManager.disableInterrupt();
8
9     enum AddressPoolType type;
10    PCB* cur = programManager.running;
11
12    /* 判断是线程还是进程 */
13    if (!(cur->pageDirectoryAddress)) { // 若为内核线程
14        type = AddressPoolType::KERNEL;
15    }
16    else { // 用户进程pcb中的pgdir会在为其分配页表时创建
17        type = AddressPoolType::USER;
18    }
19
20    mem_block* b = (mem_block*)ptr;
21    arena* a = block2arena(b); // 把mem_block转换成arena,获取元信息
22
23    if (a->desc == 0 && a->large == true) { // 大于1024的内存
24        releasePages(type, (int)a, a->cnt);
25    }
26    else
27    { // 小于等于1024的内存块
```

```

28      /* 先将内存块回收到free_list */
29      a->desc->free_list.push_back(&(b->free_elem));
30
31      /* 再判断此arena中的内存块是否都是空闲,如果是就释放arena */
32      if (++a->cnt == a->desc->blocks_per_arena) { //如果该arena空闲
33          for (int block_idx = 0; block_idx < a->desc->
34              blocks_per_arena; block_idx++) {
35              b = arena2block(a, block_idx);
36              if(a->desc->free_list.find(&(b->free_elem))==-1){
37                  interruptManager.setInterruptStatus(status);
38                  return;
39              }
40              a->desc->free_list.erase(&(b->free_elem));
41          }
42          releasePages(type, (int)a, 1);
43      }
44
45      interruptManager.setInterruptStatus(status);
46  }

```

13~18 行, 先判断调用者是内核线程, 还是用户进程, 以便于后面释放虚拟页的时候指明虚拟地址池的类型。

23~25 行, 若要 free 是大内存, 则直接释放其 arena 对应的 cnt 个虚拟页即可。

若要 free 是小内存块, 先将此内存块回收到此内存块描述符的 free\_list 中。接下来在第 32 行将 “++a->cnt” 后的结果与内存块描述符中的 blocks\_per\_arena 比较, 如果相等, 这表示此 arena 中的空闲内存块已经达到最大数, 说明此 arena 已经没人使用了, 可以释放。于是接下来通过 for 循环, 将此 arena 中的所有内存块从内存块描述符的 free\_list 中去掉, 遍历结束后, 通过 releasePages 释放此 arena 对应的虚拟页。

至此, 内存回收工作完成。

将函数定义加入内存管理器类定义中, 并将其定义为系统调用。

(src/PROJECT2/include/syscall.h)

```

1 void syscall_free(const int ptr);

```

(src/PROJECT2/src/kernel/syscall.cpp)

```

1 void syscall_free(const int ptr){
2     return memoryManager.free(ptr);
3 }

```

在启动内核时, 在内核线程中对齐进行测试, 测试方法如下:

(src/PROJECT2/src/kernel/setup.cpp)

```

1 void second_thread(void *arg){
2     printf("start second thread.\n");

```

```

3
4     int addr[10];
5     int size = 16;
6     for(int i = 0; i < 10; i++){
7         addr[i] = syscall_malloc(size);
8         printf("malloc(%d), addr is %x\n", size, addr[i]);
9         size *= 2;
10    }
11
12    for(int i = 9; i >= 0; i--){
13        printf("free %x \n", addr[i]);
14        syscall_free(addr[i]);
15    }
16
17    asm_halt();
18 }
19
20 void first_thread(void *arg)
21 {
22     printf("start first thread.\n");
23     programManager.executeThread(second_thread, nullptr, "second thread"
24         , 1);
25
26     int addr[10];
27     for(int i = 0; i < 7; i++){
28         addr[i] = syscall_malloc(513);
29         printf("malloc(513), addr is %x\n", addr[i]);
30     }
31
32     for(int i = 0; i < 3; i++){
33         printf("free %x \n", addr[i]);
34         syscall_free(addr[i]);
35     }
36
37     for(int i = 7; i < 10; i++){
38         addr[i] = syscall_malloc(513);
39         printf("malloc(513), addr is %x\n", addr[i]);
40     }
41
42     for(int i = 0; i < 10; i++){
43         printf("addr[%d]: %x\n", i, addr[i]);
44     }
45
46     for(int i = 3; i < 10; i++){
47         syscall_free(addr[i]);

```

