



本科生实验报告

实验课程：_____操作系统_____

实验名称：_____

专业名称：_____网络空间安全_____

学生姓名：_____陈浚铭_____

学生学号：_____20337021_____

实验地点：_____温暖的家_____

实验成绩：_____

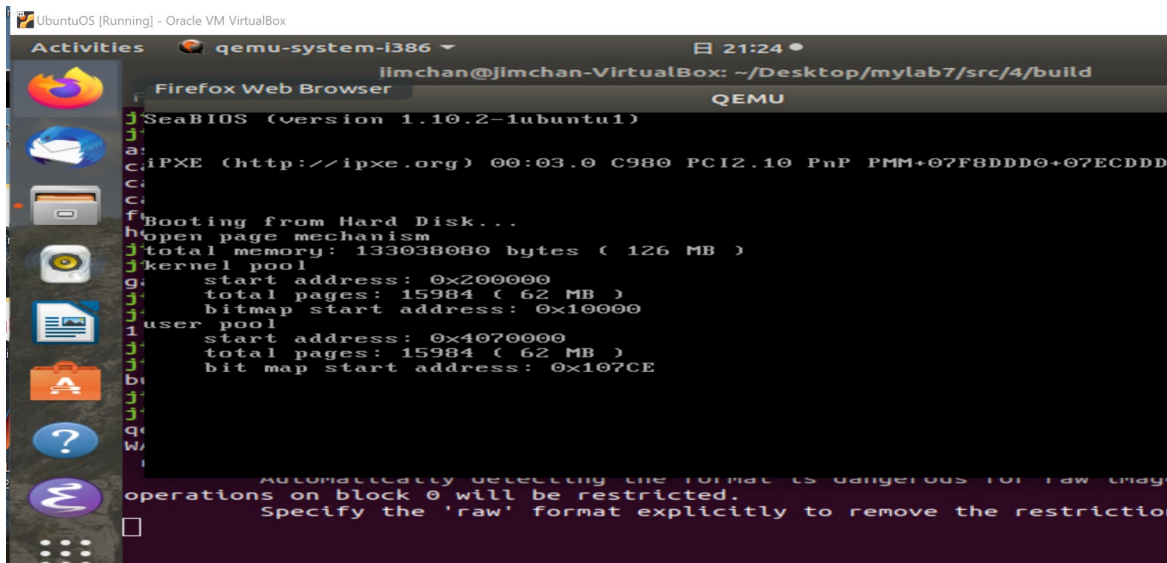
报告时间：_____

1. 实验要求

通过这次实验深入了解二级分页机制，理论课上学习到的动态分区算法和页面置换算法和虚拟内存机制。

2. 实验过程和实验结果

Assignment 1:



在这里我们通过二级分页机制来把虚拟地址转换成物理地址。二级分页的好处是能够减小使用内存的空间。这是因为在一级页表机制，我们在生成每一个进程的时候，进程空间里都会有 4Mb 空间的页表。而在二级分页机制，我们有一个 4KB 的页目录表，而每一个进程的页表大小为 4KB，这样就减小了一个 PCB 里面的页表所需要的内存空间。理论上看起来分页的级数越多就越好，因为理论上使用的内存有可能越小，但是理论课上刘老师说过目前极限为 7 级分页机制，这是因为分页机制的级数越大，有效寄存器的访问时间越长，减小效率。

二级分页机制的结构一个页目录表当中的么一页指向对应一个页表，而该页表指向物理地址。以 32 位地址空间为例。如果令页目录表，页表和物理页大小均为 4KB, 而页目录项和页表项为 4B (32 位地址)，那么我们有 $4KB/4B = 1024$ 个页目录项，因为每一个页目录项对应一个页表，所以我们有 1024 个页表。在每一个页表里面，页表项的数目为 $4KB/4B = 1024$ 个。因此我们总共有 $1024 \times 1024 = 1M$ 个物理页，而每一个物理页的大小为 4KB，所以我们有 $1024 \times 4KB = 4GB$ 的总内存大小。

Assignment 2:

在这个实验里面我们需要通过首次适合 (First fit)，最优适合 (Best fit) 来剖析动态储存分配问题 (dynamic storage allocation problem)。我们这里实现的是可变分区的方案：在操作系统有一个表，用于记录那些内存可用和那些内存已用。开始的时候，所有内存都可用于用户内存，因此可以看作为一个很大的孔。我们的算法需要记录内存里孔的集合。

