**同济大学计算机系**

**操作系统课程实验报告**

****

**学 号 2251557**

**姓 名 代文波**

**专 业 计算机科学与技术**

**授课老师 方钰**

**实验六：去除UNIX V6++的相对虚实地址映射表**

**一 实验目的**

本次实验旨在尝试去除UNIX V6++的相对虚实地址映射表，并保证其页表系统仍能正常工作。在此过程中，读者将进一步加深对页表系统的理解，及深刻体会内存管理在整个进程生命周期管理中的重要性。同时，实践的过程也提供了一个梳理UNIX V6++的代码的好机会。

**二 实验设备及工具**

已配置好UNIX V6++运行和调试环境的PC机一台。

**三 预备知识**

（1）UNIX V6++完整进程图象的构成。

（2）UNIX V6++如何利用相对虚实地址映射表和页表完成进程的地址变换

（3）UNIX V6++进程的生命周期中所有和进程管理相关的操作。

**四 实验内容**

**4.1. 实验准备**

（1）做好备份

（2）打开MemoryDescriptor.h 和 MemoryDescriptor.cpp 文件，仔细阅读其中关于MemoryDescriptor的数据成员与方法的描述和定义

**4.2 修改页表构建过程**

MemoryDescriptor 类中，利用相对虚实地址映射表构建物理页表的函数是：



代码中涉及使用相对虚实地址映射表的部分需要全部修改，主要包括：

（1）利用相对虚实地址映射表中的R/W位判断代码段和数据段的起始位置；

（2）利用相对虚实地址映射表中的基地址项和p\_addr，x\_caddr生成页表的每一条记录。

**4.2.1 NMapToPageTable函数的算法说明**：

（1）获取进程代码段、数据段、堆栈段所占物理空间大小

在MemoryDescriptor类中，记录了进程代码段、数据段、堆栈段的逻辑起始地址和所占逻辑空间的大小。因为物理空间大小和逻辑空间大小相同，所以可以直接从这里获得对应部分所占物理空间的大小。

（2）获取进程代码段、数据段、堆栈段的物理起始地址

① 代码段：通过u\_procp 找到进程的Proc结构 ，然后从Proc结构中的p\_textp 找到进程对应的Text结构，最后从Text结构的x\_caddr 找到代码段的起始物理地址。

② 数据段：通过u\_procp 找到进程的Proc结构 ，然后从Proc结构中的p\_addr 找到进程对应的 PPDA区，PPDA区后面紧跟着数据段。又PPDA区占4KB，所以p\_addr +4KB就是数据段的地址。

③ 堆栈段：堆栈段在进程图像中是紧跟着数据段的，所以堆栈段的物理地址是数据段的物理地址加数据段的所占物理空间的大小。

（3）计算进程代码段、数据段、堆栈段所占用的页框数（向上取整）

（4）遍历进程用户页表中的每一项，进行填充

① 首先，把每一项的存在标志位m\_Present置0

② 因为第0张页表以及第1张页表的第一项均预留给编译器，所以从第1张页表的第1项开始填充

③ 先填充代码段，将存在标志位m\_Present置1，读写标志置为只读（不可写），并填充页基地址。

④ 接着填充数据段，将存在标志位m\_Present置1，读写标志置为可读可写，并填充页基地址。

⑤ 最后填充堆栈段，将存在标志位m\_Present置1，读写标志置为可读可写，并填充页基地址。

（5）剩下部分与MapToPageTable一致，先把第一个页表的第一项存在标志位m\_Present置1，读写标志置为可读可写，并填充页基地址为0，之后刷新页目录使之生效。