```

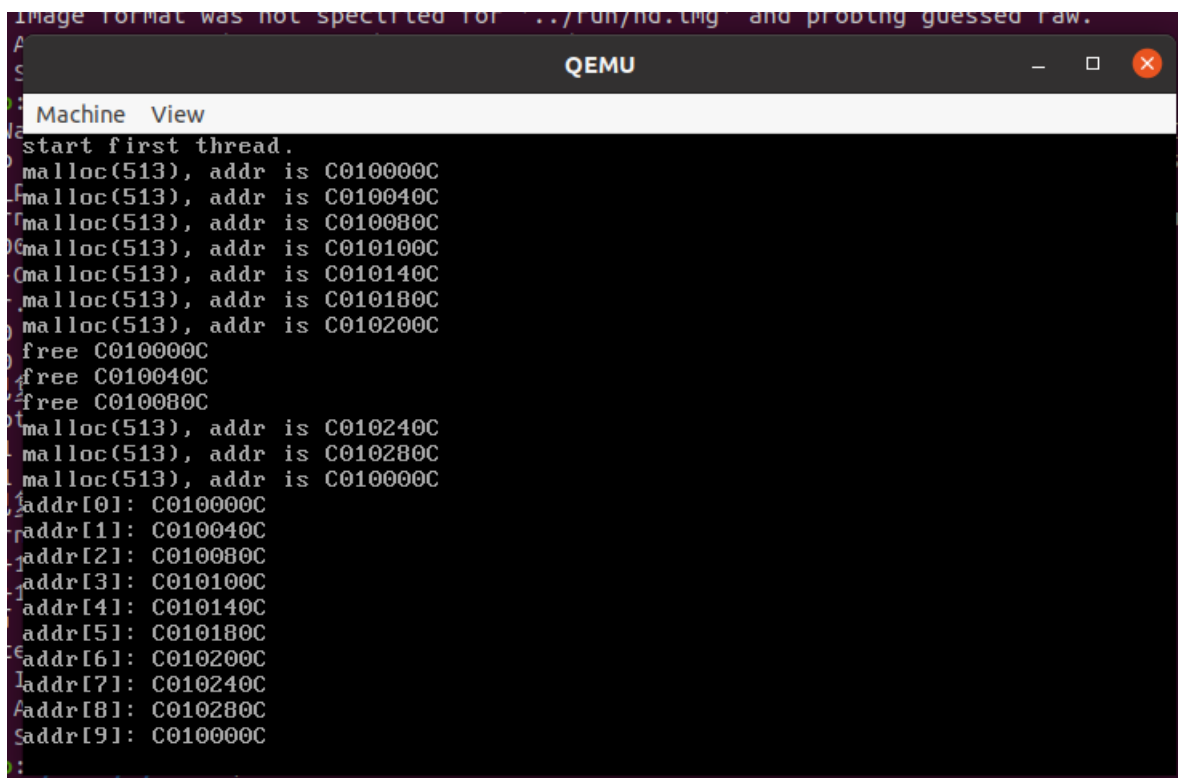
47     }
48
49     asm_halt();
50 }

```

第一个内核线程调用 7 次 malloc，每次申请 513 字节内存。随后将申请来的前 3 块内存释放掉，随后再调用 3 次 malloc，每次也申请 513 字节内存。最后再将所有申请来的内存块释放掉。

第二个内核线程调用 10 次 malloc，依次申请 16、32、...、8192 字节内存。随后再依次将 8192、4096、...、16 字节的内存块释放掉。

运行结果如图 2.8、2.9。