在任何时候，都有一个可用块大小的列表和输入队列。内存会不断分配给进程，知道下一个进程的内存需求不能满足为止，这是没有足够的可用块 (或孔) 来加载进程。

实验部分：注意我们在这里是使用 /src/3/ 文件夹里面的代码改进代码的，也就是使用到物理页内存管理的代码

部分而没有用到二级机制或则虚拟内存的知识点（因为所需要显示的知识点不需求）。在这里我们是使用使用分页机制，也就是内存以物理页为单位的内存机制。我们对于用户物理内存为例子。假设内存已经被其他 4 个用户进程占用（我们在这里不把他们的名字进行分类，直接叫他们为 OCCUPIED，大小都为 14284 KB），然后当中先拥有 5 个小孔大小分别为 400KB, 2000KB, 800KB, 2400KB。

```
jimchan@jimchan-VirtualBox: ~/Desktop/mylab7/src/3/build
QEMU
total pages: 15984 ( 62 MB )
bitmap start address: 0x10000
user pool
start address: 0x4070000
total pages: 15984 ( 62 MB )
bit map start address: 0x107CE
-----
Free space: Page 0 to 99 (400 KB)
-----
Occupied: Page 100 to 3670 (14284 KB)
-----
Free space: Page 3671 to 4170 (2000 KB)
-----
Occupied: Page 4171 to 7741 (14284 KB)
-----
Free space: Page 7742 to 7941 (800 KB)
-----
Occupied: Page 7942 to 11512 (14284 KB)
-----
Free space: Page 11513 to 11812 (1200 KB)
-----
Occupied: Page 11813 to 15383 (14284 KB)
-----
Free space: Page 15384 to 15983 (2400 KB)
```

之后我们依次序输入 P1: 414KB, P2: 1668KB,
P3: 448KB, P4: 1704KB。

我们使用 **first fit** 会期望得到以下的结果。当中 P4 不能够进入内存（没有足够大的孔）

```
Jimchan@Jimchan-VirtualBox: ~/Desktop/mylab7/src/3/build
QEMU
-----
Free space: Page 0 to 99 (400 KB)
-----
Occupied: Page 100 to 3670 (14284 KB)
-----
Process 1: Page 3671 to 3882 (848 KB)
-----
Process 3: Page 3883 to 3994 (448 KB)
-----
Free space: Page 3995 to 4170 (704 KB)
-----
Occupied: Page 4171 to 7741 (14284 KB)
-----
Free space: Page 7742 to 7941 (800 KB)
-----
Occupied: Page 7942 to 11512 (14284 KB)
-----
Free space: Page 11513 to 11812 (1200 KB)
-----
Occupied: Page 11813 to 15383 (14284 KB)
-----
Process 2: Page 15384 to 15800 (1668 KB)
-----
Free space: Page 15801 to 15983 (732 KB)
-----
WARNING: image format was not specified for ../run/init.img and pro
raw.
Automatically detecting the format is dangerous for raw im
```

而如果使用 best fit 的话我们会得到以下的结果。 注意这里我们 4 个进程都能够成功载入内存里面，相比 first fit 更优。

```
UbuntuOS [Running] - Oracle VM VirtualBox
Activities qemu-system-i386 16:52
QEMU
-----
Occupied: Page 100 to 3670 (14284 KB)
-----
Process 2: Page 3671 to 4087 (1668 KB)
-----
Free space: Page 4088 to 4170 (332 KB)
-----
Occupied: Page 4171 to 7741 (14284 KB)
-----
Process 3: Page 7742 to 7853 (448 KB)
-----
Free space: Page 7854 to 7941 (352 KB)
-----
Occupied: Page 7942 to 11512 (14284 KB)
-----
Process 1: Page 11513 to 11724 (848 KB)
-----
Free space: Page 11725 to 11812 (352 KB)
-----
Occupied: Page 11813 to 15383 (14284 KB)
-----
Process 4: Page 15384 to 15809 (1704 KB)
-----
Free space: Page 15810 to 15983 (696 KB)
-----
```

Assignment 3:

使用 FIFO 页面置换

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2	2	4	4	4	0		0	0		7	7	7
	0	0	0		3	3	3	2	2	2		1	1		1	0	0
		1	1		1	0	0	0	3	3		3	2		2	2	1

