



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 内存管理

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

二〇二一年六月六日

目录

1	Assignment 1	1
1.1	复现代码	1
1.2	内存管理	1
2	Assignment 2	4
2.1	first-fit	5
2.2	next-fit	7
2.3	best-fit	9
3	Assignment 3	11
3.1	FIFO	11
3.2	LRU	11
4	Assignment 4	13
4.1	Part 1	13
4.1.1	申请	13
4.1.2	释放	17
4.2	Part 2	18
5	总结	21

Assignment 1

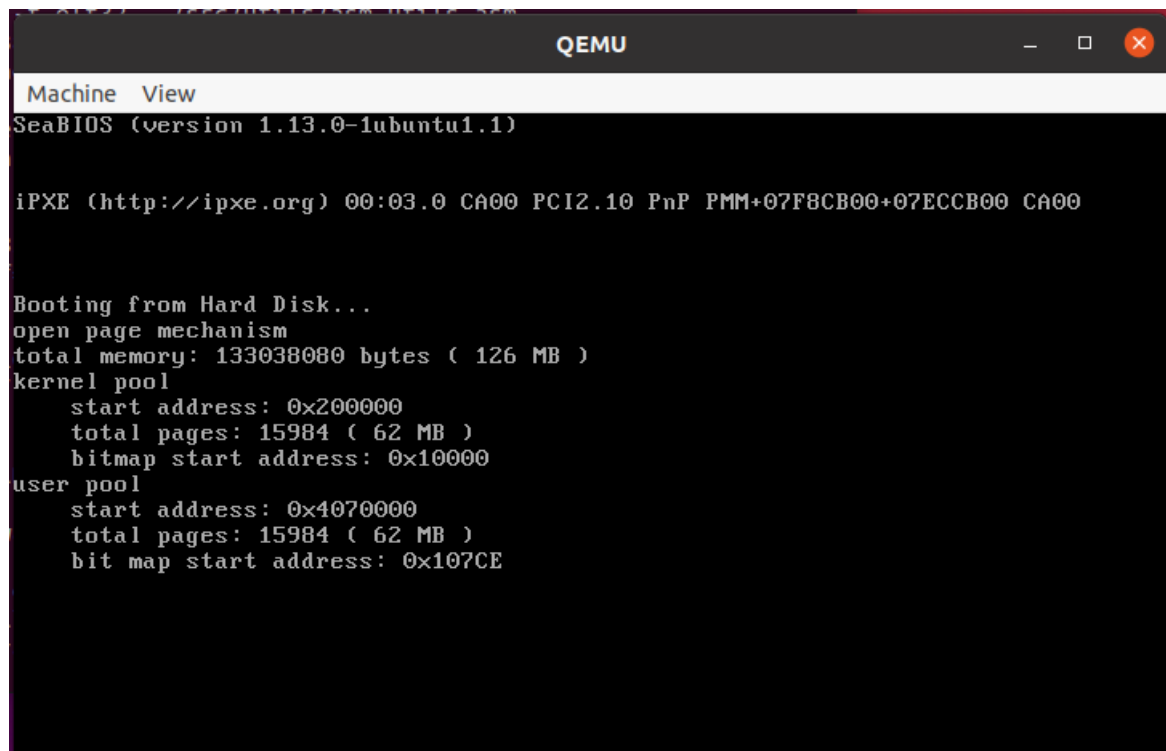
复现参考代码，实现二级分页机制，并能够在虚拟机地址空间中进行内存管理，包括内存的申请和释放等，截图并给出过程解释。

复现代码

在工程中建立相应的文件和代码，执行

```
1 make && make run
```

完成初始化后，屏幕的输出如下：



内核物理地址池和用户物理地址池创建成功，成功开启了分页机制。

内存管理

在虚拟机地址空间中进行内存管理：

```
1 void first_thread(void *arg)
2 {
3     // 第1个线程不可以返回
4     // stdio.moveCursor(0);
5     // for (int i = 0; i < 25 * 80; ++i)
6     // {
7     //     stdio.print(' ');
8     // }
9     // stdio.moveCursor(0);
10    printf("Allocate 4 pages.\n");
```