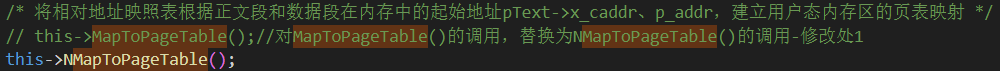
**4.2.2 NMapToPageTable函数的具体代码（完整代码加注释）**

|  |
| --- |
| **void** MemoryDescriptor::NMapToPageTable()  {      User& u = Kernel::Instance().GetUser();        PageTable\* pUserPageTable = Machine::Instance().GetUserPageTableArray();      //获取代码段大小      unsigned **int** text\_size = **this**->GetTextSize();      //获取数据段大小      unsigned **int** data\_size = **this**->GetDataSize();      //获取堆栈段大小      unsigned **int** stack\_size = **this**->GetStackSize();        //获取代码段起始地址（物理地址）      unsigned **int** textAddress = 0;  **if** ( u.u\_procp->p\_textp != NULL )      {          textAddress = u.u\_procp->p\_textp->x\_caddr;      }      //获取数据段起始地址（物理地址）      unsigned **int** dataAddress = u.u\_procp->p\_addr + PageManager::PAGE\_SIZE;      //获取堆栈段起始地址（物理地址）      unsigned **int** stackAddress = dataAddress + data\_size;        //代码段所占页框数      unsigned **int** text\_page = (text\_size + (PageManager::PAGE\_SIZE - 1)) / PageManager::PAGE\_SIZE;      //数据段所占页框数      unsigned **int** data\_page = (data\_size + (PageManager::PAGE\_SIZE - 1)) / PageManager::PAGE\_SIZE;      //堆栈段所占页框数      unsigned **int** stack\_page = (stack\_size + (PageManager::PAGE\_SIZE - 1)) / PageManager::PAGE\_SIZE;    **for** (unsigned **int** i = 0; i < Machine::USER\_PAGE\_TABLE\_CNT; i++)      {  **for** ( unsigned **int** j = 0; j < PageTable::ENTRY\_CNT\_PER\_PAGETABLE; j++ )          {              pUserPageTable[i].m\_Entrys[j].m\_Present = 0;   //先清0              //下面是修改后的内容：  **if**( i == 1 )//第一个用户页表不用管，预留给编译器了，我们只需要修改第二个页表就好              {                  //代码段 存在&不可写&地址转换  **if**(j >= 1 && j <=text\_page)                  {                      pUserPageTable[i].m\_Entrys[j].m\_Present = 1;                      pUserPageTable[i].m\_Entrys[j].m\_ReadWriter = 0;                      pUserPageTable[i].m\_Entrys[j].m\_PageBaseAddress = j - 1 + ( textAddress>>12 );                  }                  //数据段 存在&可写&地址转换  **else** **if**(j > text\_page && j <= text\_page + data\_page )                  {                      pUserPageTable[i].m\_Entrys[j].m\_Present = 1;                      pUserPageTable[i].m\_Entrys[j].m\_ReadWriter = 1;                      pUserPageTable[i].m\_Entrys[j].m\_PageBaseAddress = j - 1 - text\_page + ( dataAddress>>12 );                  }                  //堆栈段 存在&可写&地址转换  **else** **if**(j >= PageTable::ENTRY\_CNT\_PER\_PAGETABLE - stack\_page)                  {                      pUserPageTable[i].m\_Entrys[j].m\_Present = 1;                      pUserPageTable[i].m\_Entrys[j].m\_ReadWriter = 1;                      pUserPageTable[i].m\_Entrys[j].m\_PageBaseAddress = j - ( PageTable::ENTRY\_CNT\_PER\_PAGETABLE - stack\_page ) + (stackAddress>>12);                  }              }          }      }        pUserPageTable[0].m\_Entrys[0].m\_Present = 1;      pUserPageTable[0].m\_Entrys[0].m\_ReadWriter = 1;      pUserPageTable[0].m\_Entrys[0].m\_PageBaseAddress = 0;        FlushPageDirectory();  } |