page frames

Figure 10.12 FIFO page-replacement algorithm.

```
jimchan@jimchan-VirtualBox: ~/Desktop/originalLab7/src/5/build
QEMU

Booting from Hard Disk...
open page mechanism
1: Page Fault, added page table entry: virtual page 7(28672) to frame 2097152
2: Page Fault, added page table entry: virtual page 0(0) to frame 2101248
3: Page Fault, added page table entry: virtual page 1(4096) to frame 2105344
4: swapped out page 7 (28672) swapped in page 2(8192), maps to frame 2097152
5: Success: page 0(0) to physical addr(2101248)
6: swapped out page 0 (0) swapped in page 3(12288), maps to frame 2101248
7: swapped out page 1 (4096) swapped in page 0(0), maps to frame 2105344
8: swapped out page 2 (8192) swapped in page 4(16384), maps to frame 2097152
9: swapped out page 3 (12288) swapped in page 2(8192), maps to frame 2101248
10: swapped out page 0 (0) swapped in page 3(12288), maps to frame 2105344
11: swapped out page 4 (16384) swapped in page 0(0), maps to frame 2097152
12: Success: page 3(12288) to physical addr(2105344)
13: Success: page 2(8192) to physical addr(2101248)
14: swapped out page 2 (8192) swapped in page 1(4096), maps to frame 2101248
15: swapped out page 3 (12288) swapped in page 2(8192), maps to frame 2105344
16: Success: page 0(0) to physical addr(2097152)
17: Success: page 1(4096) to physical addr(2101248)
18: swapped out page 0 (0) swapped in page 7(28672), maps to frame 2097152
19: swapped out page 1 (4096) swapped in page 0(0), maps to frame 2101248
20: swapped out page 2 (8192) swapped in page 1(4096), maps to frame 2105344
v)
raw.
Automatically detecting the format is dangerous for raw images, write
```

使用 LRU 页面置换：

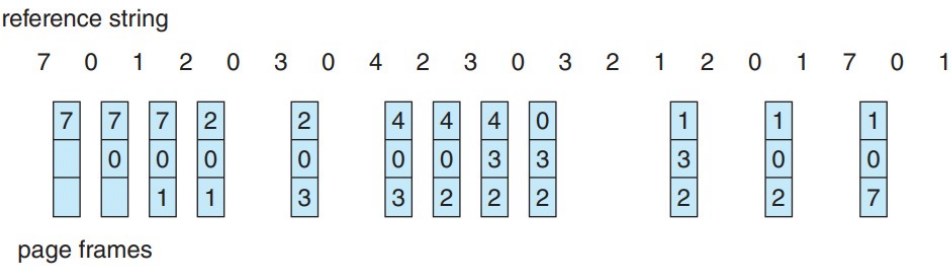
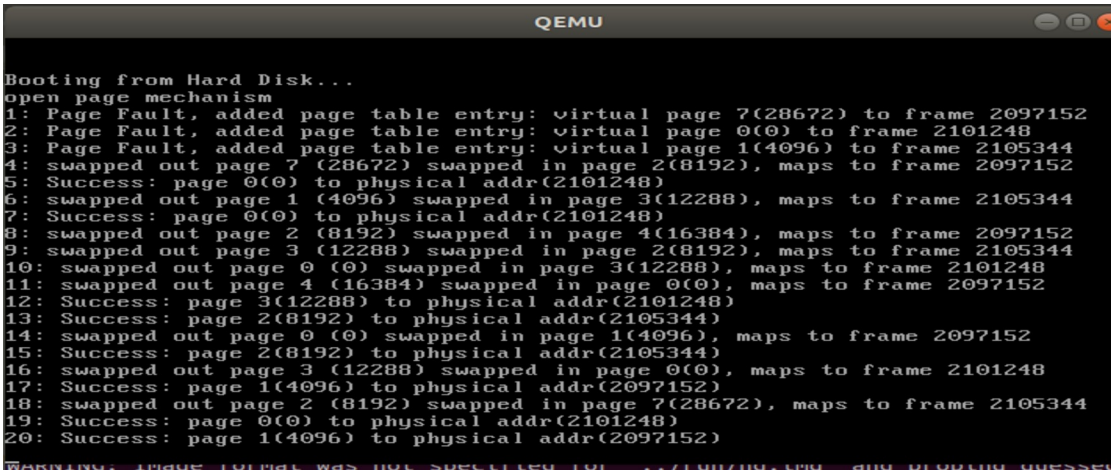


Figure 10.15 LRU page-replacement algorithm.



页面置换算法的原理是当我们因为所查询的页不在内存里面，而因为没有空的物理帧让我们分配进去，因此要通过找出一个虚拟页，把它从释放到备用储存（硬盘里），让后把我们需要的虚拟页分配到该物理帧的位置上。选择哪一个物理帧来进行我们的置换有不同的算法，其中比较典型的有FIFO，LRU等。

我们在这个实验上通过 reference bit 定义为我们需要在内存朝访问虚拟页对应的页项，从而可以分别出我们所访问的虚拟也。在置换算法的例子演示中，我们通过 reference bit, 只理会我们