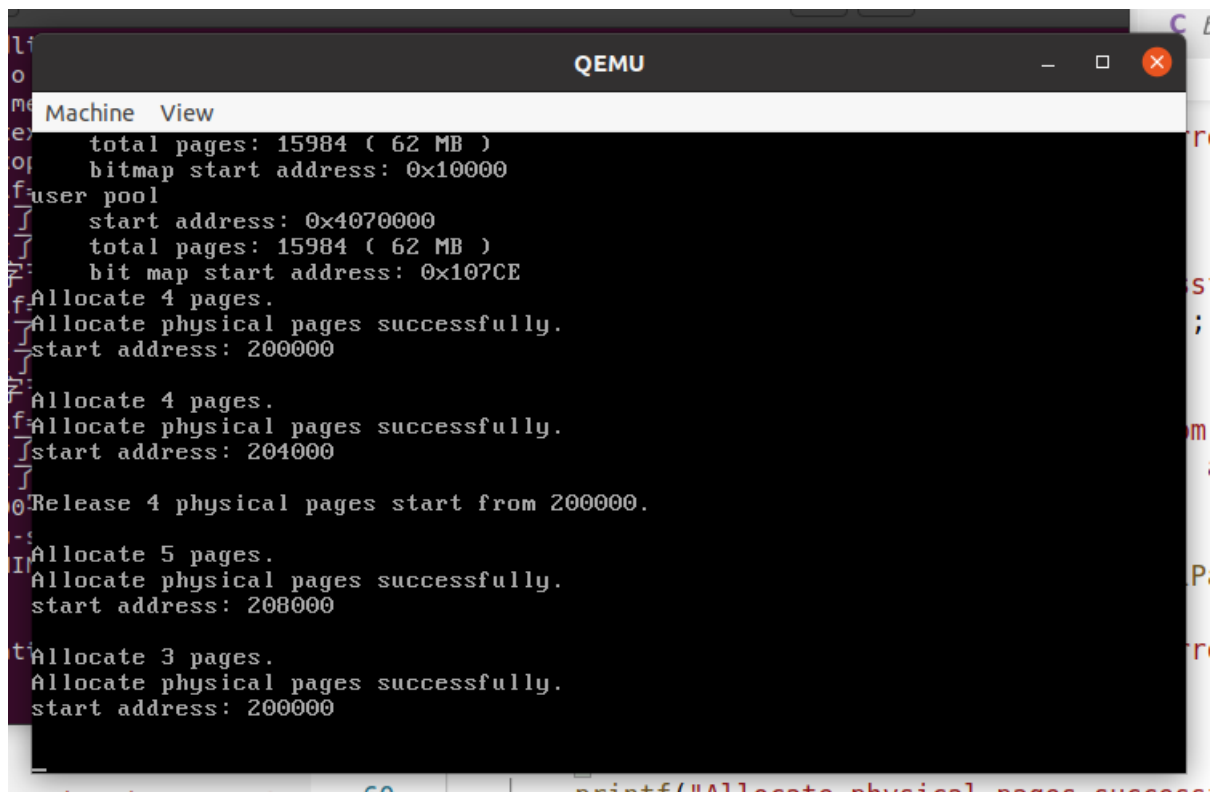
```

11     int addr1 = memoryManager.allocatePhysicalPages(KERNEL, 4);
12     if(!addr1){
13         printf("KERNEL PhysicalPages allocate error!\n");
14         asm_halt();
15     }
16     else{
17         printf("Allocate physical pages successfully.\n");
18         printf("start address: %x\n\n", addr1);
19     }
20     printf("Allocate 4 pages.\n");
21     int addr2 = memoryManager.allocatePhysicalPages(KERNEL, 4);
22     if(!addr2){
23         printf("KERNEL PhysicalPages allocate error!\n");
24         asm_halt();
25     }
26     else{
27         printf("Allocate physical pages successfully.\n");
28         printf("start address: %x\n\n", addr2);
29     }
30     printf("Release 4 physical pages start from %x.\n\n", addr1);
31     memoryManager.releasePhysicalPages(KERNEL, addr1, 4);
32     printf("Allocate 5 pages.\n");
33     int addr3 = memoryManager.allocatePhysicalPages(KERNEL, 5);
34     if(!addr3){
35         printf("KERNEL PhysicalPages allocate error!\n");
36         asm_halt();
37     }
38     else{
39         printf("Allocate physical pages successfully.\n");
40         printf("start address: %x\n\n", addr3);
41     }
42     printf("Allocate 3 pages.\n");
43     int addr4 = memoryManager.allocatePhysicalPages(KERNEL, 3);
44     if(!addr4){
45         printf("KERNEL PhysicalPages allocate error!\n");
46         asm_halt();
47     }
48     else{
49         printf("Allocate physical pages successfully.\n");
50         printf("start address: %x\n\n", addr4);
51     }
52     asm_halt();
53 }

```

在第一个线程中，先向内核物理地址池申请物理页，分两次，每次申请 4 个物理页，随后将申请来的第一个物理页释放，在依次申请 5 个物理页和 3 个物理页。

运行结果如下：

A screenshot of a QEMU Machine View window. The window has a title bar with the text 'QEMU' and standard window controls. The main area is a black terminal with white text showing memory allocation logs. The logs include information about total pages (15984, 62 MB), bitmap start addresses (0x10000 and 0x107CE), and several allocation and deallocation operations for physical pages. The operations are: 1. Allocate 4 pages, start address: 200000. 2. Allocate 4 pages, start address: 204000. 3. Release 4 physical pages start from 200000. 4. Allocate 5 pages, start address: 208000. 5. Allocate 3 pages, start address: 200000. The logs are partially obscured by a vertical scrollbar on the left and a horizontal scrollbar at the bottom.

```
Machine View
total pages: 15984 ( 62 MB )
bitmap start address: 0x10000
user pool
start address: 0x4070000
total pages: 15984 ( 62 MB )
bit map start address: 0x107CE
Allocate 4 pages.
Allocate physical pages successfully.
start address: 200000
Allocate 4 pages.
Allocate physical pages successfully.
start address: 204000
Release 4 physical pages start from 200000.
Allocate 5 pages.
Allocate physical pages successfully.
start address: 208000
Allocate 3 pages.
Allocate physical pages successfully.
start address: 200000
```