**4.2.3 将所有对MapToPageTable()的调用替换为NMapToPageTable()**

查找所有对MapToPageTable()的调用，替换为NMapToPageTable()，包括：

（1）MemoryDescriptor::EstablishUserPageTable函数



（2）Process::SStack函数



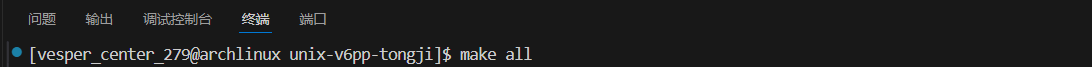
（3）Process::Expand函数

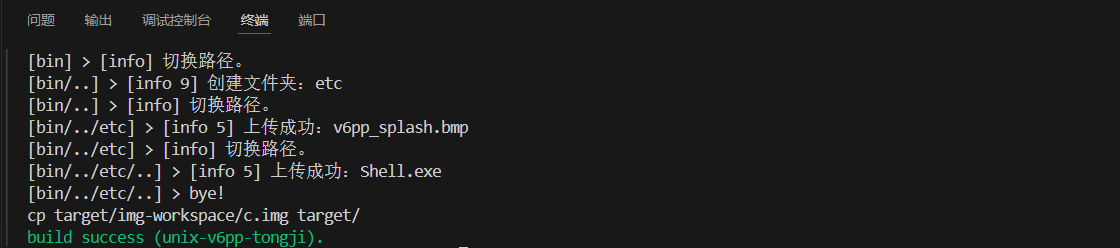


（4）ProcessManager::Swtch()函数



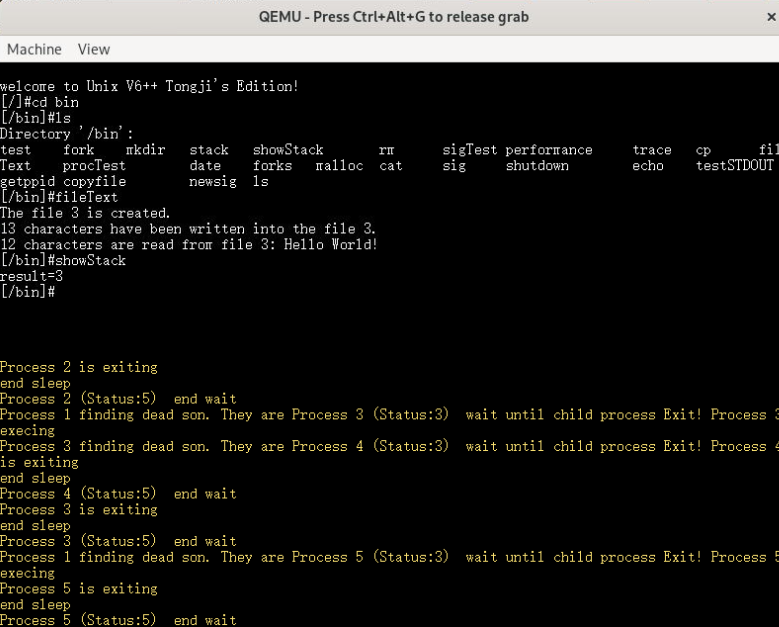
**4.2.4 重新编译运行UNIX V6++代码**





尽管有些warning，但是没有报错，还是编译成功了

**4.2.5 UNIX V6++运行测试**



发现UNIX V6++可以正常运行

**4.3. 删除MEMORYDESCRIPTOR类中其他与相对虚实地址映射表有关的函数**

查看MemoryDescriptor 类的所有数据成员和成员函数，去除其中用于初始化、设置、释放等和相对虚实地址映射表相关或使用了它的函数。所有去除的函数在UNIX V6++中查找被调用的地方，对可以直接去除的予以去除，不能直接去除的，请认真思考需要如何修改。

注意点：

（1）指向相对虚实地址映射映射表的指针不使用，但是其空间必须预留出来

（2）进程相对虚实地址映射映射表的构建函数被调用前，可能会有虚地址空间是否在允许范围内的判断，这个功能需要保留下来。

**4.3.1 所去除的函数**

|  |  |
| --- | --- |
| void Initialize():初始化相对虚实地址映射表 | |
| 函数定义与实现： | 直接删除 |
| 调用位置1： | void ProcessManager::SetupProcessZero()：创建0#进程 |
|  | 处理方式：直接删除，0#进程创建时不再需要相对虚实地址映射表 |
| 调用位置2： | int ProcessManager::NewProc()：创建子进程 |
|  | 处理方式：以下代码全部删除，父进程不再需要为子进程申请相对虚实地址映射表，也不需要将自己的 相对虚实地址映射表复制给子进程 |