所访问的虚拟页，而不理会偏移量。

在虚拟页访问有三种情况：

(1): 如果我们命中虚拟页，该虚拟页在内存里，我们就成功访问。

(2): 如果我们访问虚拟页命中失败，如果有空物理帧，就把从磁盘上找出虚拟页并分配到该物理帧。如果没有空物理帧，就需要通过置换算法的机制来进行页面置换。

我们发现 FIFO 的置换算法有 15 次的命中失败，而 LRU 的置换算法有 12 次的命中失败。而根据 Belady' s anomaly 知道有些置换算法随着物理页的数量增加，命中失败率页会随着增加，而 FIFO 置换算法就是其中之一。我们发现 LRU 置换算法是比 FIFO 好的，而且是在操作系统经常使用的。

assignment 4:

1. 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。

分配虚拟页内存内存的三个过程为：

(1) 从虚拟地址池中分配连续的多个虚拟页。注意，虚拟页之间的虚拟地址是连续的。

(2) 从物理地址池中为每一个虚拟页分配相应大小的物理页。

(3) 在页目录表和页表中建立虚拟页和物理页之间的对应关系。此时，由于分页机制的存在，物理页的地址可以不连续。CPU 的 MMU 会在程序执行过程中将虚拟地址翻译成物理地址。

MemoryManager::allocatePages(enum AddressPoolType type, const int count) 的函数能够实现以上的三个步骤：

```
166 int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
167 {
168     // 第一步：从虚拟地址池中分配若干虚拟页
169     int virtualAddress = allocateVirtualPages(type, count);
170     if (!virtualAddress)
171     {
172         return 0;
173     }
174
175     bool flag;
176     int physicalPageAddress;
177     int vaddress = virtualAddress;
178
179     // 依次为每一个虚拟页指定物理页
180     for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
181     {
182         flag = false;
183         // 第二步：从物理地址池中分配一个物理页
184         physicalPageAddress = allocatePhysicalPages(type, 1);
185         if (physicalPageAddress)
186         {
187             //printf("allocate physical page 0x%x\n", physicalPageAddress);
188
189             // 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。
190             flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
191
192         }
193         else
194         {
195             flag = false;
196         }
197
198         // 分配失败，释放前面已经分配的虚拟页和物理页表
199         if (!flag)
200         {
201             // 前i个页表已经指定了物理页
202             releasePages(type, virtualAddress, i);
203             // 剩余的页表未指定物理页
204             releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
205             return 0;
206         }
207     }
208     return virtualAddress;
209 }
```

对于第（1）步： 第169 行通过 allocateVirtualPages 分配连续

的 `count` 个页，并且返回开始的虚拟地址为 `virtualAddress`。我们同通过第 170-173 行的 `if` 语句来判断是否成功分配连续虚拟页。之后，对于第（2）步，**我们需要对于每一个虚拟页连接上一个物理页。**

通过 `for` 语句我们对于每一个虚拟页进行物理页的连接，当中在第 180 行的 `for` 语句我们通过 `vaddress += PAGE_SIZE` 来在每一次循环中更新到下一个虚拟页的地址。在 `for` 语句的代码块中 我们同通过 `allocatePhysicalPages()` 函数来分配一个物理页，我们在这里传递的参数为 `type = ADDRESS_POOL_TYPE::KERNEL`，`count = 1`，输出为 `physicalPageAddress`（物理页地址）。这样就能够分配一个内核的物理页。如果我们不能够成功分配一个物理页，那么我们就需要对于之前已经分配的虚拟也以及对应连接的物理进行页释放，并 `return 0` 以表示不能够对于连续的虚拟页分配到内存理。

这段代码在第 198-205 行实现到。最后，对于第三步，我们通过 `connectPhysicalVirtualPage()` 来连接虚拟页和物理页。

对于页内存释放，通过

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

实现这个功能。释放内存意味着**释放虚拟页以及相应物理页**。我们在 `releasePages` 传递了 `type`（释放的内存类型分别为 `user` 还是

kernel), virtualAddress (开始释放的虚拟页地址), count (释放的页数量)。

```
void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int count)
{
    int vaddr = virtualAddress;
    int *pte, *pde;
    bool flag;
    const int ENTRY_NUM = PAGE_SIZE / sizeof(int);

    for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
    {
        releasePhysicalPages(type, vaddr2paddr(vaddr), 1);

        // 设置页表项为不存在, 防止释放后被再次使用
        pte = (int *)toPTE(vaddr);
        *pte = 0;
    }

    releaseVirtualPages(type, virtualAddress, count);
}
```

在 releasePages() 函数, 我们首先需要释放每一个虚拟页对应的物理页。通过 for 语句, 从 vaddr = virtualAddress 开始, 我们通过 for 语句对于每一个虚拟页对应的物理页进行释放, 当中 vaddr += Page_SIZE 用来更新每次循环的虚拟页。在 for 模块里面, 通过 releasePhysicalPages(type, vaddr2paddr(vaddr), 1) 来释放虚拟页对应的物理页。之后, 我们需要在对应虚拟页地址的页表项设置为 0, 从而表示页表项存在, 防止释放后被再次使用。在释放了所有物理页以后, 通过 releaseVirtualPage(type, virtualAddress, count) 释放对应用户还是内核的以 virtualAddress 虚拟页地址为开始的连续 count 个虚拟也。