第一次申请 4 个物理页，起始地址为 0x20000，即内核物理地址池的起始地址，第二次申请 4 个物理页，起始地址为 0x24000，由于每个页大小为 4KB，第一次申请的地址空间为 0x20000-0x23fff，第二次申请的起始地址正是从第一次申请的末尾。

释放掉第一次申请的地址空间后，第三次申请 5 个物理页，由于申请的地址是连续，第一次申请释放后的 4 个物理页的空间显然无法满足需求，因此第三次申请的空间只能从第二次申请的末尾开始，即 0x28000。第四次申请 3 个物理页，从内核物理地址池的头部开始寻找空间，第一次申请所释放的 4 个物理页的空间大小满足需求，因此第四次申请的起始地址为 0x20000。

Assignment 2

参照理论课上的学习的物理内存分配算法如 first-fit, best-fit 等实现动态分区算法等，或者自行提出自己的算法。

编写测试代码如下：

```
1 void first_thread(void *arg)
2 {
3     // 第1个线程不可以返回
4     // stdio.moveCursor(0);
5     // for (int i = 0; i < 25 * 80; ++i)
6     // {
7     //     stdio.print(' ');
8     // }
9     // stdio.moveCursor(0);
10    printf("Allocate 4 pages.\n");
11    int addr1 = memoryManager.allocatePhysicalPages(KERNEL, 4);
12    if(!addr1){
13        printf("KERNEL PhysicalPages allocate error!\n");
14        asm_halt();
15    }
16    else{
17        printf("Allocate physical pages successfully.\n");
18        printf("start address: %x\n\n", addr1);
19    }
20    printf("Allocate 1 pages.\n");
21    int addr2 = memoryManager.allocatePhysicalPages(KERNEL, 1);
22    if(!addr2){
23        printf("KERNEL PhysicalPages allocate error!\n");
24        asm_halt();
25    }
26    else{
27        printf("Allocate physical pages successfully.\n");
28        printf("start address: %x\n\n", addr2);
29    }
30    printf("Allocate 5 pages.\n");
31    int addr3 = memoryManager.allocatePhysicalPages(KERNEL, 5);
32    if(!addr3){
33        printf("KERNEL PhysicalPages allocate error!\n");
34        asm_halt();
35    }
36    else{
37        printf("Allocate physical pages successfully.\n");
38        printf("start address: %x\n\n", addr3);
39    }
40    printf("Allocate 1 pages.\n");
41    int addr4 = memoryManager.allocatePhysicalPages(KERNEL, 1);
42    if(!addr4){
43        printf("KERNEL PhysicalPages allocate error!\n");
44        asm_halt();
45    }
46    else{
47        printf("Allocate physical pages successfully.\n");
48        printf("start address: %x\n\n", addr4);
49    }
}
```

```

50     printf("Allocate 3 pages.\n");
51     int addr5 = memoryManager.allocatePhysicalPages(KERNEL, 3);
52     if(!addr5){
53         printf("KERNEL PhysicalPages allocate error!\n");
54         asm_halt();
55     }
56     else{
57         printf("Allocate physical pages successfully.\n");
58         printf("start address: %x\n\n", addr5);
59     }
60     printf("Allocate 1 pages.\n");
61     int addr6 = memoryManager.allocatePhysicalPages(KERNEL, 1);
62     if(!addr6){
63         printf("KERNEL PhysicalPages allocate error!\n");
64         asm_halt();
65     }
66     else{
67         printf("Allocate physical pages successfully.\n");
68         printf("start address: %x\n\n", addr6);
69     }
70     printf("Release 4 physical pages start from %x.\n\n", addr1);
71     memoryManager.releasePhysicalPages(KERNEL, addr1, 4);
72     printf("Release 5 physical pages start from %x.\n\n", addr3);
73     memoryManager.releasePhysicalPages(KERNEL, addr3, 5);
74     printf("Release 3 physical pages start from %x.\n\n", addr5);
75     memoryManager.releasePhysicalPages(KERNEL, addr5, 3);
76     int addr7 = memoryManager.allocatePhysicalPages(KERNEL, 3);
77     if(!addr7){
78         printf("KERNEL PhysicalPages allocate error!\n");
79         asm_halt();
80     }
81     else{
82         printf("Allocate physical pages successfully.\n");
83         printf("start address: %x\n\n", addr7);
84     }
85     asm_halt();
86 }

```

它先依次向内核物理地址池申请物理页（共进行 6 次申请），页数分别为 4、1、5、1、3、1，随后释放掉第 1、3、5 次申请的物理页，最后内核物理地址池的存储空间分布如下：