|  |  |
| --- | --- |
| void Release()：释放相对虚实地址映射表 | |
| 函数定义与实现： | 直接删除 |
| 调用位置1: | void Process::Exit():进程结束 |
|  | 处理方式：直接删除，进程结束不再需要删除相对虚实地址映射表 |

|  |  |
| --- | --- |
| void MapTextEntrys(unsigned long textStartAddress, unsigned long textSize, unsigned long textPageIdxInPhyMemory)：完成对user结构中页表Entry的填充 | |
| void MapDataEntrys(unsigned long dataStartAddress, unsigned long dataSize, unsigned long dataPageIdxInPhyMemory) ：完成对user结构中页表Entry的填充 | |
| void MapStackEntrys(unsigned long stackSize, unsigned long stackPageIdxInPhyMemory) ：完成对user结构中页表Entry的填充 | |
| 函数定义与实现: | 直接删除 |
| 调用位置： | 这几个函数暂无其它函数调用 |

|  |  |
| --- | --- |
| void MapToPageTable()：利用相对虚实地址映射表构建物理页表 | |
| 函数定义与实现： | 直接删除 |
| 调用位置： | 4.2中调用MapToPageTable()的位置已经被改成了NMapToPageTable() |

|  |  |
| --- | --- |
| void ClearUserPageTable()：清除相对虚实地址映射表 | |
| 函数定义与实现： | 直接删除 |
| 调用位置1： | bool MemoryDescriptor::EstablishUserPageTable  创建虚实地址映射表并构建物理页表 |
|  | 处理方式：直接删除，因为不需要构建相对虚实地址映射表，所以不需要清除 |

|  |  |
| --- | --- |
| PageTable\* GetUserPageTableArray()：获得指向相对虚实地址映射表的指针 | |
| 函数定义与实现： | 直接删除 |
| 调用位置： | 该函数暂无其它函数调用 |

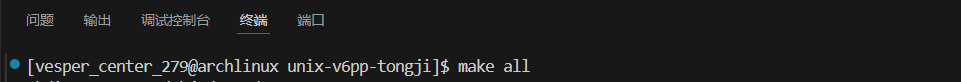
|  |  |
| --- | --- |
| bool EstablishUserPageTable(unsigned long textVirtualAddress, unsigned long textSize, unsigned long dataVirtualAddress, unsigned long dataSize, unsigned long stackSize)：创建虚实地址映射表并构建物理页表 | |
| 函数定义与实现： | 判断虚地址空间是否合法以及构建物理页表的功能需要保留；  创建相对虚实地址映射表的功能需要被删除 |
| 调用位置1： | void Process::SStack()：用于堆栈溢出时，自动扩展堆栈 |
|  | 处理方式：不需再构建相对虚实地址映射表，只需要检查虚地址空间是否在允许范围内 |
| 调用位置2： | void Process::SBreak()：break系统调用函数 |
|  | 处理方式：不需再构建相对虚实地址映射表，只需要检查虚地址空间是否在允许范围内 |
| 调用位置3： | void ProcessManager::Exec()：进程执行函数 |
|  | 处理方式：虚地址合法已有判断，所以直接删除即可 |

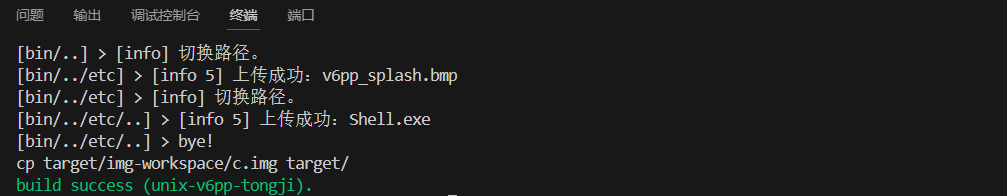
|  |  |
| --- | --- |
| unsigned int MapEntry(unsigned long virtualAddress, unsigned int size, unsigned long phyPageIdx, bool isReadWrite)：构建相对虚实地址映射表所用函数 | |
| 函数定义与实现： | 直接删除 |
| 调用位置： | 该函数暂无其它函数调用 |