注意我们在函数里的 for 模块当中调用 releasePhysicalPages 函数

时使用到 `vaddr2paddr` 函数，它的功能就是把虚拟地址转换成物理地址。把虚拟地址转换成物理地址需要通过虚拟地址给出的信息找出物理地址的位置：也就是它对应的**页目录项**和**页表项**，从而找出物理地址。原理如下：

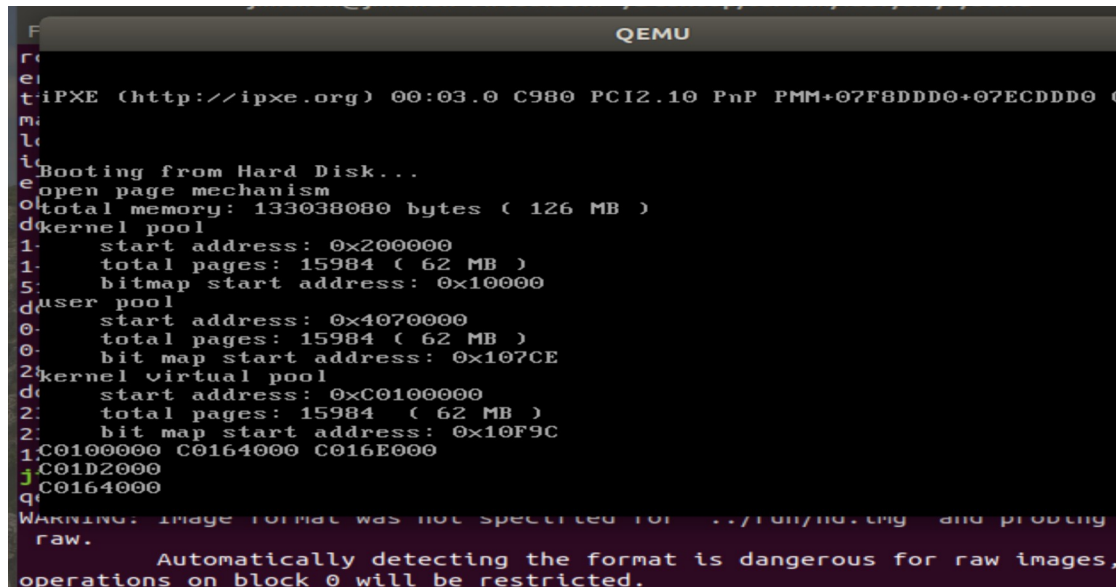
1. 通过 `virtualAddress[31:22]`，在寄存器找出页目录表的物理地址，通过这个物理地址找出第 `virtualAddress[31:22]` 的页目录项，并读出页目录项下的页表的物理地址。

2. 通过 `virtualAddress[31:22]`，通过第一步找出的页目录表的物理地址，找出第 `virtualAddress[31:22]` 的页表项，并读出页表项中的物理页的物理地址 `physicalAddress`。

3. 通过代替 `virtualAddress[31:12]` 为 `physicalAddress[31:12]` 得到物理页的物理地址为

`physicalAddress[31:12] : virtualAddress[11:0]`。

我们执行 src/5/ 里面的代码得到以下输出：



```
QEMU
Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
1: start address: 0x200000
1: total pages: 15984 ( 62 MB )
5: bitmap start address: 0x10000
user pool
0: start address: 0x4070000
0: total pages: 15984 ( 62 MB )
0: bit map start address: 0x107CE
kernel virtual pool
0: start address: 0xC0100000
2: total pages: 15984 ( 62 MB )
2: bit map start address: 0x10F9C
1: C0100000 C0164000 C016E000
j: C01D2000
q: C0164000
Warning: image format was not specified for ../run/hd.img and probing
raw.
Automatically detecting the format is dangerous for raw images,
operations on block 0 will be restricted.
```

```
17
18
19 void first_thread(void *arg)
20 {
21     char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
22     char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
23     char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
24
25     printf("%x %x %x\n", p1, p2, p3);
26
27     memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
28     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
29
30     printf("%x\n", p2);
31
32     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
33
34     printf("%x\n", p2);
35     asm_halt();
36 }
```

我们从第 20-22 行的代码是要在 Kernel 的虚拟和物理地址进行 3 次分配。我们知道 kernel 的开始虚拟地址是在 0xc0100000。我们分别分配内存大小为 100 页， 10 页， 100 页。我们观察到：



```
C0100000 C0164000 C016E000
```

