本科生实验报告

实验课程:操作系统原理实验

实验名称:实验七

专业名称:计算机科学与技术

学生姓名:邓栩瀛

学生学号:21307035

实验地点:东校园实验中心 D402

实验成绩:

报告时间:2023.6.5

lab7 内存管理

1、实验要求

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。

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。

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。

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。

2、实验过程+实验结果

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。

物理页的内存管理

使用两个地址池（内核地址池和用户地址池）来对这两部分物理地址进行管理

```
class MemoryManager
{
public:
    // 可管理的内存容量
    int totalMemory;
    // 内核物理地址池
    AddressPool kernelPhysical;
    // 用户物理地址池
    AddressPool userPhysical;
};
```

内存的申请

```
int MemoryManager::allocatePhysicalPages(enum AddressPoolType type, const int
count)
{
    int start = -1;
    if (type == AddressPoolType::KERNEL)//内核地址池
    {
        start = kernelPhysical.allocate(count);
    }
    else if (type == AddressPoolType::USER)//用户地址池
    {
        start = userPhysical.allocate(count);
    }
    return (start == -1) ? 0 : start;//返回分配的物理页内存地址
}
```

内存的释放

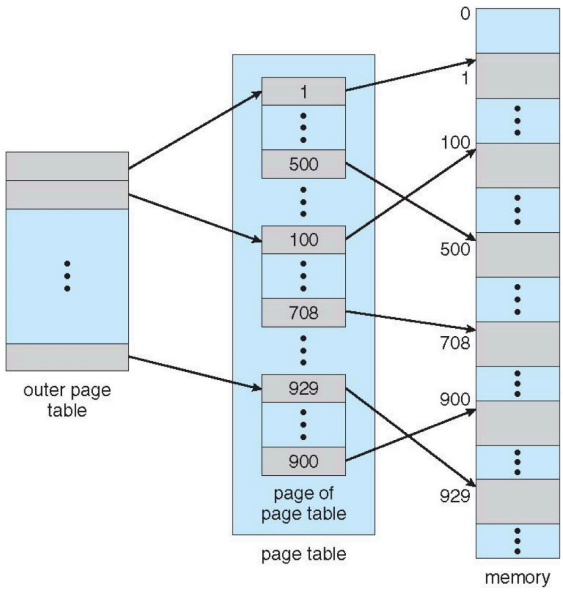
```
void MemoryManager::releasePhysicalPages(enum AddressPoolType type, const int
paddr, const int count)
{
    if (type == AddressPoolType::KERNEL)//内核地址池
    {
        kernelPhysical.release(paddr, count);
    }
    else if (type == AddressPoolType::USER)//用户地址池
    {
        userPhysical.release(paddr, count);
    }
}
```

二级页表introduction

一级页表的缺陷：可能一个进程所占的空间太大，一个进程的页表是连续存放的，当一个进程过大的时候，页表也会变得非常庞大

解决方法：将那个超大体积的页表再做一次分页处理，为页表创建页表，即页目录表，页目录表中的每一项被称为页目录项，页目录项的内容是页表的物理地址。通过页目录表来访问页表，然后通过页表访问物理页的方式被称为二级分页机制。

分层页表图示：



一个32位的虚拟地址被划分为以下三部分

- 31-22，共10位，是页目录项的序号，可以表示$2^{10} = 1024$个页目录项
- 21-12，共10位，是页表项的序号，可以表示$2^{10} = 1024$个页表项

- 11-0，共12位，是页内偏移，可以表示$2^{12} = 4KB$的物理页内的偏移地址

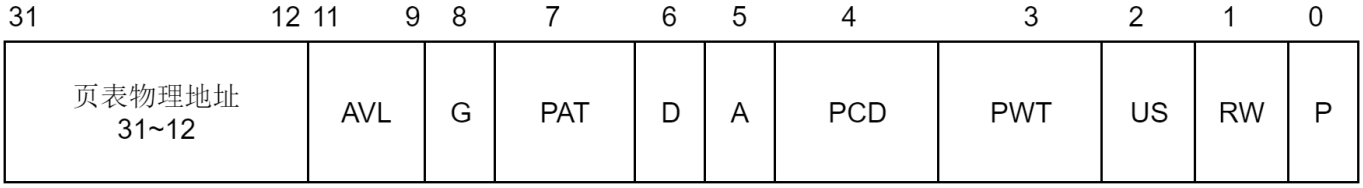
二级页表的虚拟地址到物理地址的转换关系:

- 给定一个虚拟地址，先取31-22位，其数值乘4后得到页目录表项在页目录表的偏移地址。这个偏移地址加上页目录表的物理地址后得到页目录项的物理地址。
- 取页目录项中的内容，得到页表的物理地址。页表的物理地址加上21-12位乘4的结果后，得到页表项的物理地址。
- 取页表项的内容，即物理页的物理地址，加上11-0位的内容后便得到实际的物理地址。

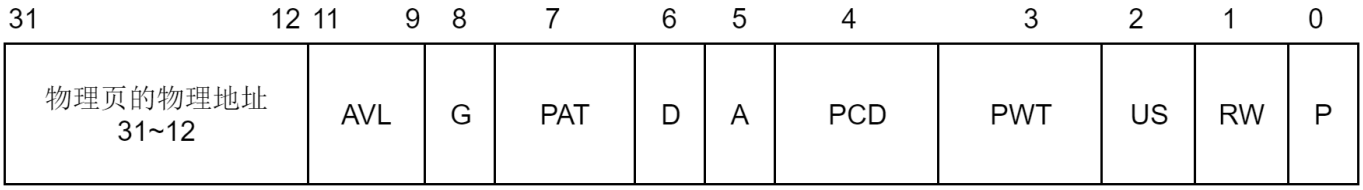
启动二级页表分页机制的流程

1、规划好页目录表和页表在内存中的位置并写入内容

页目录项与页表项的结构基本相同，对于页目录项，31~12为页表的物理地址位的高20位



对于页表项，31~12页为页的物理地址的高20位



> - **31-12位**是页表的物理地址位的高20位，因此页目录表的地址必须是4KB的整数倍。页目录表和页表实际上也是内存中的一个页，而内存被划分成了大小为4KB的页。自然地，这些物理页的起始就是4KB的整数倍。
> - **P位**是存在位，1表示存在，0表示不存在。
> - **RW位**，read/write。1表示可读写，0表示可读不可写。
> - **US位**，user/supervisor。若为1时，表示处于User级，任意级别（ 0、 1、 2、3）特权的程序都可以访问该页。若为0，表示处于 Supervisor 级，特权级别为3的程序不允许访问该页，该页只允许特权级别为0、1、2的程序可以访问。
> - **PWT位**，这里置0。PWT， Page-level Write-Through，意为页级通写位，也称页级写透位。若为 1 表示此项采用通写方式，表示该页不仅是普通内存，还是高速缓存。