```

Image format was not specified for ../run/hd.img and probing guessed raw.
A
S
Machine View
start first thread.
malloc(513), addr is C010000C
malloc(513), addr is C010040C
malloc(513), addr is C010080C
malloc(513), addr is C010100C
malloc(513), addr is C010140C
malloc(513), addr is C010180C
malloc(513), addr is C010200C
free C010000C
free C010040C
free C010080C
malloc(513), addr is C010240C
malloc(513), addr is C010280C
malloc(513), addr is C010000C
addr[0]: C010000C
addr[1]: C010040C
addr[2]: C010080C
addr[3]: C010100C
addr[4]: C010140C
addr[5]: C010180C
addr[6]: C010200C
addr[7]: C010240C
addr[8]: C010280C
addr[9]: C010000C

```

图 2.8: 运行结果: first\_thread

第一个线程，申请 513 字节大小的内存块时，会为其分配 1024 字节规格的内存块，按照前面定义，这种规格的内存块的 blocks\_per\_arena 为  $\lfloor (4096 - 12) / 1024 \rfloor = 3$ ，因此在前 7 次 malloc 时将会创建 3 个内存块规格为 1024 字节的 arena，释放掉前 3 次申请的内存块后，再进行 3 次申请，由于我在此实现的 free\_list 是 FIFO 的，因此前两次申请来的是第 3 个 arena 空闲的两个内存块 0xC010240C、0xC010280C，随后申请来的是释放过后空闲出来的 0xC010000C。

第二个线程，由于所有内核线程共享一个堆，第一个线程申请后释放的内存都已全部归还，因此第二个线程申请的内存块位置从第一个虚拟页内开始。一直申请到 2048 字节，返回地址都以一个页框大小递增，申请到 4096 字节时，由于  $4096 > 4096 - 12$  故

```
Machine View
addr[7]: C010240C
addr[8]: C010280C
addr[9]: C010000C
start second thread.
malloc(16), addr is C010000C
malloc(32), addr is C010100C
malloc(64), addr is C010200C
malloc(128), addr is C010300C
malloc(256), addr is C010400C
malloc(512), addr is C010500C
malloc(1024), addr is C010600C
malloc(2048), addr is C010700C
malloc(4096), addr is C010800C
malloc(8192), addr is C010A00C
free C010A00C
free C010800C
free C010700C
free C010600C
free C010500C
free C010400C
free C010300C
free C010200C
free C010100C
free C010000C
```

图 2.9: 运行结果: second\_thread

需要 2 个页框, 故申请 8192 字节时返回地址从  $0xC010800C + 0x2000 = 0xC010A00C$  开始。

## 系统调用 malloc 和 free

将上述实现的 malloc 和 free 封装成系统调用以供用户进程调用。

在 PCB 中加入用户内存块描述符数组。

(src/PROJECT2/include/thread.h)

```
1 struct PCB
2 {
3     ...
4     mem_block_desc user_block_descs[DESC_CNT]; // 用户内存块描述符数组
5 };
```

在用户进程初始化时初始化进程的内存块描述符。

(src/PROJECT2/src/kernel/program.cpp)

```
1 int ProgramManager::executeProcess(const char *filename, int priority)
2 {
3     ...
4     // 初始化进程的内存块描述符
5     memoryManager.block_desc_init(process->user_block_descs);
6 }
```



随后添加系统调用。

(src/PROJECT2/include/syscall.h)

```
1 // 第6个系统调用, malloc
2 int malloc(const int size);
3 int syscall_malloc(const int size);
4
5 // 第7个系统调用, free
6 void free(const int ptr);
7 void syscall_free(const int ptr);
```

(src/PROJECT2/src/kernel/syscall.cpp)

```
1 int malloc(const int size){
2     asm_system_call(6, size);
3 }
4 int syscall_malloc(const int size){
5     return memoryManager.malloc(size);
6 }
7
8 void free(const int ptr){
9     asm_system_call(7, ptr);
10 }
11 void syscall_free(const int ptr){
12     return memoryManager.free(ptr);
13 }
```

(src/PROJECT2/src/kernel/setup.cpp)

```
1 ...
2 // 设置6号系统调用
3 systemService.setSystemCall(6, (int)syscall_malloc);
4 // 设置7号系统调用
5 systemService.setSystemCall(7, (int)syscall_free);
6 ...
```

至此准备工作完成, 下面在用户进程中进行测试。

(src/PROJECT2/src/kernel/setup.cpp)