因为 1 页 为 4K bytes = 4096 bytes, 而么一地址对应 1 byte (字节)

第一个分配的内存空间为 C010000 到 C0164000, 分配内

存空间 (C0164000 - C010000) = 64000 = $6*16*16*16*16$
+ $4*16*16*16$ = **409600** bytes,

页数 = 409600 bytes / 4096 bytes = 100 页。

第二个分配的内存空间为 C0164000 到 C016E000。分配内存
空间 = C016E000 - C0164000 = B000 = $10*16*16*16$ =
40960 bytes

页数 = 40960 bytes / 4096 bytes = 10 页。



A screenshot of a memory dump with a black background and green text. It shows four memory addresses: C0100000, C0164000, C016E000, and C01D2000, arranged in two rows of two.

注意这里第 4 个地址为 C01D2000 我们知道第三个地址
C016E000 开始有 100 个连续的物理帧被分配个 100 个虚拟页。

由分配内存空间 = C01D2000 - C016E000 = 409600

页数 = 409600 bytes / 4096 bytes = 100 页。

因此与我们的期望符合。

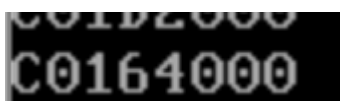
我们先把 p2 的 10 个虚拟页释放到

```
memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
```

而之后我们再分配了 10 个虚拟页：

```
p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
```

而我们发现它的开始物理地址跟之前 p2 释放前的物理地址
一样。



A screenshot of a memory dump with a black background and green text. It shows two memory addresses: C01D2000 and C0164000, arranged in two rows of one.

这是因为我们分配内存的机制是把从最低的地址开始，找出可以最小的“孔”能够分配我们所需要的虚拟页数量。

关键代码

assignment 2:

对应用户或则内核空间进行首次或者最优适合。

```
void MemoryManager::firstFit(int processMemorySize[4], enum AddressPoolType type)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelPhysical.firstFit(processMemorySize);
    }
    else if (type == AddressPoolType::USER)
    {
        userPhysical.firstFit(processMemorySize);
    }
}

void MemoryManager::bestFit(int processMemorySize[4], enum AddressPoolType type)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelPhysical.bestFit(processMemorySize);
    }
    else if (type == AddressPoolType::USER)
    {
        userPhysical.bestFit(processMemorySize);
    }
}
```

最优适合算法:

```
void BitMap::firstFit(int processMemorySize[4])
{
    memoryType processList[4] = {P1, P2, P3, P4};
    for(int i = 0; i < 4; i++)
    {
        int start = allocate(processMemorySize[i]/4);
        if(start != -1)
        {
            for(int j = 0; j < processMemorySize[i]/4; j++)
            {
                memoryTypeList[start + j] = processList[i];
            }
        }
    }
}
```


找出最优适合的“孔”的物理地址的算法

```
int BitMap::findBestFitAddress(int processMemoryPages)
{
    int bestFitStartAddress;
    int numOfUnallocatedPages = numOfUserPhysicalPages;
    int tempStartAddress;
    int tempNumOfUnallocatedPages = 0;
    int unallocatedFlag = 0;
    for(int i = 0; i < numOfUserPhysicalPages; i++)
    {
        if(get(i) == 0 && !unallocatedFlag) // free
        {
            unallocatedFlag = 1;
            tempStartAddress = i;
            tempNumOfUnallocatedPages++;
        }
        if(get(i) == 0 && unallocatedFlag)
            tempNumOfUnallocatedPages++;

        if((get(i) == 1 || i == 15983) && unallocatedFlag)
        {
            if(processMemoryPages < tempNumOfUnallocatedPages
                && tempNumOfUnallocatedPages < numOfUnallocatedPages)
            {
                numOfUnallocatedPages = tempNumOfUnallocatedPages;
                bestFitStartAddress = tempStartAddress;
            }
            tempNumOfUnallocatedPages = 0;
            unallocatedFlag = 0;
        }
        else continue;
    }
    return numOfUnallocatedPages == numOfUserPhysicalPages ? -1 : bestFitStartAddress;
}
```

最优适合算法

```
void BitMap::bestFit(int processMemorySize[4])
{
    memoryType processList[4] = {P1, P2, P3, P4};
    for(int i = 0; i < 4; i++)
    {
        int start = findBestFitAddress(processMemorySize[i]/4);

        if(start != -1)
        {
            for(int j = 0; j < processMemorySize[i]/4; j++)
            {
                set(start+j, 1);
                memoryTypeList[start + j] = processList[i];
            }
        }
    }
}
```

assignment 3

演示置换算法的线程

```
void first_thread(void *arg)
{
    memoryManager.initFIFOQueue();
    int queryVAddrList[] = {7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1};
    //memoryManager.queryListOfKernelPhysicalPagesFIFO(queryVAddrList, 20);

    memoryManager.queryListOfKernelPhysicalPagesLRU(queryVAddrList, 20);

    asm_halt();
}
```