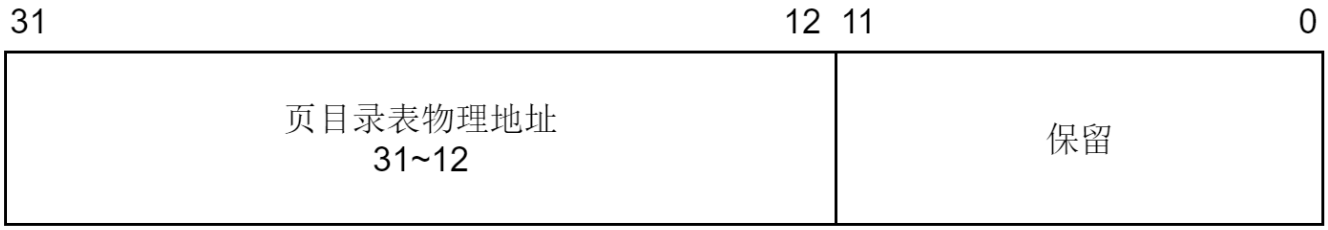
- **PCD位**，这里置0。PCD， Page-level Cache Disable，意为页级高速缓存禁止位。若为 1 表示该页启用高速缓存，为 0 表示禁止将该页缓存。
- **A位**，访问位。1表示被访问过，0表示未被访问，由CPU自动置位。
- **D位**，Dirty，意为脏页位。当CPU对一个页面执行写操作时，就会设置对应页表项的D位为1。此项 仅针对页表项有效，并不会修改页目录项中的D位。
- **G位**，这里置0，和TLB相关。
- **PAT**， 这里置0。Page Attribute Table，意为页属性表位，能够在页面一级的粒度上设置内存属性。

2、将页目录表的地址写入cr3

cr3寄存器保存的是页目录表的地址，使得CPU的MMU（内存管理单元）能够找到页目录表的地址，然后自动地将线性地址转换成物理地址

在建立页目录表和页表后，需要将页目录表地址放到CPU所约定的地方，即cr3，可以直接使用mov指令赋值

cr3又被称为页目录基址寄存器 PDBR，其结构如下：



3、将cr0的PG位置1

开启分页机制

在MomeryManager类中实现 `openPageMechanism()`

在开启分页机制前，需要建立内核所在地址的页目录表和页表，否则一旦开启分页机制，CPU就会出现寻址异常。由于内核很小，于是假设内核只会放在0~1MB的内存区域。

```
    // 页目录表指针
    int *directory = (int *)PAGE_DIRECTORY;
    //线性地址0~4MB对应的页表
    int *page = (int *)(PAGE_DIRECTORY + PAGE_SIZE);
    // 初始化页目录表
    memset(directory, 0, PAGE_SIZE);
    // 初始化线性地址0~4MB对应的页表
    memset(page, 0, PAGE_SIZE);
```

对于0~1MB的内存区域建立的是虚拟地址到物理地址的恒等映射，即虚拟地址和物理地址相同。此时需要设置相应的页目录项和页表项。

```
int address = 0;
    // 将线性地址0~1MB恒等映射到物理地址0~1MB
    for (int i = 0; i < 256; ++i)
    {
        // U/S = 1, R/W = 1, P = 1
        page[i] = address | 0x7;
        address += PAGE_SIZE;
    }
```

初始化页目录项：由于0~1MB的线性地址对应于第0个页目录项，于是用刚刚放入了256个页表项的页表作为第0个页目录项指向的页表。同样地，设置U/S，R/W和P位为1。

```
    // 初始化页目录项
    // 0~1MB
    directory[0] = ((int)page) | 0x07;
    // 3GB的内核空间
    directory[768] = directory[0];
    // 最后一个页目录项指向页目录表
    directory[1023] = ((int)directory) | 0x7;
```

将页目录表的地址放入cr3寄存器，然后将cr0的PG位置1，开启分页机制

```
    // 初始化cr3，cr0，开启分页机制
    asm_init_page_reg(directory);
```

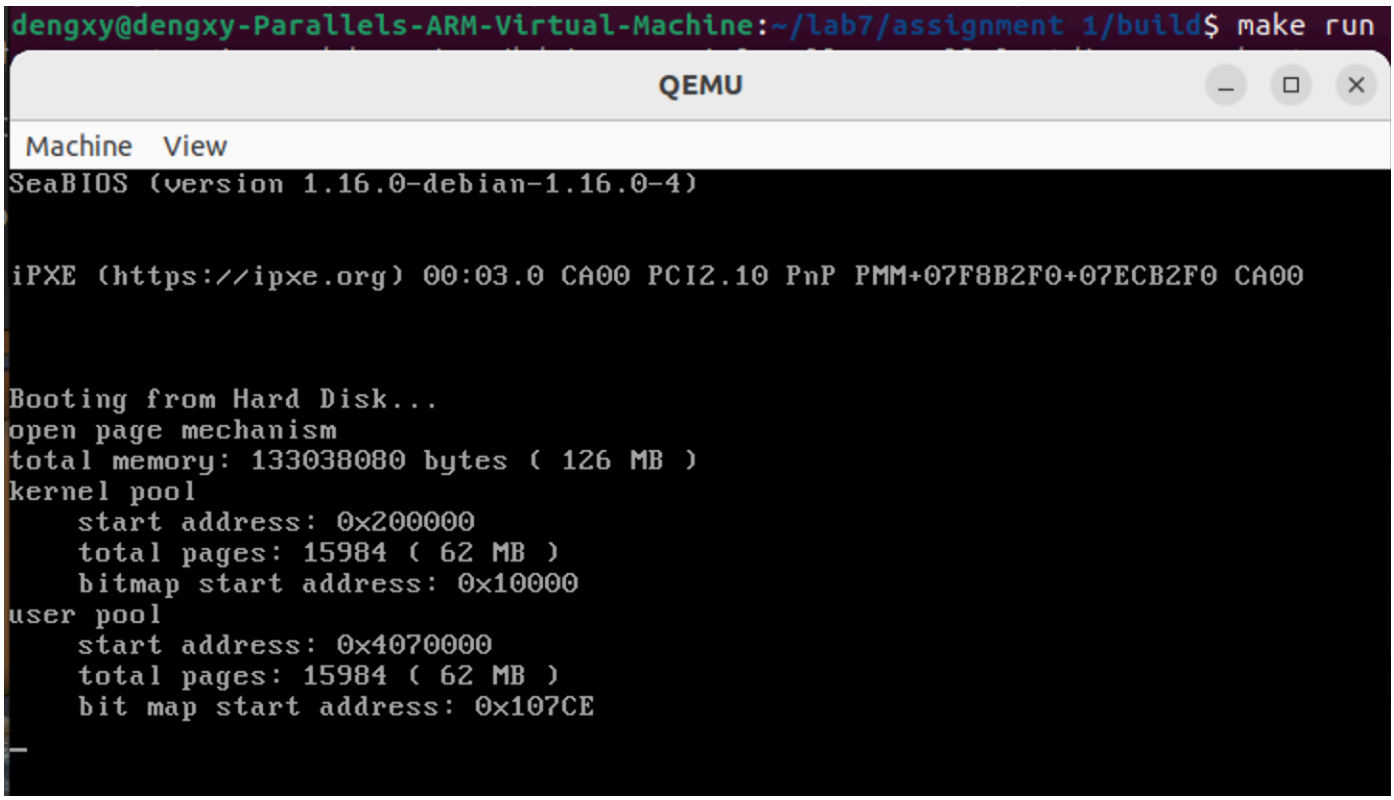
`asm_init_page_reg(directory)` 的实现

```
asm_init_page_reg:
    push ebp
    mov ebp, esp
    push eax
    mov eax, [ebp + 4 * 2]
    mov cr3, eax ; 放入页目录表地址
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax            ; 置PG=1，开启分页机制
    pop eax
    pop ebp
    ret
```