最后再向内核物理地址池进行第 7 次申请，申请 3 个物理页，不同的动态分区方法会产生不同的结果。

first-fit

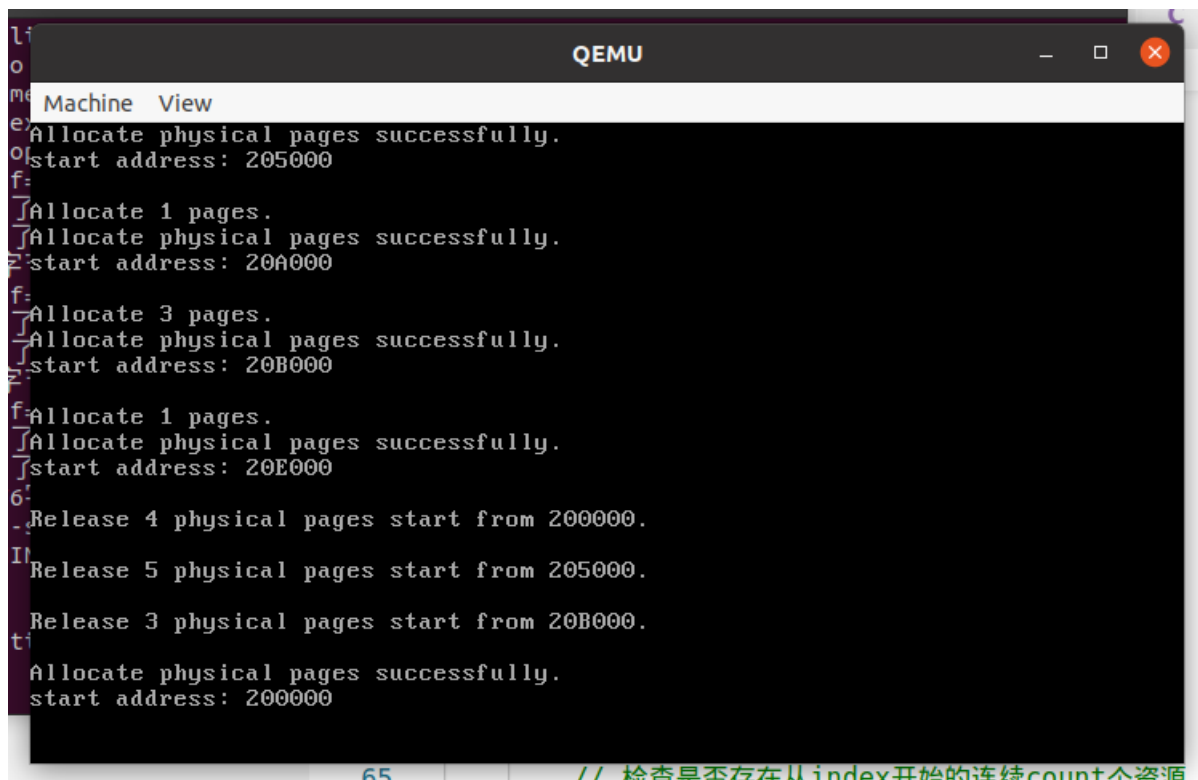
教程中采取的分区方法为 first-fit，它在找到第一个满足需求大小的空闲块就将其占用并返回该空闲块的地址，代码如下：

```

1  int BitMap::allocate(const int count)
2  {
3      if (count == 0)
4          return -1;
5      int index, empty, start;
6      index = 0;
7      while (index < length)
8      {
9          // 越过已经分配的资源
10         while (index < length && get(index))
11             ++index;
12         // 不存在连续的count个资源
13         if (index == length)
14             return -1;
15         // 找到1个未分配的资源
16         // 检查是否存在从index开始的连续count个资源
17         empty = 0;
18         start = index;
19         while ((index < length) && (!get(index)) && (empty < count))
20         {
21             ++empty;
22             ++index;
23         }
24         // 存在连续的count个资源
25         if (empty == count)
26         {
27             for (int i = 0; i < count; ++i)
28             {
29                 set(start + i, true);
30             }
31             return start;
32         }
33     }
34     return -1;
35 }

```

结果如下所示：



```
Machine View
Allocate physical pages successfully.
start address: 205000
Allocate 1 pages.
Allocate physical pages successfully.
start address: 20A000
Allocate 3 pages.
Allocate physical pages successfully.
start address: 20B000
Allocate 1 pages.
Allocate physical pages successfully.
start address: 20E000
Release 4 physical pages start from 200000.
Release 5 physical pages start from 205000.
Release 3 physical pages start from 20B000.
Allocate physical pages successfully.
start address: 200000
```