**4.3.2 所去除的变量**

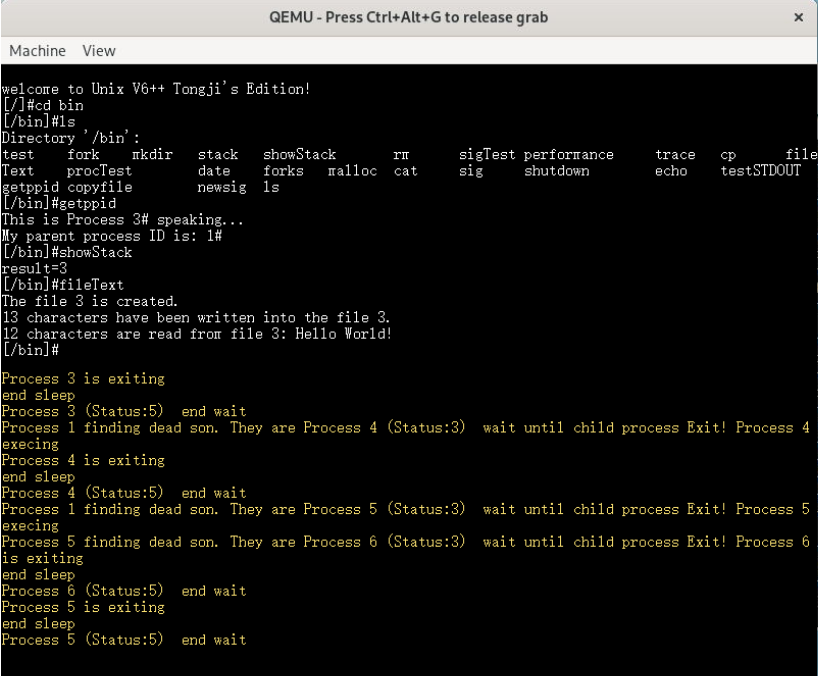
|  |  |
| --- | --- |
| PageTable\* m\_UserPageTableArray:指向相对虚实地址映射表的指针 | |
| 定义： | 不删除，但要保留所占空间，所以将其设置为空指针 |
| 使用位置： | void ProcessManager::SetupProcessZero()：创建0#进程 |
|  | 处理方法：保留空间，不删除，但是设置为NULL |