在 `setup.cpp` 中开启分页机制

```
    memoryManager.openPageMechanism();
    memoryManager.initialize();
```

运行结果如图：



Assignment 2

动态分区

如何满足空闲孔列表中大小为 n 的请求？

- 首次适应：分配足够大的第一个孔；first-fit
- 最佳匹配：分配足够大的最小孔；必须搜索整个列表，除非按大小排序；best-fit
  - 生成最小的剩余孔
- 最差适应：分配最大的孔；还必须搜索整个列表；
  - 产生最大的剩余孔

在速度和存储利用率方面，First fit和best fit优于最差fit

下面分别实现first-fit算法和best-fit算法

设置场景，初始有5个可分配的分区，大小分别为800KB，2400KB，2000KB，1200KB，800KB，有4个待分配的进程，大小分别为1000KB，600KB，1800KB，1600KB

first-fit

reference:https://www.geeksforgeeks.org/program-first-fit-algorithm-memory-management/

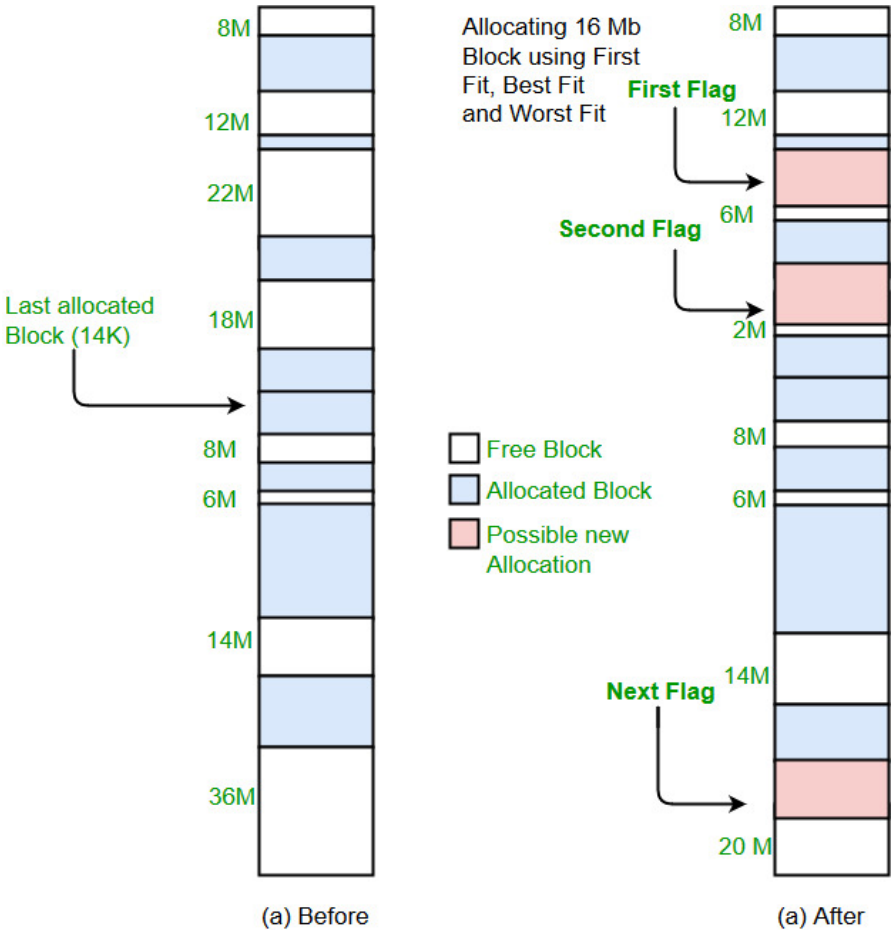
从主内存的顶部开始搜索第一个可分配的分区

搜索速度快，但是可能会导致部分进程没有分配到可用的空间，即使存在可分配的可能

> **Example :**
>
> ```
> Input : blockSize[]   = {100, 500, 200, 300, 600};
>         processSize[] = {212, 417, 112, 426};
> Output:
> Process No.    Process Size    Block no.
>    1              212           2
>    2              417           5
>    3              112           2
>    4              426        Not Allocated
> ```
>
> - Its advantage is that it is the fastest search as it searches only the first block

i.e. enough to assign a process.

- It may have problems of not allowing processes to take space even if it was possible to allocate. Consider the above example, process number 4 (of size 426) does not get memory. However it was possible to allocate memory if we had allocated using best fit allocation [block number 4 (of size 300) to process 1, block number 2 to process 2, block number 3 to process 3 and block number 5 to process 4].



实现步骤：

1、初始化输入内存块的大小和进程的大小

2、将所有内存块初始化为空闲状态

3、循环遍历，对于每个进程，检查当是否可以分配到当前块

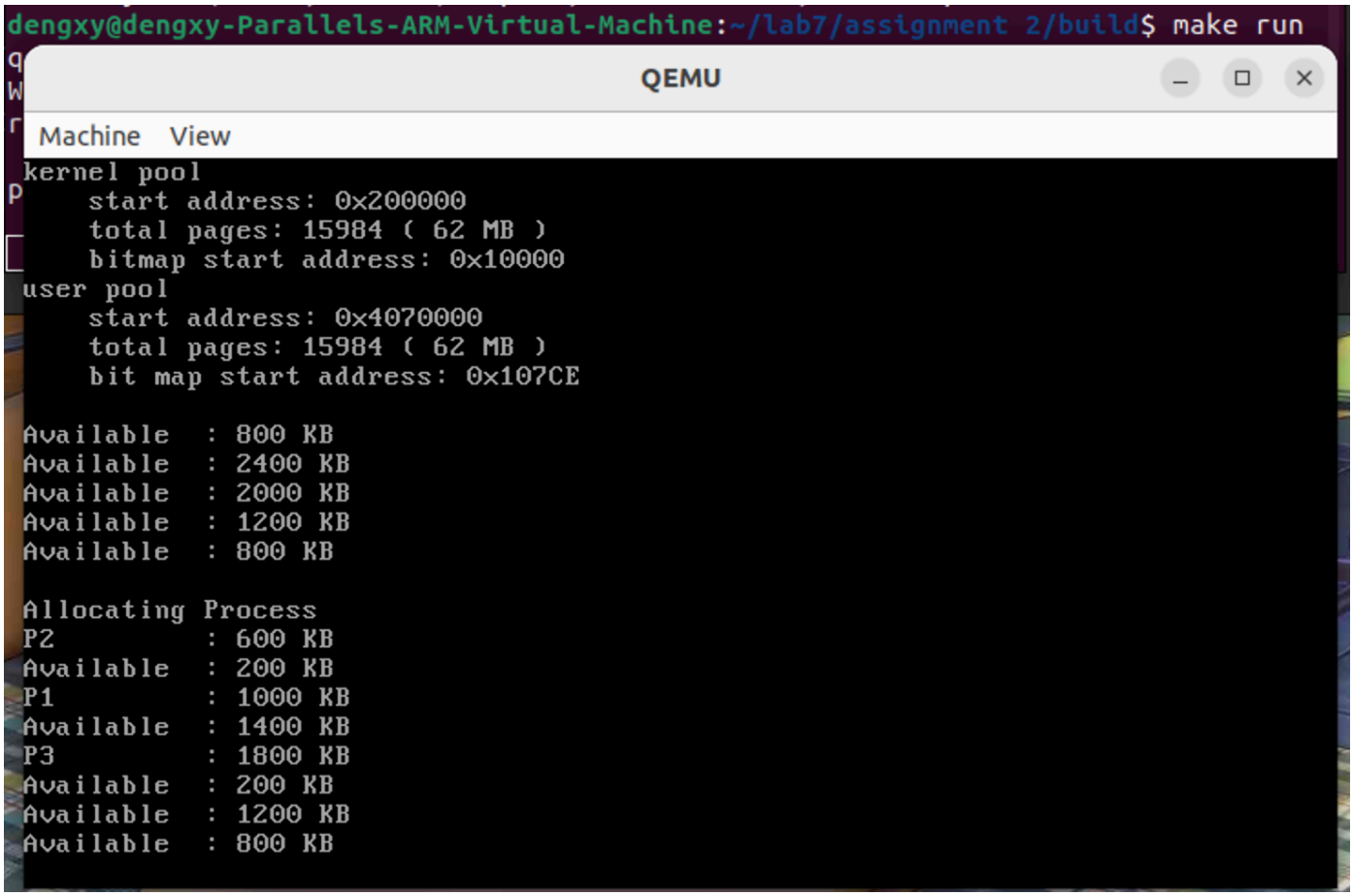
4、如果process_size<=block_size，则分配当前进程，并继续检查下一进程；否则，继续遍历其它的内存块