FIFO 置换算法

```
void MemoryManager::queryKernelPhysicalPageFIFO(const int referenceBit)
{
    int vaddr = referenceBit * PAGE_SIZE;
    if(kernelVirtual.get(referenceBit) == 0) // unallocated
    {
        if(FIFOQueue.isFull())
        {
            int *addressPair = FIFOQueue.dequeue();
            int firstInVAddr = addressPair[0];
            int firstInPAddr = addressPair[1];

            // swap in and swap out virtual page
            // remove old virtual page and allocate new virtual page
            int virtualPageToReplace = firstInVAddr / PAGE_SIZE;
            kernelVirtual.set(virtualPageToReplace, 0);
            int *pte = (int *)toPTE(firstInVAddr);
            *pte = 0; // set the virtual address that we swap out to 0
            kernelVirtual.set(referenceBit, 1);

            // set the page table entry of virtual address to physical address
            int flag = connectPhysicalVirtualPage(vaddr, firstInPAddr);
            if(!flag)
            {
                printf("ERROR: Failed to make page directory entry and page table entry for mapping virtual page to physical frame\n");
            }
            int virtualPageToSwapOut = firstInVAddr / PAGE_SIZE;
            int virtualPageToSwapIn = referenceBit;
            printf("swapped out page %d (%d) swapped in page %d (%d), maps to frame %d\n", virtualPageToSwapOut, firstInVAddr, virtualPageToSwapIn, vaddr, firstInPAddr);
            int newAddressPairItem[] = {vaddr, firstInPAddr};
            FIFOQueue.enqueue(newAddressPairItem);
        }
        else // not full
        {
            int mappingPAddr = allocatePhysicalPages(AddressPoolType::KERNEL, 1);

            if(!mappingPAddr)
            {
                printf("ERROR: Failed to allocate physical page");
                return;
            }

            kernelVirtual.set(referenceBit, 1); // allocate a virtual page
            int flag = connectPhysicalVirtualPage(vaddr, mappingPAddr);
            if(!flag)
            {
                printf("ERROR: Failed to make page directory entry and page table entry for mapping virtual page to physical frame\n");
            }
            int virtualPage = referenceBit;
            printf("Page Fault, added page table entry: virtual page %d (%d) to frame %d\n", virtualPage, vaddr, vaddr2paddr(vaddr));
            int newAddressPair[] = {vaddr, mappingPAddr};
            FIFOQueue.enqueue(newAddressPair);
        }
    }
    else
    {
        printf("Success: page %d (%d) to physical addr (%d)\n", referenceBit, vaddr, vaddr2paddr(vaddr));
    }
}
```

LRU 置换算法

```
void MemoryManager::queryKernelPhysicalPageLRU(int referenceBit)
{
    int vaddr = referenceBit * PAGE_SIZE;
    int vaddrToReplace;
    enum QUERY_TYPE queryResult = LRUHeap.append(&vaddrToReplace, vaddr);
    if(queryResult == MISS_AND_APPEND)
    {
        int mappingPAddr = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
        if(!mappingPAddr)
        {
            printf("ERROR: Failed to allocate physical page");
            return;
        }
        kernelVirtual.set(referenceBit, 1); // allocate a virtual page
        int flag = connectPhysicalVirtualPage(vaddr, mappingPAddr);
        if(!flag)
        {
            printf("ERROR: Failed to make page directory entry and page table entry for mapping virtual page to physical frame\n");
        }
        int virtualPage = referenceBit;
        printf("Page Fault, added page table entry: virtual page %d(%d) to frame %d\n", virtualPage, vaddr, vaddr2paddr(vaddr));
        LRUArrayList.append(vaddr, mappingPAddr);
    }
    else if(queryResult == MISS_AND_REPLACE)
    {
        /*
        int paddr = vaddr2paddr(vaddrToReplace);
        kernelVirtual.set(vaddrToReplace, 0);
        connectPhysicalVirtualPage(vaddr, paddr);
        kernelVirtual.set(vaddr, 1);
        printf("Replaced virtual page %d with %d, maps to physical page %d\n", vaddrToReplace, vaddr, paddr);
        */

        int paddr = LRUArrayList.at(vaddrToReplace);
        // swap in and swap out virtual page
        // remove old virtual page and allocate new virtual page
        //releaseVirtualPages(AddressPoolType::KERNEL, vaddrToReplace, 1);
        int virtualPageToReplace = vaddrToReplace / PAGE_SIZE;
        kernelVirtual.set(virtualPageToReplace, 0);
        int *pte = (int *)toPTE(vaddrToReplace);
        *pte = 0; // set the virtual address that we swap out to 0
        kernelVirtual.set(referenceBit, 1);

        // set the page table entry of virtual address to physical address
        int flag = connectPhysicalVirtualPage(vaddr, paddr);
        if(!flag)
        {
            printf("ERROR: Failed to make page directory entry and page table entry for mapping virtual page to physical frame\n");
        }
        int virtualPageToSwapOut = vaddrToReplace / PAGE_SIZE;
        int virtualPageToSwapIn = referenceBit;
        printf("swapped out page %d(%d) swapped in page %d(%d), maps to frame %d\n", virtualPageToSwapOut, vaddrToReplace, virtualPageToSwapIn, vaddr, paddr);
        LRUArrayList.remove(vaddrToReplace);
        LRUArrayList.append(vaddr, paddr);
    }
    else if(queryResult == HIT)
    {
        printf("Success: page %d(%d) to physical addr(%d)\n", referenceBit, vaddr, vaddr2paddr(vaddr));
    }
}
```