**4.3.3 重新编译运行UNIX V6++代码**





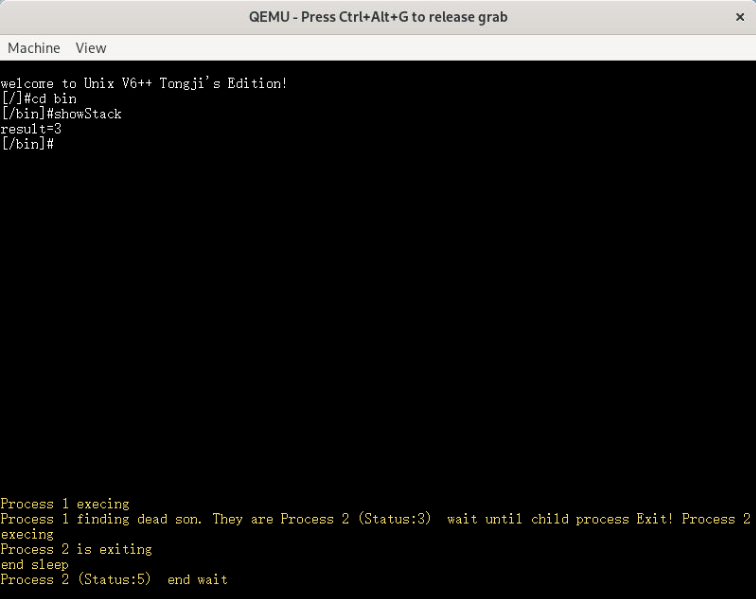
**4.3.4 UNIX V6++运行测试**



发现UNIX V6++可以正常运行

**4.4. 调试验证新的页表系统**

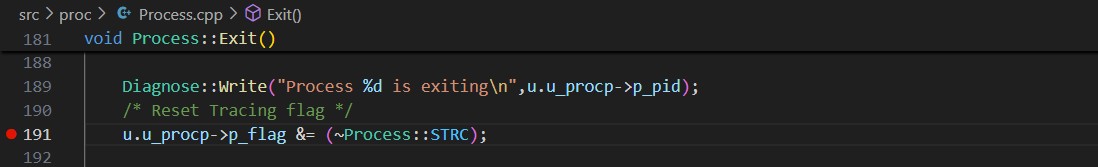
**4.4.1 重新运行程序showStack**



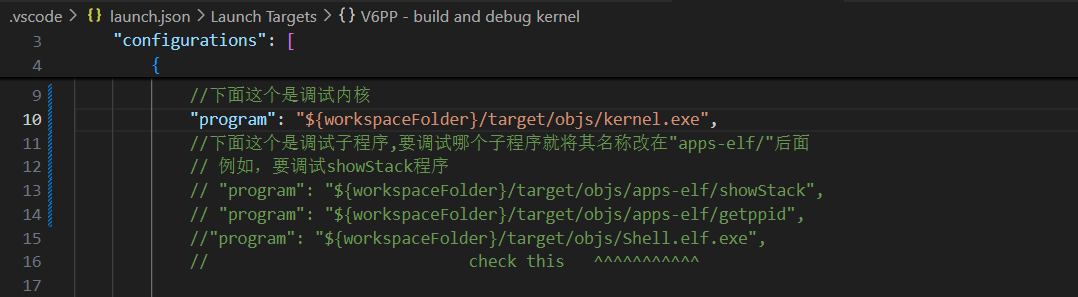
发现运行结果正确

**4.4.2 在调试模式下，让程序和实验三中停在相同的位置**

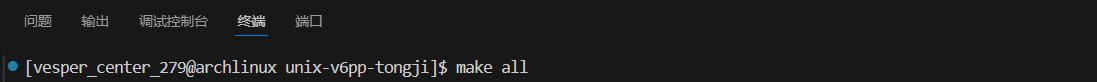
（1）设置断点

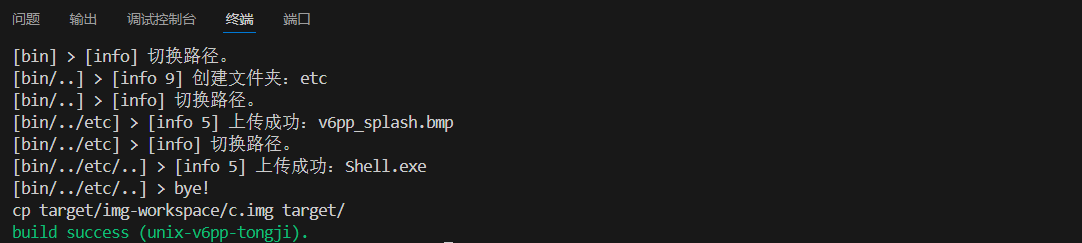


（2）设置调试对象为Kernel.exe（这里之前就是内核，所以不用再修改）

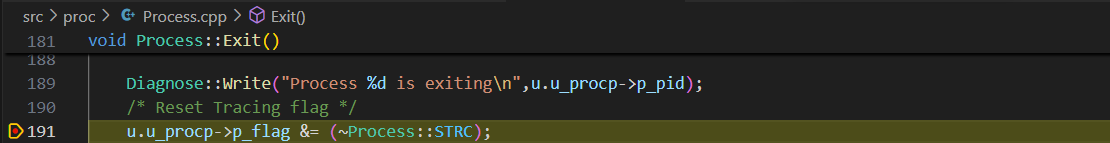


（3）重新编译





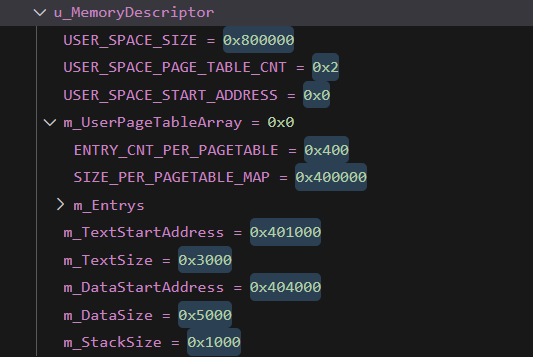
（4）运行UNIX V6++的调试模式并调试停止在断点处



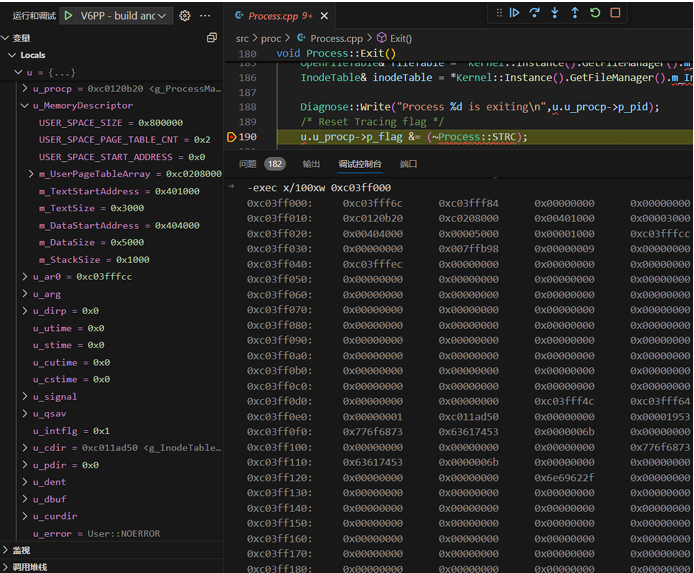