代码实现：在 `BitMap` 类中实现 `FirstFit` 函数

```
void BitMap::FirstFit(int ProcessSize[4])
{
```

```
    // 初始化4个进程的名称
    MemoryType ProcessList[4] = {P1, P2, P3, P4};
    for(int i = 0; i < 4; i++)
    {
      // 找到可分配的内存块的首地址
      int start = allocate(ProcessSize[i]/4);
      if(start != -1)//可分配
      {
        for(int j = 0; j < ProcessSize[i]/4; j++)
        {
          set(start+j, 1);//设置该分区已被使用
          MemoryTypeList[start + j] = ProcessList[i];
        }
      }
    }
}
```

实验结果如下（为方便起见，所有数据均取整百）：



实验结果解释：

- P1为1000KB，而第一个内存块的大小为800KB，不可分配，于是继续搜索，第二个内存块的大小为2400KB，可分配，分配后，available变成2400KB-100KB=1400KB
- P2为600KB，显然，第一个内存块（800KB）可以满足，分配后，available变成200KB

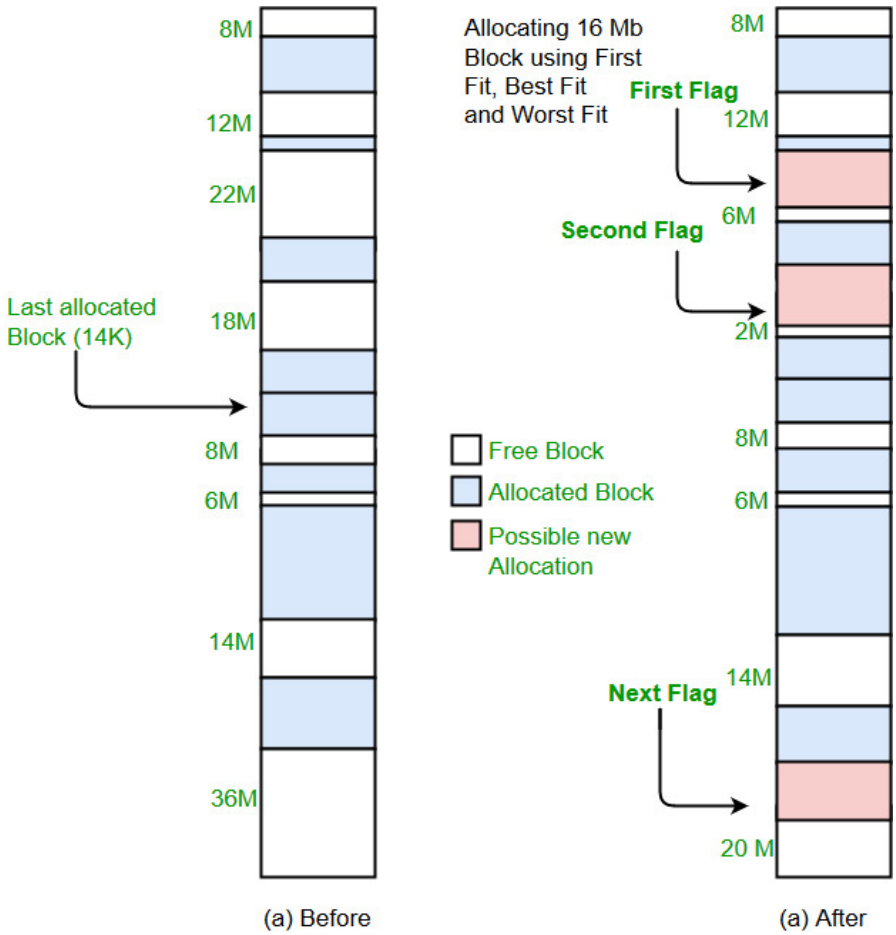
- P3为1800KB，搜索到第三个内存块（2000KB）可以满足，分配后，available变成2000KB-1800KB=200KB
- 而此处，所有available大小分别为：200KB，1400KB，200KB，1200KB，800KB，显然无法满足P4（1600KB）的需求，因此P4无法得到分配
- 然而，available的大小其实是足够满足所有4个进程的，因此，我们需要优化算法，采用best-fit

best-fit

reference:https://www.geeksforgeeks.org/program-best-fit-algorithm-memory-management/

best-fit将进程分配到空闲可用分区中足够但最小的分区，即需要遍历所有空闲可用分区

```
Input : blockSize[]   = {100, 500, 200, 300, 600};
        processSize[] = {212, 417, 112, 426};
Output:
Process No.     Process Size    Block no.
 1              212             4
 2              417             2
 3              112             3
 4              426             5
```



实现步骤：

1、初始化内存块和进程大小

2、将所有内存块初始化为空闲状态

3、循环遍历，对于每个进程，找到可以分配给当前进程的最小内存块，即找到`min(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current]`，将其分配给当前进程；否则，继续遍历其它进程