```
1 ...
2 void first_process()
3 {
4     printf("start first process.\n");
5
6     int addr[10];
7     for(int i = 0; i < 7; i++){
8         addr[i] = malloc(513);
9         printf("malloc(513), addr is %x\n", addr[i]);
10    }
```

```

11
12     for(int i = 0; i < 3; i++){
13         printf("free %x \n", addr[i]);
14         free(addr[i]);
15     }
16
17     for(int i = 7; i <10; i++){
18         addr[i] = malloc(513);
19         printf("malloc(513), addr is %x\n", addr[i]);
20     }
21
22     for(int i = 0; i < 10; i++){
23         printf("addr[%d]: %x\n", i, addr[i]);
24     }
25
26     for(int i = 3; i < 10; i++){
27         free(addr[i]);
28     }
29     asm_halt();
30
31 }
32 void second_process()
33 {
34     printf("start second process.\n");
35
36     int addr[10];
37     int size = 16;
38     for(int i = 0; i < 10; i++){
39         addr[i] = malloc(size);
40         printf("malloc(%d), addr is %x\n", size, addr[i]);
41         size *= 2;
42     }
43
44     for(int i = 9; i >= 0; i--){
45         printf("free %x \n", addr[i]);
46         free(addr[i]);
47     }
48     asm_halt();
49
50 }
51 ...
52 void first_thread(void *arg)
53 {
54     printf("start first thread.\n");
55     programManager.executeThread(second_thread, nullptr, "second thread"