它将会占用找到的第一个空闲块并返回其起始地址，故返回了 0X200000。

next-fit

next-fit 在找到下一个满足需求大小的空闲块，将其占用并返回该空闲块的地址，代码如下：

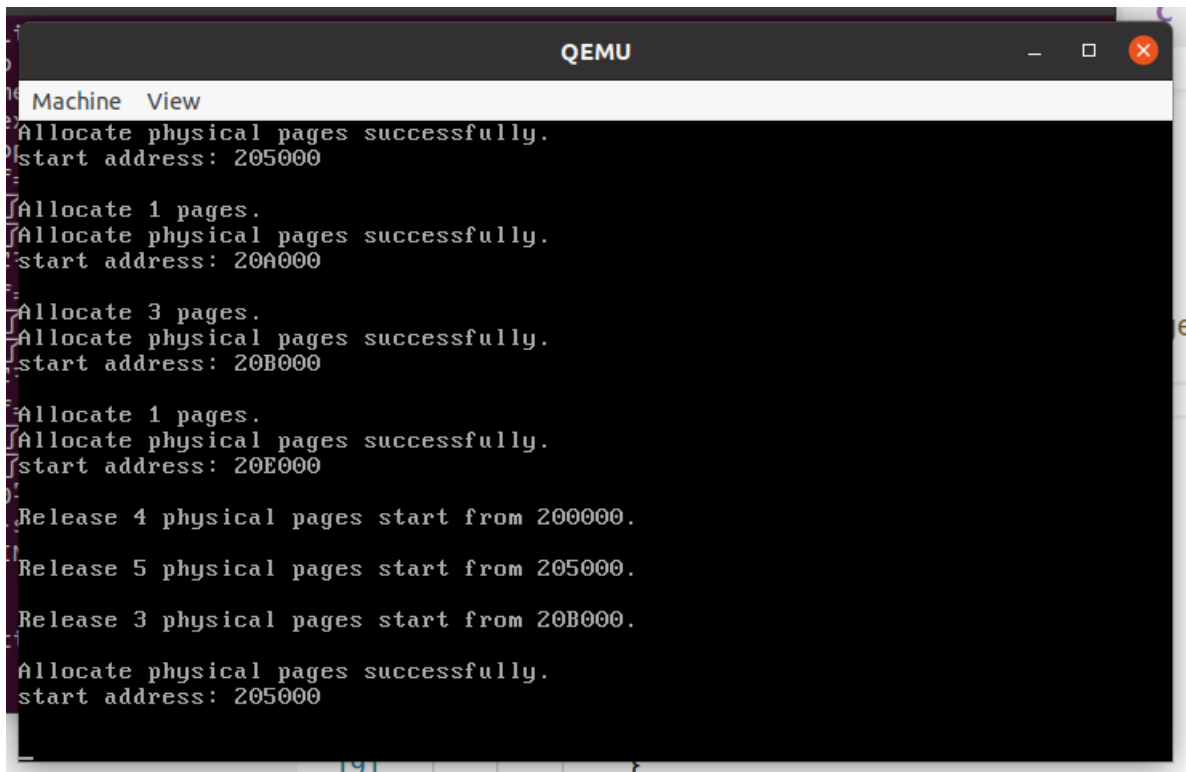
```
1 int BitMap::allocate(const int count)
2 {
3     if (count == 0)
4         return -1;
5     int index, empty, start;
6     int first_start = -1;
7     bool first = true;
8     index = 0;
9     while (index < length)
10    {
11        // 越过已经分配的资源
12        while (index < length && get(index))
13            ++index;
14        // 不存在连续的 count 个资源
15        if (index == length && first_start < 0)
16            return -1;
17        // 找到 1 个未分配的资源
18        // 检查是否存在从 index 开始的连续 count 个资源
19        empty = 0;
20        start = index;
21        while ((index < length) && (!get(index)) && (empty < count))
22        {
23            ++empty;
```

```

24         ++index;
25     }
26     // 存在连续的count个资源
27     if (empty == count)
28     {
29         // 第一个可用分区
30         if(first){
31             first = false;
32             first_start = start;
33             while ((index < length) && (!get(index)))
34             {
35                 index++;
36             }
37             continue;
38         }
39         // 下一个可用分区
40         for (int i = 0; i < count; ++i)
41         {
42             set(start + i, true);
43         }
44         return start;
45     }
46 }
47 // 若找不到下一个可用分区则返回第一个可用分区
48 if(first_start >= 0){
49     for (int i = 0; i < count; ++i){
50         set(first_start + i, true);
51     }
52     return first_start;
53 }
54 return -1;
55 }

```

结果如下所示：



```
Machine View
Allocate physical pages successfully.
start address: 205000

Allocate 1 pages.
Allocate physical pages successfully.
start address: 20A000

Allocate 3 pages.
Allocate physical pages successfully.
start address: 20B000

Allocate 1 pages.
Allocate physical pages successfully.
start address: 20E000

Release 4 physical pages start from 200000.

Release 5 physical pages start from 205000.

Release 3 physical pages start from 20B000.

Allocate physical pages successfully.
start address: 205000
```