代码实现：

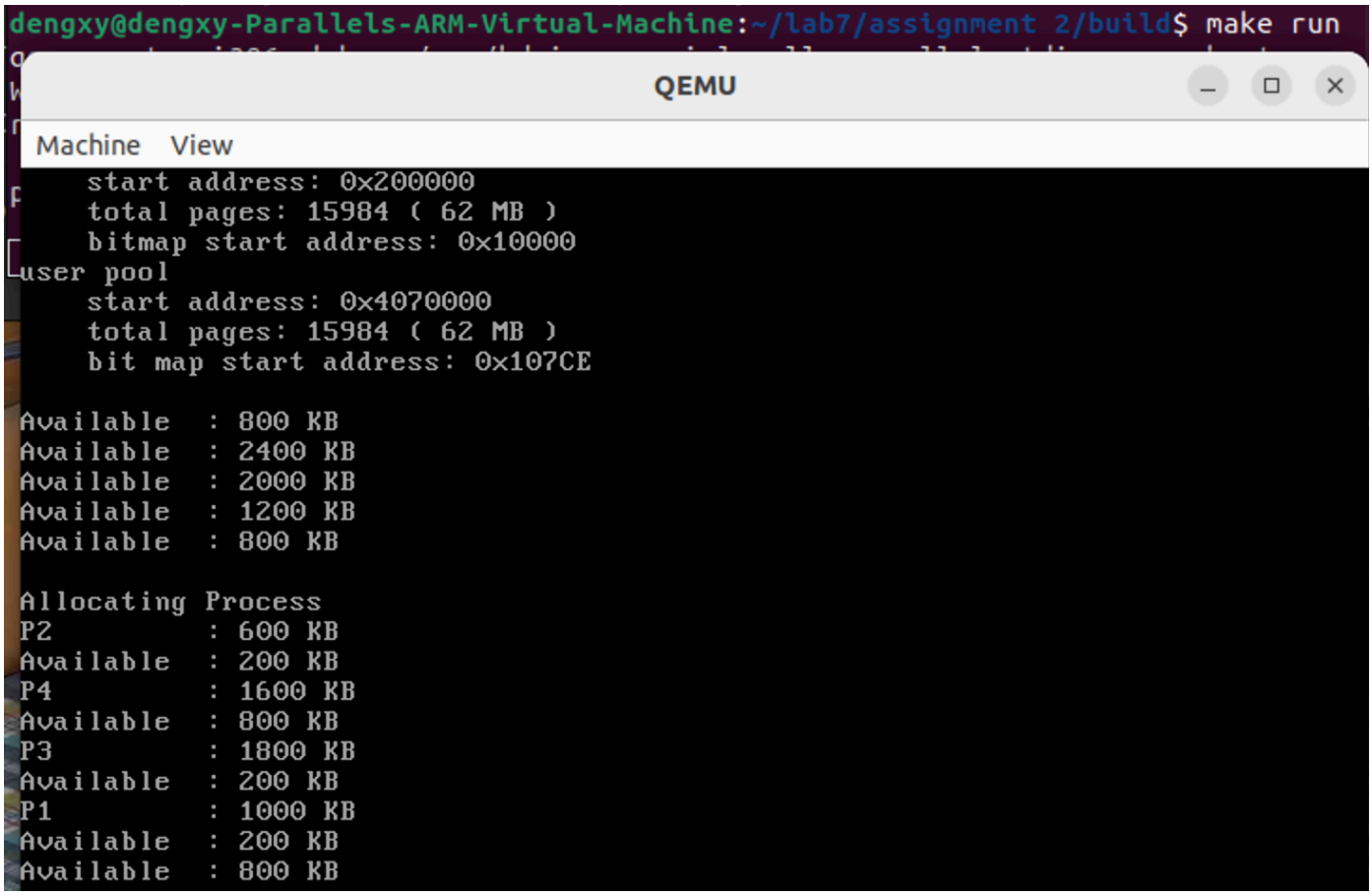
```
void BitMap::BestFit(int ProcessSize[4])
{
  //初始化4个进程的名称
  MemoryType ProcessList[4] = {P1, P2, P3, P4};
  for(int i = 0; i < 4; i++)
  {
    //找到可分配的内存块的首地址
    int start = BestFitAddress(ProcessSize[i]/4);
    if(start != -1)//可分配
    {
      for(int j = 0; j < ProcessSize[i]/4; j++)
      {
        set(start+j, 1);//设置改分区已被使用
        MemoryTypeList[start + j] = ProcessList[i];
      }
    }
  }
}
```

关于步骤3，如何找到可以分配给当前进程的最小内存块的代码如下

```
int BitMap::BestFitAddress(int ProcessPages)
{
  int BestFitStartAddress;//best-fit的首地址
  int TotalUnallocated = UserPages;//总的未分配的地址数
  int StartAddress;//当前连续未分配区域的起始地址
  int unallocated = 0;//当前连续未分配区域的大小
  int Flag = 0;//标记是否已经找到起始地址
  for(int i = 0; i < UserPages; i++)
  {
    if(get(i) == 0 && !Flag) //如果当前位未被使用且之前未找到起始地址
```

```
    {
      Flag = 1;//标记已经找到起始地址
      StartAddress = i;//记录当前连续未分配区域的起始地址
      unallocated++;//记录当前连续未分配区域的大小
    }
    if(get(i) == 0 && Flag)//如果当前位未被使用且之前已经找到起始地址
      unallocated++;//记录当前连续未分配区域的大小
    //如果当前位已被使用或者已经到达末尾，且之前已经找到起始地址
    if((get(i) == 1  || i == 15983) && Flag)
    {
      //如果当前连续未分配区域的大小符合要求
      if(ProcessPages < unallocated && unallocated < TotalUnallocated)
      {
        TotalUnallocated = unallocated;//更新最小的未分配地址数
        BestFitStartAddress = StartAddress; //更新最佳起始地址
      }
      unallocated = 0;//重置当前连续未分配区域的大小
      Flag = 0;//重置标记
    }
    else continue;// 如果当前位已被使用且之前未找到起始地址，则继续向后查找
  }
}
```

实验结果如下：



实验结果解释：

- P1为1000KB，第4个可分配分区最优，为1200KB，分配完后available变为1200KB-1000KB=200KB
- P2为600KB，第1个可分配分区最优，为800KB，分配完成后available变为800KB-600KB=200KB
- P3为1800KB，第3个可分配分区最优，为2000KB，分配完成后available变为2000KB-1800KB=200KB
- P4为1600KB，第2个可分配分区最优，为2400KB，分配完成后available变为2400KB-1600KB=800KB
- 所有4个进程都能够成功分配，但是每次分配都需要遍历全部可分配分区，从时间上不如first-fit，但实际分配的结果要好于best-fit

Assignment 3

Page Fault: 当运行中的程序访问一个映射到虚拟地址空间但没有加载到物理内存的内存页时，就会发生页故障。由于实际的物理内存要比虚拟内存小得多，因此会发生页故障。在发生页面故障的情况下，操作系统可能不得不用新需要的页面替换现有的页面之一。不同的页面替换算法提出了不同的方法来决定替换哪个页面。所有算法的目标都是为了减少页面故障的数量。

使用FIFO算法实现页面置换

FIFO页面置换算法是一种简单而直观的页面置换算法，但是只考虑页面进入内存的时间先后顺序，而不考虑页面的访问频率和重要性导致算法效率较低，不能保证最优页面置换，导致大量的缺页错误。