一些辅助的数据结构:

FIFO 队列

```
Queue::Queue()
{
    capacity = MEM_PAGES_TO_ALLOCATE;
    front = 0;
    rear = -1;
    count = 0;
}

void Queue::init(int size)
{
    this->capacity = 3;
    this->front = 0;
    this->rear = -1;
    this->count = 0;
}

int Queue::getCapacity()
{
    return capacity;
}

int *Queue::dequeue()
{
    if(isEmpty())
        printf("Underflow terminated\n");
    int *x = arr[front];

    front = (front + 1) % capacity;
    count--;

    return x;
}

void Queue::enqueue(int item[2])
{
    if(isFull())
    {
        printf("Overflow");
    }

    rear = (rear+1) % capacity;
    arr[rear][0] = item[0]; // virtual address
    arr[rear][1] = item[1]; // physical address
    count++;
}

int *Queue::peek()
{
    return arr[front];
}

int Queue::size()
{
    return count;
}

bool Queue::isEmpty()
{
    return (size() == 0);
}

bool Queue::isFull()
{
    return (size() == capacity);
}
```

实现 LRU 置换算法的堆栈和 arrayList

```
bool MyHeap::isFull()
{
    return (count == MEM_PAGES_TO_ALLOCATE);
}

int MyHeap::findIndex(int elem)
{
    for(int i = 0; i < count; i++)
    {
        if(elem == arr[i])
        {
            return i;
        }
    }
    return -1;
}

enum QUERY_TYPE MyHeap::append(int *replacedVAddr, int elem)
{
    int elemIndex = findIndex(elem);
    if(elemIndex == -1) // cannot find virtual address int the LRU heap
    {
        if(!isFull())
        {
            for(int i = count++; i > 0; i--)
            {
                arr[i] = arr[i-1];
            }
            arr[0] = elem;
            return MISS_AND_APPEND;
        }
        else
        {
            *replacedVAddr = popback();
            for(int i = count-1; i > 0; i--)
            {
                arr[i] = arr[i-1];
            }
            arr[0] = elem;

            return MISS_AND_REPLACE;
        }
    }
    else // hit
    {
        for(int i = elemIndex; i > 0; i--)
        {
            arr[i] = arr[i-1];
        }
        arr[0] = elem;

        return HIT;
    }
}

int MyHeap::popback()
{
    return arr[count-1];
}
```

```

-
#define MEM_PAGES_TO_ALLOCATE 3
#define HEAPSIZE 3

int ArrayList::at(int vaddr)
{
    for(int i = 0; i < len; i++)
    {
        if(arr[i][0] == vaddr)
        {
            return arr[i][1];
        }
    }
}

void ArrayList::append(int vaddr, int paddr)
{
    arr[len][0] = vaddr;
    arr[len][1] = paddr;
    len++;
}

void ArrayList::remove(int vaddr)
{
    for(int i = 0; i < len; i++)
    {
        if(arr[i][0] == vaddr)
        {
            for(int j = i; j < len; j++)
            {
                arr[j][0] = arr[j+1][0];
                arr[j][1] = arr[j+1][1];
            }
            len--;
        }
    }
}

void MyHeap::init()
{
}

bool MyHeap::isFull()
{
    return (count == MEM_PAGES_TO_ALLOCATE);
}

int MyHeap::findIndex(int elem)
{
    for(int i = 0; i < count; i++)
    {
        if(elem == arr[i])
        {
            return i;
        }
    }

    return -1;
}
-

```

3. 总结

在实验通过不断的尝试，让我更深入理解虚拟内存的原理，其中因为对当中的理论理解错误，而有多次失败。最后通过实验的实现终于能够更正了对理论的错误！！