下一个满足需求的空闲块起始地址为 0x205000。

best-fit

best-fit 将会选择所有满足需求的空闲块中大小最小的，以减少不必要的空间浪费。故需要遍历整个内核物理地址池，找出所有满足需求的空闲块，在这些空闲块中选择最小的，代码如下所示：

```
1 int BitMap::allocate(const int count)
2 {
3     if (count == 0)
4         return -1;
5     int index, empty, start;
6     int BF, size = length + 1;
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        // 不存在连续的 count 个资源
14        if (index == length)
15            return -1;
16        // 找到 1 个未分配的资源
17        // 检查是否存在从 index 开始的连续 count 个资源
18        empty = 0;
19        start = index;
20        while ((index < length) && (!get(index)) && (empty < count))
21        {
22            ++empty;
```

```

23         ++index;
24     }
25     // 存在连续的 count 个资源
26     if (empty == count)
27     {
28         while ((index < length) && (!get(index)))
29         {
30             ++empty;
31             ++index;
32         }
33         if(empty < size){
34             BF = start;
35             size = empty;
36         }
37     }
38 }
39 if(size <= length){
40     for (int i = 0; i < count; ++i){
41         set(BF + i, true);
42     }
43     return BF;
44 }
45 return -1;
46 }

```

结果如下所示：

```

QEMU
Machine View
Allocate physical pages successfully.
start address: 205000
Allocate 1 pages.
Allocate physical pages successfully.
start address: 20A000
Allocate 3 pages.
Allocate physical pages successfully.
start address: 20B000
Allocate 1 pages.
Allocate physical pages successfully.
start address: 20E000
Release 4 physical pages start from 200000.
Release 5 physical pages start from 205000.
Release 3 physical pages start from 20B000.
Allocate physical pages successfully.
start address: 20B000

```

best-fit 将会选择第 3 个空闲块，它的大小刚好为 3 个页，起始地址为 0x20b000

Assignment 3

参照理论课上虚拟内存管理的页面置换算法如 FIFO、LRU 等，实现页面置换，也可以提出自己的算法。

FIFO

代码如下所示：

```
1 int size_pages = 1024, num_pages = 0;
2 int head = 0, tail = 0;
3 int addr_pages[size_pages]; // 循环数组实现队列
4
5 // addr_page为所请求的页的地址，该页不在内存中
6 void FIFO(int addr_page){
7
8     if(num_pages < size_pages){
9         // 内存未满
10        num_pages++;
11
12        // 直接将请求的页加入队尾并换入
13        get(addr_page);
14        addr_pages[tail] = addr_page;
15        tail = (tail + 1) % size_pages;
16    }
17    else{
18        // 队列已满，需要进行置换
19
20        // 释放队列头部的页
21        release(addr_pages[head]);
22
23        // 换入请求的页
24        addr_pages[tail]=addr_page;
25        tail = (tail + 1) % size_pages;
26    }
27 }
```

LRU

代码如下所示：

```
1 int size_pages = 1024, num_pages = 0;
2 bool used[size_pages];
3 int unuse_times[size_pages]; // 记录了最近未被使用的次数
4 int addr_pages[size_pages]; // 记录地址
5 void LRU(int addr_page){
6     if(num_pages < size_pages){
7         // 内存未满
8         num_pages++;
9
10        // 找到空位，将请求的页换入
11        get(addr_page);
12        for(int i=0; i < size_pages; ++i){
```

```

13         if(!used[i]){
14             used[i] = true;
15             unuse_times[i] = 0;
16             addr_pages[i] = addr_page;
17             return ;
18         }
19     }
20 }
21 else{
22     // 内存已满
23     int target = 0;
24     int max_unuse_times = 0;
25     // 记录下最近未被使用次数最多的页
26     for (int i = 0; i < size_pages; ++i){
27         if(unuse_times[i] > max_unuse_times){
28             target = i;
29             max_unuse_times = unuse_times[i];
30         }
31     }
32     // 将其释放并换入请求的页
33     release(addr_pages[target]);
34     get(addr_page);
35     addr_pages[target] = addr_page;
36 }
37 }

```

Assignment 4

复现“虚拟页内存管理”一节的代码，完成如下要求。

- 结合代码分析虚拟页内存分配的三步过程和虚拟页内存释放。
- 构造测试例子来分析虚拟页内存管理的实现是否存在 bug。如果存在，则尝试修复并再次测试。否则，结合测例简要分析虚拟页内存管理的实现的正确性。
- **(不做要求，对评分没有影响)** 如果你有想法，可以在自己的理解的基础上，参考 ucore，《操作系统真象还原》，《一个操作系统的实现》等资料来实现自己的虚拟页内存管理。在完成之后，你需要指明相比较于本教程，你的实现的虚拟页内存管理的特点所在。