实现步骤：

遍历页面

1、如果当前页面不在队列中

- 如果置换队列未满，则将页面加入队列中，直到队列已满
- 否则，删除队列中的第一页，并将其替换为当前页

2、如果当前页面在队列中，则不需要执行任何过程

```
void MemoryManager::FIFO(const int referenceBit)
{
    int vaddr = referenceBit * PAGE_SIZE;
    //当前虚拟页表中对应页面没有referenceBit，即出现缺页错误
    if(kernelVirtual.get(referenceBit) == 0)
```

```
{
    //FIFO页面置换队列是已满，将队列前面的页面替换出去
    if(FIFO_Queue.isFull())
    {
      //从FIFO队列中出队第一个项
      int *addressPair = FIFO_Queue.dequeue();
      int VirAddr = addressPair[0];
      int PhyAddr = addressPair[1];
      int PageReplace = VirAddr / PAGE_SIZE;
      kernelVirtual.set(PageReplace, 0);
      //获取被替换页面的页表项（PTE）
      int *pte = (int *)toPTE(VirAddr);
      *pte = 0;
      kernelVirtual.set(referenceBit, 1);
      int flag = connectPhysicalVirtualPage(vaddr, PhyAddr);
      if(!flag)
      {
          printf("ERROR\n");
      }
      //计算被替换和被添加页面的虚拟页号
      int PageOut = VirAddr / PAGE_SIZE;
      int PageIn = referenceBit;
      printf("Swap page %d with page %d \n",PageOut, PageIn);
      int AddrPair[] = {vaddr, PhyAddr};
      FIFO_Queue.enqueue(AddrPair);
    }
    //FIFO队列未满，则将页面分配一个新的物理页面
    else
    {
      int MapPhyAddr = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
      if(!MapPhyAddr)
      {
          printf("ERROR\n");
          return;
      }
      kernelVirtual.set(referenceBit, 1);
      int flag = connectPhysicalVirtualPage(vaddr, MapPhyAddr);
      if(!flag)
      {
          printf("ERROR\n");
      }
      int virtualPage = referenceBit;
      printf("Page Fault, add page %d\n", virtualPage);
      //创建一个新的FIFO队列项，并将其加入到队列的末尾
```

```
            int newAddressPair[] = {vaddr, MapPhyAddr};
            FIFO_Queue.enqueue(newAddressPair);
        }
    }
    else
    {
        printf("SUCCESS: page %d\n", referenceBit);
    }
}
```

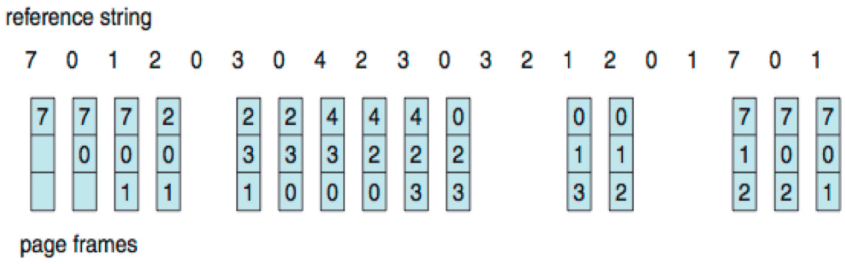
在 `setup.cpp` 中修改 `first_thread`，以实现上述例子的全过程

```
void first_thread(void *arg)
{
    memoryManager.initQueue();
    int FIFO_Pages[20] = {7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1};
    memoryManager.FIFO_PageReplace(FIFO_Pages, 20);
    asm_halt();
}
```

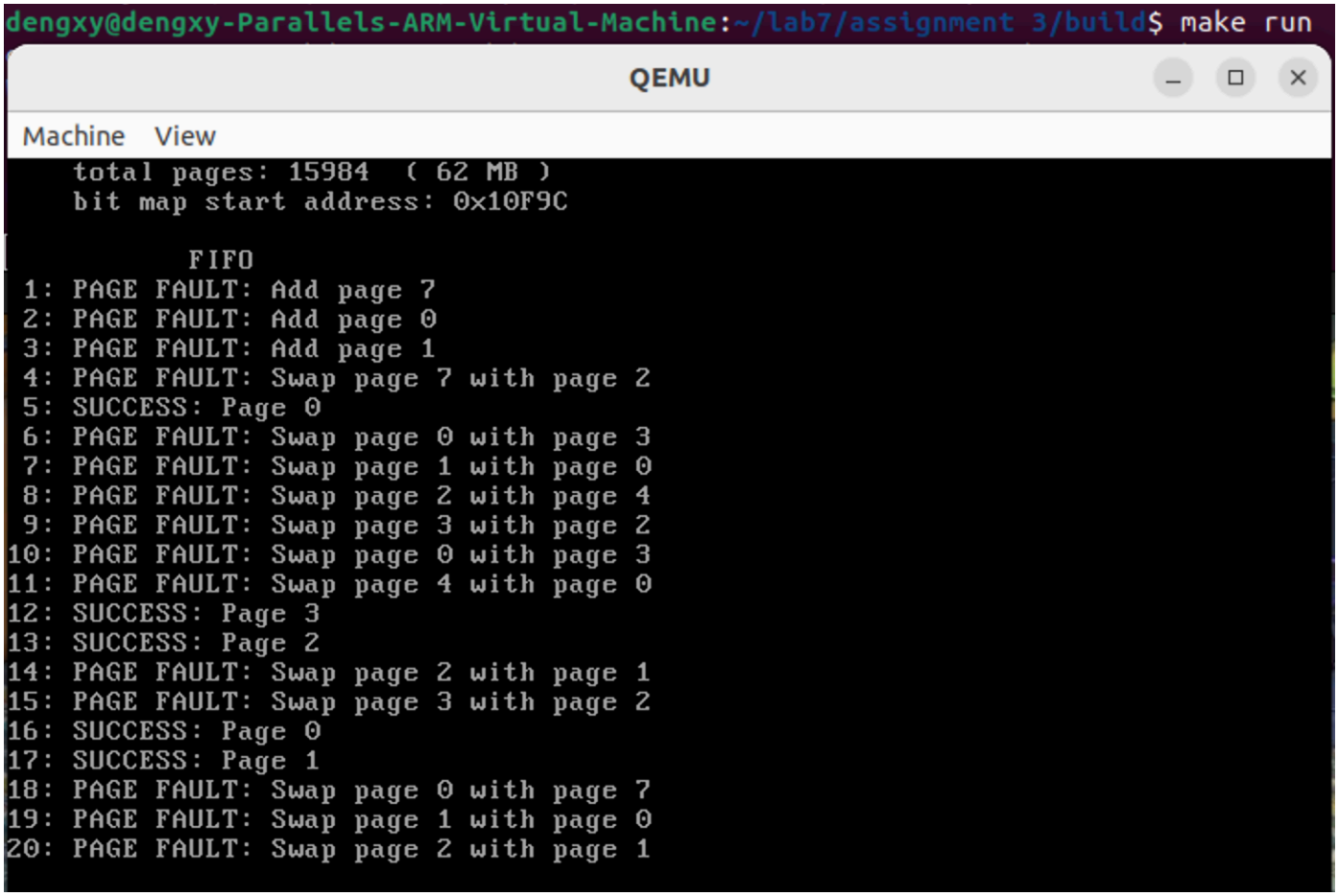
实验结果：出现15个Page Fault

先进先出FIFO算法

- 引用字符串：7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- 3帧（每个进程一次可在内存中存储3页）



15个缺页错误



Assignment 4

虚拟页内存分配

1、从虚拟地址池中分配若干连续的虚拟页

```
int allocateVirtualPages(enum AddressPoolType type, const int count)
{
    int start = -1;
    if (type == AddressPoolType::KERNEL)
    {
        start = kernelVrirtual.allocate(count);
    }
    return (start == -1) ? 0 : start;
}
```

2、对每一个虚拟页，从物理地址池中分配1页

```
int MemoryManager::allocatePhysicalPages(enum AddressPoolType type, const int
count)
{
    int start = -1;
    if (type == AddressPoolType::KERNEL)
    {
        start = kernelPhysical.allocate(count);
    }
    else if (type == AddressPoolType::USER)
    {
        start = userPhysical.allocate(count);
    }
    return (start == -1) ? 0 : start;
}
```