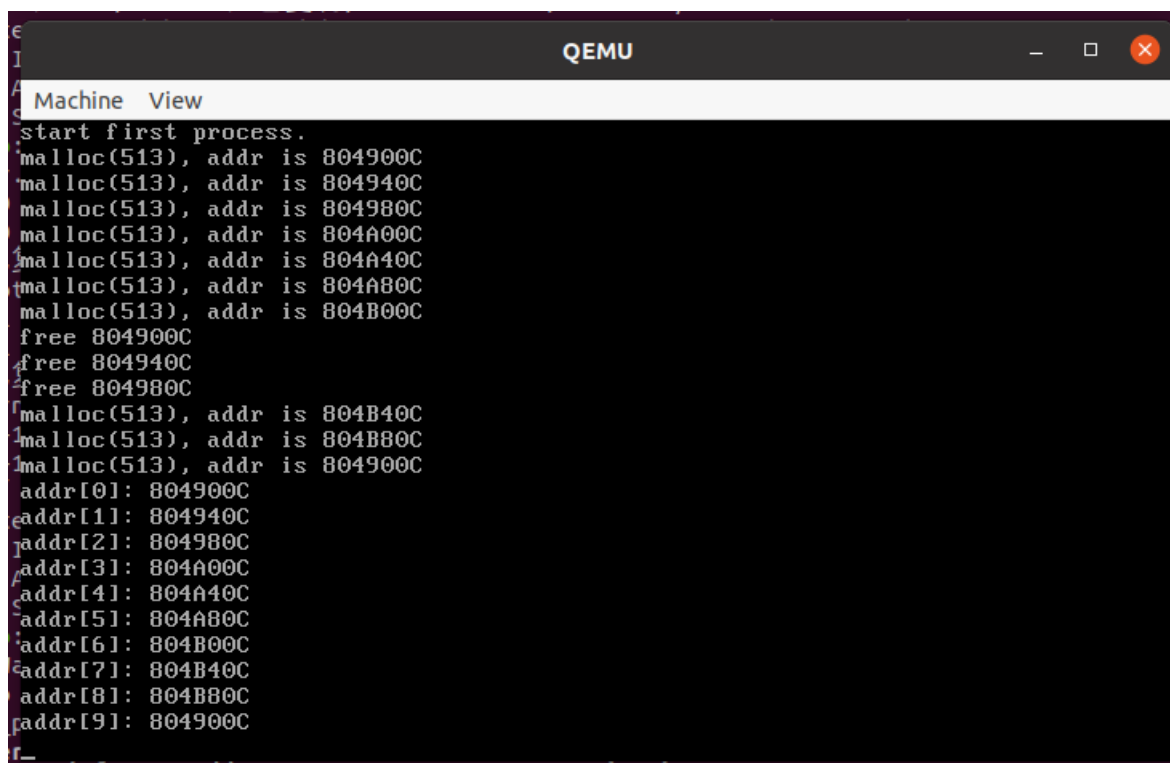
```

```

        , 1);
56     programManager.executeProcess((const char *)first_process, 1);
57     programManager.executeProcess((const char *)second_process, 1);
58     ...
59 }
60 ...

```

此处的测试方法和前面测试内核态时完全相同，运行结果如图 2.10、2.11。



```

Machine View
start first process.
malloc(513), addr is 804900C
malloc(513), addr is 804940C
malloc(513), addr is 804980C
malloc(513), addr is 804A00C
malloc(513), addr is 804A40C
malloc(513), addr is 804A80C
malloc(513), addr is 804B00C
free 804900C
free 804940C
free 804980C
malloc(513), addr is 804B40C
malloc(513), addr is 804B80C
malloc(513), addr is 804900C
addr[0]: 804900C
addr[1]: 804940C
addr[2]: 804980C
addr[3]: 804A00C
addr[4]: 804A40C
addr[5]: 804A80C
addr[6]: 804B00C
addr[7]: 804B40C
addr[8]: 804B80C
addr[9]: 804900C

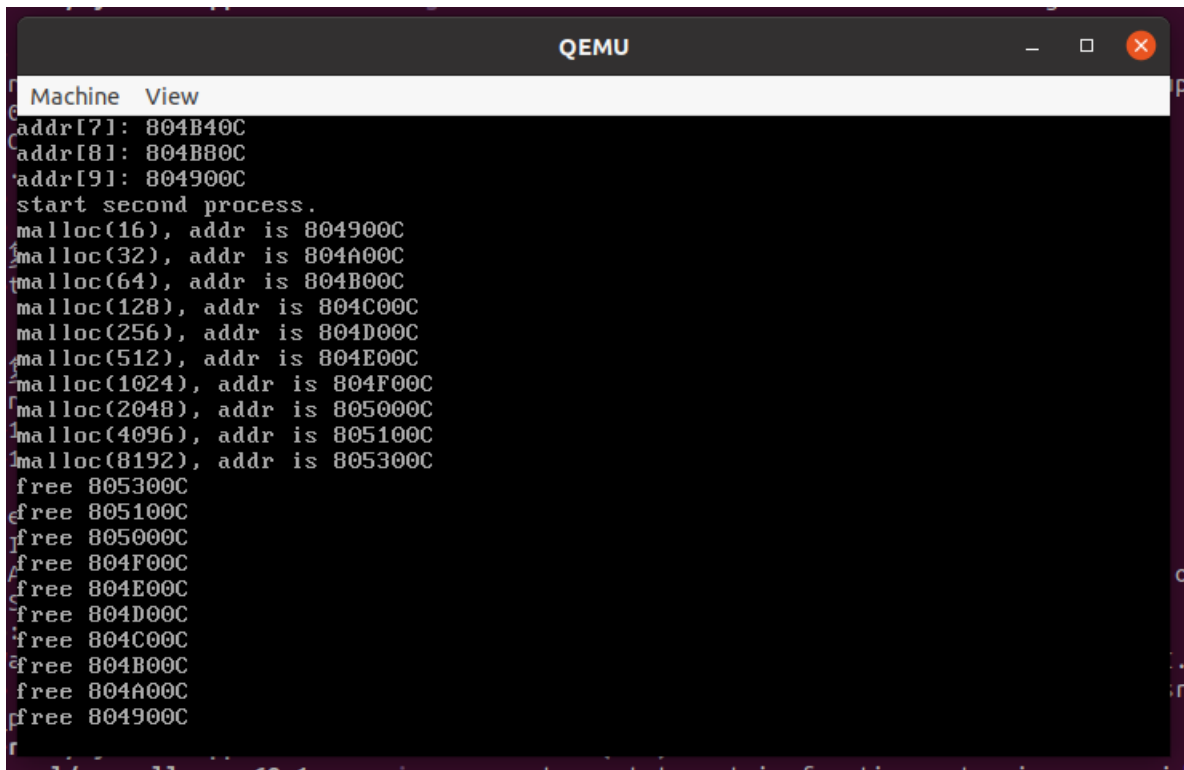
```

图 2.10: 运行结果：first\_process

第一个进程，申请 513 字节大小的内存块时，会为其分配 1024 字节规格的内存块，按照前面定义，这种规格的内存块的 `blocks_per_arena` 为  $\lfloor (4096 - 12) / 1024 \rfloor = 3$ ，因此在前 7 次 `malloc` 时将会创建 3 个内存块规格为 1024 字节的 arena，释放掉前 3 次申请的内存块后，再进行 3 次申请，由于我在此实现的 `free_list` 是 FIFO 的，因此前两次申请来的是第 3 个 arena 空闲的两个内存块 0x804B40C、0x804B80C，随后申请来的是释放过后空闲出来的 0X804900C。

第二个进程，由于用户进程之间内存不共享，第二个进程申请的内存块位置从第一个虚拟页内开始。一直申请到 2048 字节，返回地址都以一个页框大小递增，申请到 4096 字节时，由于  $4096 > 4096 - 12$  故需要 2 个页框，故申请 8192 字节时返回地址从  $0x805100C + 0x2000 = 0x805300C$  开始。

实验结果符合预期。