Part 1

申请

在第一个线程中，第 11 至 13 行依次请求了 3 次：

```
1 void first_thread(void *arg)
2 {
3     // 第1个线程不可以返回
4     // stdio.moveCursor(0);
5     // for (int i = 0; i < 25 * 80; ++i)
6     // {
7     //     stdio.print(' ');
8     // }
9     // stdio.moveCursor(0);
10
11     char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
12     char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
13     char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
14
15     printf("%x %x %x\n", p1, p2, p3);
16
17     memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
18     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
19
20     printf("%x\n", p2);
21
22     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
23
24     printf("%x\n", p2);
25
26     asm_halt();
27 }
```

申请内存的完整过程如下：

```
1 int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
2 {
```

```

3 // 第一步：从虚拟地址池中分配若干虚拟页
4 int virtualAddress = allocateVirtualPages(type, count);
5 if (!virtualAddress)
6 {
7     return 0;
8 }
9
10 bool flag;
11 int physicalPageAddress;
12 int vaddress = virtualAddress;
13
14 // 依次为每一个虚拟页指定物理页
15 for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
16 {
17     flag = false;
18     // 第二步：从物理地址池中分配一个物理页
19     physicalPageAddress = allocatePhysicalPages(type, 1);
20     if (physicalPageAddress)
21     {
22         //printf("allocate physical page 0x%x\n", physicalPageAddress);
23
24         // 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理
                页内。
25         flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
26     }
27     else
28     {
29         flag = false;
30     }
31
32     // 分配失败，释放前面已经分配的虚拟页和物理页表
33     if (!flag)
34     {
35         // 前i个页表已经指定了物理页
36         releasePages(type, virtualAddress, i);
37         // 剩余的页表未指定物理页
38         releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
39         return 0;
40     }
41 }
42
43 return virtualAddress;
44 }

```

第一步先向内核虚拟地址池申请 100 个页，申请的方式为 first-fit：

```

1 int BitMap::allocate(const int count)
2 {
3     if (count == 0)
4         return -1;
5
6     int index, empty, start;
7
8     index = 0;
9     while (index < length)
10    {
11        // 越过已经分配的资源

```



```

12     while (index < length && get(index))
13         ++index;
14
15     // 不存在连续的 count 个资源
16     if (index == length)
17         return -1;
18
19     // 找到 1 个未分配的资源
20     // 检查是否存在从 index 开始的连续 count 个资源
21     empty = 0;
22     start = index;
23     while ((index < length) && (!get(index)) && (empty < count))
24     {
25         ++empty;
26         ++index;
27     }
28
29     // 存在连续的 count 个资源
30     if (empty == count)
31     {
32         for (int i = 0; i < count; ++i)
33         {
34             set(start + i, true);
35         }
36
37         return start;
38     }
39 }
40
41 return -1;
42 }

```

这个过程在 Assignment 2 中有详细的说明，分配的 count 个页在内核虚拟地址池中是连续的。

```

1 int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
2 {
3     // 第一步：从虚拟地址池中分配若干虚拟页
4     int virtualAddress = allocateVirtualPages(type, count);
5     if (!virtualAddress)
6     {
7         return 0;
8     }
9
10    bool flag;
11    int physicalPageAddress;
12    int vaddress = virtualAddress;
13
14    // 依次为每一个虚拟页指定物理页
15    for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
16    {
17        flag = false;
18        // 第二步：从物理地址池中分配一个物理页
19        physicalPageAddress = allocatePhysicalPages(type, 1);
20        if (physicalPageAddress)
21        {
22            //printf("allocate physical page 0x%x\n", physicalPageAddress);

```

```

23
24         // 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理
           页内。
25         flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
26     }
27     else
28     {
29         flag = false;
30     }
31
32     // 分配失败，释放前面已经分配的虚拟页和物理页表
33     if (!flag)
34     {
35         // 前i个页表已经指定了物理页
36         releasePages(type, virtualAddress, i);
37         // 剩余的页表未指定物理页
38         releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
39         return 0;
40     }
41 }
42
43 return virtualAddress;
44 }

```

第二步，对第一步申请到的每一个虚拟页，从物理地址池中分配 1 页，采用一个 for 循环来遍历所有的虚拟页地址，对每一个虚拟页地址为其申请一个物理页，这些物理页在内存当中其实是不连续的。

从内核物理地址池中分配 1 一个物理页：

```

1         // 第二步：从物理地址池中分配一个物理页
2         physicalPageAddress = allocatePhysicalPages(type, 1);

```

第三步，若申请成功，将申请到的物理页映射到虚拟页地址上：

```

1         if (physicalPageAddress)
2         {
3             //printf("allocate physical page 0x%x\n", physicalPageAddress);
4
5             // 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理
               页内。
6             flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
7         }

```

将物理页映射到虚拟页地址的具体过程如下：

```

1 bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int
           physicalPageAddress)
2 {
3     // 计算虚拟地址对应的页目录项和页表项
4     int *pde = (int *)toPDE(virtualAddress);
5     int *pte = (int *)toPTE(virtualAddress);
6
7     // 页目录项无对应的页表，先分配一个页表
8     if (!(*pde & 0x00000001))
9     {
10         // 从内核物理地址空间中分配一个页表
11         int page = allocatePhysicalPages(AddressPoolType::KERNEL, 1);