3、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

在二级分页机制下，虚拟地址变换到物理地址的过程如下

- CPU先取虚拟地址的31-22位 `virtual[31:22]` ，在 `cr3` 寄存器中找到页目录表的物理地址，然后根据页目录表的物理地址在页目录表中找到序号为 `virtual[31:22]` 的页目录项，读取页目录项中的页表的物理地址。
- CPU再取虚拟地址的21-12位 `virtual[21:12]` ，根据第一步取出的页表的物理地址，在页表中找到序号为 `virtual[21:12]` 的页表项，读取页表项中的物理页的物理地址 `physical` 。
- 用物理页的物理地址的31-12位 `physical[31:12]` 替换虚拟地址的31-12位 `virtual[31:12]` 得到的结果即为虚拟地址对应的物理地址。

构造页目录项和页表项的虚拟地址

```
// 最后一个页目录项指向页目录表
directory[1023] = ((int)directory) | 0x7;
```

构建页目录项和页表项

```
int toPDE(const int virtualAddress)
{
    return (0xfffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
}
int toPTE(const int virtualAddress)
{
    return (0xffc00000 + ((virtualAddress & 0xffc00000) >> 10) +
(((virtualAddress & 0x003ff000) >> 12) * 4));
}
```

建立虚拟页到物理页的映射关系

- 检查pde中是否有对应的页表：如果没有，则先分配一个物理页，然后初始化新分配的物理页并将其地址写入pde，作为pde指向的页表。
- 在pde对应的物理页存在的前提下，将之前为虚拟页分配的物理页地址写入pte。

```
bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const
int physicalPageAddress)
{
    // 计算虚拟地址对应的页目录项和页表项
    int *pde = (int *)toPDE(virtualAddress);
    int *pte = (int *)toPTE(virtualAddress);
    // 页目录项无对应的页表，先分配一个页表
    if(!(*pde & 0x00000001))
    {
        // 从内核物理地址空间中分配一个页表
        int page = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
        if (!page)
            return false;
        // 使页目录项指向页表
        *pde = page | 0x7;
        // 初始化页表
        char *pagePtr = (char *)(((int)pte) & 0xfffff000);
        memset(pagePtr, 0, PAGE_SIZE);
    }
    // 使页表项指向物理页
    *pte = physicalPageAddress | 0x7;
    return true;
}
```

结合以上3个步骤，虚拟页内存分配的过程为

```
int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
{
    // 第一步：从虚拟地址池中分配若干虚拟页
    int virtualAddress = allocateVirtualPages(type, count);
    if (!virtualAddress)
    {
        return 0;
    }
    bool flag;
    int physicalPageAddress;
    int vaddress = virtualAddress;
```

```
    // 依次为每一个虚拟页指定物理页
    for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
    {
        flag = false;
        // 第二步：从物理地址池中分配一个物理页
        physicalPageAddress = allocatePhysicalPages(type, 1);
        if (physicalPageAddress)
        {
            //printf("allocate physical page 0x%x\n", physicalPageAddress);
         // 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。
            flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
        }
        else
        {
            flag = false;
        }
        // 分配失败，释放前面已经分配的虚拟页和物理页表
        if (!flag)
        {
            // 前i个页表已经指定了物理页
            releasePages(type, virtualAddress, i);
            // 剩余的页表未指定物理页
           releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
            return 0;
        }
    }
    return virtualAddress;
}
```

虚拟页内存释放

1、对每一个虚拟页，释放为其分配的物理页

物理地址池存放的是物理地址，为释放物理页，要找到虚拟页对应的物理页的物理地址

```
int MemoryManager::vaddr2paddr(int vaddr)
{
    int *pte = (int *)toPTE(vaddr);
    int page = (*pte) & 0xfffff000;
    int offset = vaddr & 0xfff;
    return (page + offset);
}
```

2、释放虚拟页

```
void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int
vaddr, const int count)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelVirtual.release(vaddr, count);
    }
}
```

综合以上两个步骤，释放虚拟页内存

```
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);
}
```

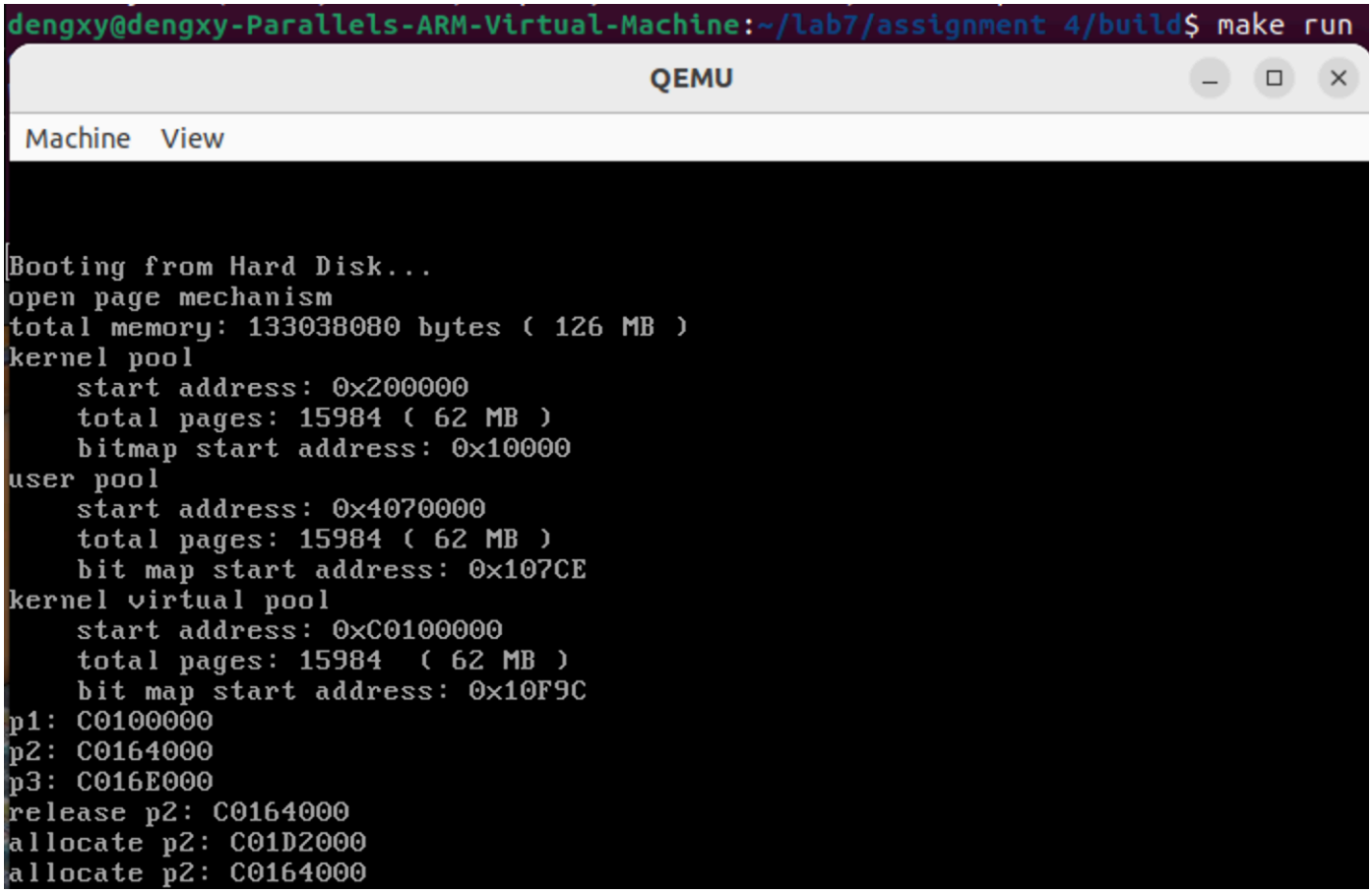
测试虚拟页内存管理的实现

修改 `first_thread` 函数

```
void first_thread(void *arg)
{
   char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
   char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
   char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    printf("p1: %x\np2: %x\np3: %x\n", p1, p2, p3);
    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
    printf("release p2: %x\n", p2);
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
```

```
    printf("allocate p2: %x\n", p2);
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    printf("allocate p2: %x\n", p2);
    asm_halt();
}
```