```
Machine View
addr[7]: 804B40C
addr[8]: 804B80C
addr[9]: 804900C
start second process.
malloc(16), addr is 804900C
malloc(32), addr is 804A00C
malloc(64), addr is 804B00C
malloc(128), addr is 804C00C
malloc(256), addr is 804D00C
malloc(512), addr is 804E00C
malloc(1024), addr is 804F00C
malloc(2048), addr is 805000C
malloc(4096), addr is 805100C
malloc(8192), addr is 805300C
free 805300C
free 805100C
free 805000C
free 804F00C
free 804E00C
free 804D00C
free 804C00C
free 804B00C
free 804A00C
free 804900C
```

图 2.11: 运行结果: second\_process

## 同步和互斥的实现

### Race Condition 的分析

当多个进程/线程对同一个共享资源进行修改时，会出现竞争。在前面的实验中实现了自旋锁和信号量，通过使用自旋锁或信号量来实现临界区能有效地避免竞争的出现。

在本节实现的 malloc 和 free 中，涉及进程/线程竞争的部分在于内存页的分配，所有的内核线程共享一个内核物理地址池和一个内核虚拟地址池，所有的进程共享一个用户物理地址池，当多个内核线程申请物理页/虚拟页，或多个用户进程申请物理页时，就有可能出现竞争。

回忆一下内存分配的本质：将内存页在对应的地址池的位图中的对应位置置位。同时有多个请求对同一个地址池申请内存页时有可能会出现竞争，下面来分析竞争可能出现的原因。在位图中申请资源时，所采取方法如下：

(src/PROJECT2/src/utils/bitmap.cpp)

```
1 int BitMap::allocate(const int count)
2 {
3     if (count == 0)
4         return -1;
5
6     int index, empty, start;
```

```

7     index = 0;
8     while (index < length)
9     {
10        // 越过已经分配的资源
11        while (index < length && get(index))
12            ++index;
13
14        // 不存在连续的count个资源
15        if (index == length)
16            return -1;
17
18        // 找到1个未分配的资源
19        // 检查是否存在从index开始的连续count个资源
20        empty = 0;
21        start = index;
22        while ((index < length) && (!get(index)) && (empty < count))
23        {
24            ++empty;
25            ++index;
26        }
27
28        // 存在连续的count个资源
29        if (empty == count)
30        {
31            for (int i = 0; i < count; ++i){
32                set(start + i, true);
33            }
34            return start;
35        }
36    }
37    return -1;
38 }

```

请求者会先找到满足要求的资源所在的位置，然后再对资源进行申请，即通过32~35行for循环置位，最后返回资源起始位置。申请资源的过程中不会再对资源的状态进行检查，若有两个请求者同时搜索可用资源，它们可能同时找到了同样位置的资源，然后申请资源返回，这就导致了一块内存资源被两个进程/线程获得，出现竞争，会导致后面的程序出错。

下面来模拟这种情况：两个内核线程同时请求分配一页虚拟页，打印申请到的虚拟页的起始位置。

(src/PROJECT2/src/kernel/setup.cpp)