```

```

12     if (!page)
13         return false;
14
15     // 使页目录项指向页表
16     *pde = page | 0x7;
17     // 初始化页表
18     char *pagePtr = (char *)(((int)pte) & 0xffff000);
19     memset(pagePtr, 0, PAGE_SIZE);
20 }
21
22 // 使页表项指向物理页
23 *pte = physicalPageAddress | 0x7;
24
25 return true;
26 }
27 int MemoryManager::toPDE(const int virtualAddress)
28 {
29     return (0xffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
30 }
31
32 int MemoryManager::toPTE(const int virtualAddress)
33 {
34     return (0xffc00000 + ((virtualAddress & 0xffc00000) >> 10) + (((virtualAddress & 0
35         x003ff000) >> 12) * 4));

```

映射的过程其实就是为物理地址创建页目录项和页表项，先查看虚拟地址对应的页目录项中是否有页表，虚拟地址的高 10 位就是在 cr3 寄存器中页目录表的物理地址，若页目录项的最后一位是 P 位（存在位），若存在对应页表只需使页表项指向物理页即可。

若其对应的页目录项不存在，则需要申请一个物理页用于放置页表，随后将页目录项指向页表，并对页表进行初始化。

这样就完成了 1 页虚拟页的分配，若申请失败则返回 false。

```

1     // 分配失败，释放前面已经分配的虚拟页和物理页表
2     if (!flag)
3     {
4         // 前 i 个页表已经指定了物理页
5         releasePages(type, virtualAddress, i);
6         // 剩余的页表未指定物理页
7         releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
8         return 0;
9     }

```

循环结束后，每一个虚拟页地址都获得了物理页的空间分配。

释放

下面进行内存页的释放。

```

1 void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const
2     int count)

```

```

3     int vaddr = virtualAddress;
4     int *pte;
5     for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
6     {
7         // 第一步, 对每一个虚拟页, 释放为其分配的物理页
8         releasePhysicalPages(type, vaddr2paddr(vaddr), 1);
9
10        // 设置页表项为不存在, 防止释放后被再次使用
11        pte = (int *)toPTE(vaddr);
12        *pte = 0;
13    }
14
15    // 第二步, 释放虚拟页
16    releaseVirtualPages(type, virtualAddress, count);
17 }

```

释放的过程和申请的过程相反, 先释放掉物理页, 再重置对应页表项, 最后将虚拟页从虚拟地址池中释放。

需要通过转换找到其对应的物理页, 再将该物理页释放, 虚拟页到物理页的转化如下:

```

1 int MemoryManager::vaddr2paddr(int vaddr)
2 {
3     int *pte = (int *)toPTE(vaddr);
4     int page = (*pte) & 0xfffff000;
5     int offset = vaddr & 0xfff;
6     return (page + offset);
7 }

```

将页表中重置后将虚拟页释放:

```

1 // 第二步, 释放虚拟页
2 releaseVirtualPages(type, virtualAddress, count);

```

Part 2

以我的能力暂时还找不到 BUG。

结合教程所用的测试例子说明其正确性。

```

1 void first_thread(void *arg)
2 {
3     // 第1个线程不可以返回
4     // stdio.moveCursor(0);
5     // for (int i = 0; i < 25 * 80; ++i)
6     // {
7     //     stdio.print(' ');
8     // }
9     // stdio.moveCursor(0);
10
11     char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
12     char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
13     char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
14
15     printf("%x %x %x\n", p1, p2, p3);

```

```

16
17     memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
18     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
19
20     printf("%x\n", p2);
21
22     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
23
24     printf("%x\n", p2);
25
26     asm_halt();
27 }

```

它先依次向内核虚拟地址池申请虚拟页（共进行 3 次申请），页数分别为 100、10、100，随后释放掉第 2 次申请的虚拟页，内核物理地址池的存储空间分布如下：

100	10	100	rest
-----	----	-----	------

再依次申请 100、10 个虚拟页，最后的输出如下：

```

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

Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0x10000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
C0100000 C0164000 C016E000
C01D2000
C0164000

```

第一次申请 100 个物理页，起始地址为 0xC0100000，即内核虚拟地址池的起始地址，第二次申请 10 个物理页，起始地址为 0xC0164000，由于每个页大小为 4KB，第一次申请的地址空间为 0xC0100000-0xC0163fff，第二次申请的起始地址正是从第一次申请的末尾。第三次申请同理。

释放掉第二次申请的 10 个虚拟页后，空闲出来的 10 个虚拟页空间无法满足第四次申请所需要的 100 个虚拟页的空间，因此第四次申请空间接在第三次申请的后面，即

0xC01D2000。最后申请 10 个虚拟页，第二次申请后释放的 10 个虚拟页的空间满足需求，因此起始地址为 0xC0164000。

总结

对一些 Assignment 的要求不太明白，但最终还是完成了实验。
总而言之，此次实验使我受益匪浅。