实验结果如下：



实验结果分析：

- p1分配的内存空间为从C0100000到C0164000，分配的空间和页数分布为
-

$$
\begin{aligned}
C0164000 - C0100000 = 64000_{(16)} = 409600bytes \\
PageNum = \frac{409600bytes}{4096bytes} = 100
\end{aligned} \tag{1}
$$

- p2分配的内存空间从C0164000到C016E000，分配的空间和页数分别为
-

$$
\begin{aligned}
C016E000 - C0164000 = B000_{(16)} = 40960bytes \\
PageNum = \frac{40960bytes}{4096bytes} = 10
\end{aligned} \tag{2}
$$

- 按照预期，p3分配的内存空间应为409600bytes，分配的页数应该为100，因此，p3的内存空间为从C016E000到C01D2000，该结果与从p2开始分配100个页面后得到的结果相同，符合预期
- 而将p2的10个虚拟页内存释放后再重新分配10个虚拟页内存，其开始的物理地址与p2释放前的物理地址相同，符合预期

3、关键代码

Assignment 2

first-fit

```
void BitMap::FirstFit(int ProcessSize[4])
{
  // 初始化4个进程的名称
  MemoryType ProcessList[4] = {P1, P2, P3, P4};
  for(int i = 0; i < 4; i++)
  {
    // 找到可分配的内存块的首地址
    int start = allocate(ProcessSize[i]/4);
    if(start != -1)//可分配
    {
      for(int j = 0; j < ProcessSize[i]/4; j++)
      {
        set(start+j, 1);//设置该分区已被使用
        MemoryTypeList[start + j] = ProcessList[i];
      }
    }
  }
```

best-fit

```
void BitMap::BestFit(int ProcessSize[4])
{
  //初始化4个进程的名称
  MemoryType ProcessList[4] = {P1, P2, P3, P4};
  for(int i = 0; i < 4; i++)
  {
    //找到可分配的内存块的首地址
    int start = BestFitAddress(ProcessSize[i]/4);
    if(start != -1)//可分配
```

```
    {
      for(int j = 0; j < ProcessSize[i]/4; j++)
      {
        set(start+j, 1);//设置改分区已被使用
        MemoryTypeList[start + j] = ProcessList[i];
      }
    }
  }
}
int BitMap::BestFitAddress(int ProcessPages)
{
  int BestFitStartAddress;//best-fit的首地址
  int TotalUnallocated = UserPages;//总的未分配的地址数
  int StartAddress;//当前连续未分配区域的起始地址
  int unallocated = 0;//当前连续未分配区域的大小
  int Flag = 0;//标记是否已经找到起始地址
  for(int i = 0; i < UserPages; i++)
  {
    if(get(i) == 0 && !Flag) //如果当前位未被使用且之前未找到起始地址
    {
      Flag = 1;//标记已经找到起始地址
      StartAddress = i;//记录当前连续未分配区域的起始地址
      unallocated++;//记录当前连续未分配区域的大小
    }
    if(get(i) == 0 && Flag)//如果当前位未被使用且之前已经找到起始地址
      unallocated++;//记录当前连续未分配区域的大小
    //如果当前位已被使用或者已经到达末尾，且之前已经找到起始地址
    if((get(i) == 1  || i == 15983) && Flag)
    {
      //如果当前连续未分配区域的大小符合要求
      if(ProcessPages < unallocated && unallocated < TotalUnallocated)
      {
        TotalUnallocated = unallocated;//更新最小的未分配地址数
        BestFitStartAddress = StartAddress; //更新最佳起始地址
      }
      unallocated = 0;//重置当前连续未分配区域的大小
      Flag = 0;//重置标记
    }
    else continue;// 如果当前位已被使用且之前未找到起始地址，则继续向后查找
  }
}
```

Assignment 3

FIFO算法的实现

```
void MemoryManager::FIFO(const int referenceBit)
{
    int vaddr = referenceBit * PAGE_SIZE;
    //当前虚拟页表中对应页面没有referenceBit，即出现缺页错误
    if(kernelVirtual.get(referenceBit) == 0)
    {
        //FIFO页面置换队列是已满，将队列前面的页面替换出去
        if(FIFO_Queue.isFull())
        {
          //从FIFO队列中出队第一个项
          int *addressPair = FIFO_Queue.dequeue();
          int VirAddr = addressPair[0];
          int PhyAddr = addressPair[1];
          int PageReplace = VirAddr / PAGE_SIZE;
          kernelVirtual.set(PageReplace, 0);
          //获取被替换页面的页表项（PTE）
          int *pte = (int *)toPTE(VirAddr);
          *pte = 0;
          kernelVirtual.set(referenceBit, 1);
          int flag = connectPhysicalVirtualPage(vaddr, PhyAddr);
          if(!flag)
          {
              printf("ERROR\n");
          }
          //计算被替换和被添加页面的虚拟页号
          int PageOut = VirAddr / PAGE_SIZE;
          int PageIn = referenceBit;
          printf("Swap page %d with page %d \n",PageOut, PageIn);
          int AddrPair[] = {vaddr, PhyAddr};
          FIFO_Queue.enqueue(AddrPair);
        }
        //FIFO队列未满，则将页面分配一个新的物理页面
        else
        {
          int MapPhyAddr = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
          if(!MapPhyAddr)
          {
              printf("ERROR\n");
              return;
          }
```

```
            kernelVirtual.set(referenceBit, 1);
            int flag = connectPhysicalVirtualPage(vaddr, MapPhyAddr);
            if(!flag)
            {
                printf("ERROR\n");
            }
            int virtualPage = referenceBit;
            printf("Page Fault, add page %d\n", virtualPage);
            //创建一个新的FIFO队列项，并将其加入到队列的末尾
            int newAddressPair[] = {vaddr, MapPhyAddr};
            FIFO_Queue.enqueue(newAddressPair);
        }
    }
    else
    {
        printf("SUCCESS: page %d\n", referenceBit);
    }
}
```

4、总结

1. 实现了二级分页机制和内存管理，深入地了解操作系统的内存管理机制。
2. 不同的动态分区算法（first-fit、best-fit等）有不同的优缺点，需要根据实际情况选择合适的算法。
3. 页面置换算法是操作系统中虚拟内存管理的一个重要组成部分，选择合适的页面置换算法（FIFO、LRU等）可以缓解内存不足的问题。
4. 与传统的物理内存管理相比较，虚拟内存管理可以更加高效地管理内存。同时，需要注意处理虚拟地址和物理地址之间的映射关系，处理好虚拟页内存的分配和释放的过程。