```

1  ...
2  void second_thread(void *arg){
3      printf("Start second thread.\n");

```

```

4     int add2 = memoryManager.allocateVirtualPages(AddressPoolType::
        KERNEL, 4);
5     printf("Second thread allocate start %x\n", add2);
6
7     asm_halt();
8 }
9
10 void first_thread(void *arg)
11 {
12     printf("Start first thread.\n");
13     programManager.executeThread(second_thread, nullptr, "second thread"
        , 1);
14     int add1 = memoryManager.allocateVirtualPages(AddressPoolType::
        KERNEL, 4);
15     printf("First thread allocate start %x\n", add1);
16     asm_halt();
17 }
18 ...

```

假设进程/线程找到可用资源的位置后迟迟不申请，如下：

(src/PROJECT2/src/utils/bitmap.cpp)

```

1 ...
2     // 存在连续的count个资源
3     if (empty == count)
4     {
5         PCB *run = programManager.running;
6         if(run->pid==0)
7             printf("First thread find %d pages started from %x\n", count
                , start);
8         else
9             printf("Second thread find %d pages started from %x\n",
                count, start);
10        uint32 f =0xffffffff;
11        while(f){
12            f--;
13        }
14
15        if(run->pid==0)
16            printf("First thread set.\n");
17        else
18            printf("Second thread set.\n");
19
20        for (int i = 0; i < count; ++i)
21        {
22            set(start + i, true);

```

```

23     }
24     if(run->pid==0)
25         printf("First thread finish.\n");
26     else
27         printf("Second thread finish.\n");
28     return start;
29 }
30 ...

```

运行结果如图 2.12 所示。

```

Machine View
Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
Start first thread.
First thread find 4 pages started from 0
Start second thread.
Second thread find 4 pages started from 0
First thread set.
First thread finish.
First thread allocate start C0100000
Second thread set.
Second thread finish.
Second thread allocate start C0100000

```

图 2.12: 内存分配竞争

第一个线程和第二个线程都找到了内核虚拟地址池起始位置为 0 的 4 个空闲页，随后第一个线程申请资源，将位图中的位置置位，完成后返回。第二个线程认为这 4 个页还是空闲的，于是也对其进行申请，将位图中的位置置位，完成后返回。最后回到线程中时两个线程申请的内存页是相同的，起始位置均为 0xC0100000，在使用的时候会导致程序出错。

下面将通过锁机制来避免这种情况出现，解决了这个问题其实也是解决了 malloc/free 的同步和互斥的问题。

## 解决方法

使用自旋锁来实现互斥。

首先，在地址池类中加入自旋锁成员。

(src/PROJECT2/include/address\_pool.h)

```
1 class AddressPool
2 {
3 public:
4     BitMap resources;
5     int startAddress;
6     SpinLock lock;
7 public:
8     ...
9 };
```

然后在地址池的申请函数中实现锁机制。

(src/PROJECT2/src/utls/address\_pool.cpp)

```
1 // 从地址池中分配count个连续页
2 int AddressPool::allocate(const int count)
3 {
4     lock.lock();
5     int start = resources.allocate(count);
6     lock.unlock();
7     return (start == -1) ? -1 : (start * PAGE_SIZE + startAddress);
8 }
```

下面进行测试，运行会出现竞争的代码（Race Condition 的分析中的模拟竞争的代码），如图 2.13 所示。

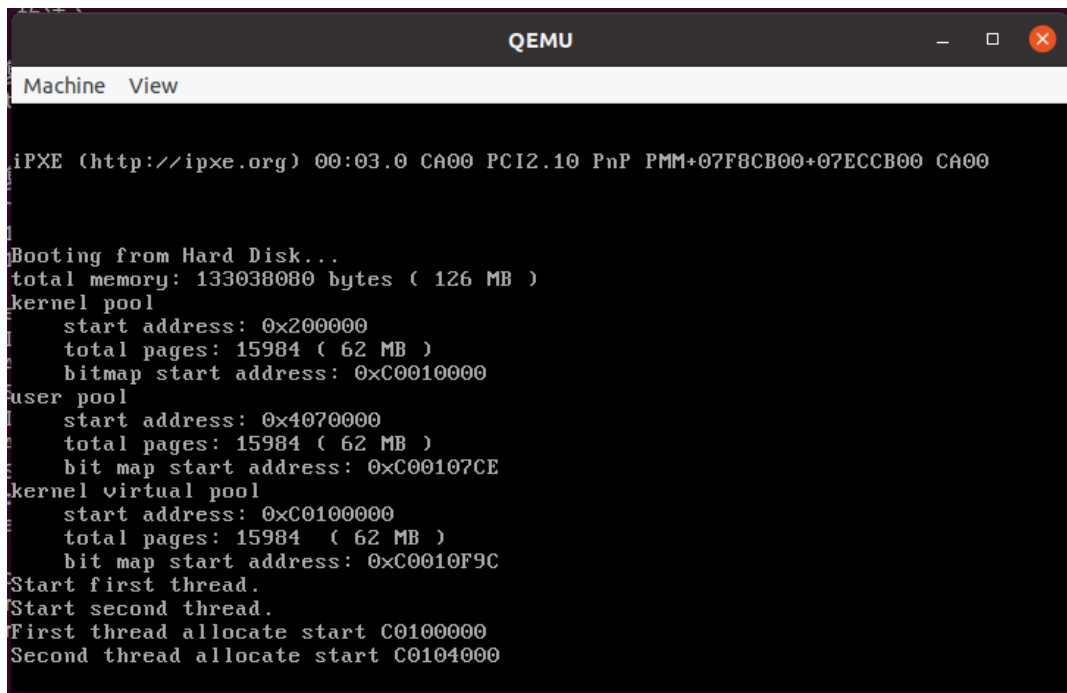


图 2.13: 实现同步和互斥



看到第二个线程再去申请时的起始地址为 0xC0100400，不再和第一个线程发生竞争冲突，符合预期。

从而实现了地址池的同步互斥，之后进程/线程在申请内存时不会发生竞争，也就解决了 malloc/free 的同步和互斥问题。

## 总结

在本次实验中，我先根据参考资料实现了 Project2，完成之后尝试去实现 Project1，但并没有完整地实现页面置换算法。

这次实验参考了《操作系统真象还原》和《ucore\_os\_labs》等教程，作者的解释已足够详尽，因此在解释实现过程免不了大量引用原文，但也只能尽可能的多加入自己的理解和总结。

最后总结一下自己的实验课。这个学期也算做了很多次的实验，做实验的过程中遇到过大大小小的问题，遭遇过一些“瓶颈”，但做完后总是能收获很多。之前一直不太清晰进程、线程的概念，尽管课上把定义熟记于心，但只有真正的做了之后才真正地理解了。看实验指导书能体会到老师和各位师兄师姐的认真负责和良苦用心，为此水平不够的自己心里常怀愧疚，详细亲切的实验指导也总能激励着自己多用功一些。总的来说不太满意，最主要的原因是自己的能力水平有限吧。此外，在自己有限的的能力范围内还要兼顾其他课程，最后每一科都囫圇吞枣，这也是自己的一个不太满意的原因。

实验课结束了，但又没有结束。最后一次的实验也只是完成了一些基本要求，但由于能力水平、时间和精力有限，没能进一步拓展。这个学期所学的知识会牢牢地记在我的心里，在以后的学习和职业生涯中一直受用，谢谢老师和各位师兄师姐的悉心指导！祝好！