**4.4.3 问题一**

MemoryDescription类中的其他成员变量与实验三中的结果是否一致。

本次实验截图如下：



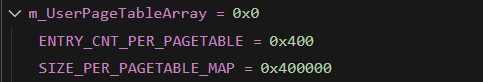
实验三中的结果如下（这里是当时实验报告的截图）：



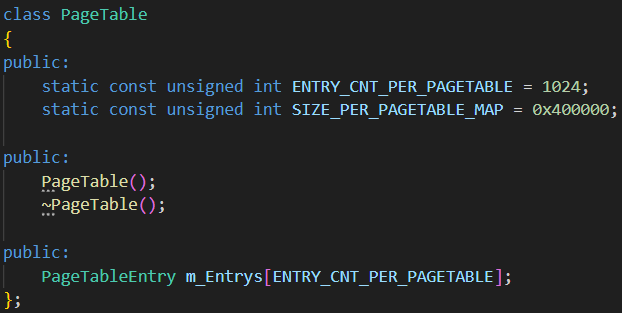
可以发现，除了m\_UserPageTableArray变量的值不同外，没有其他区别。其中，m\_UserPageTableArray变量的值不同在于本次实验中m\_UserPageTableArray设置成了NULL，所以是0x0，而第三次实验并未这样设置，所以不是0x0，是0xC0208000.

4.4.4 问题二

实验三中相对虚实地址映射表所在内存单元现在显示的内容是什么？



从截图中可以看到，m\_UserPageTableArray指针是0x0，后面是PageTable类的两个常数。



4.4.5 问题三

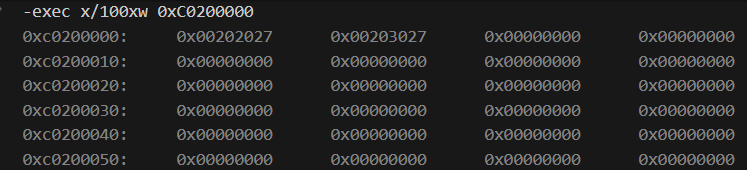
0x200~0x203四个物理页框中的页表内容与实验三中是否一致？

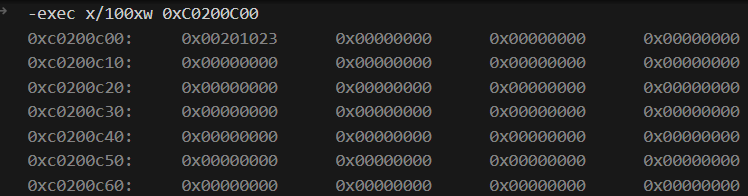
解答：

这四张表的逻辑地址：

这四张页表按照：目录页、内核页表、用户页表1、用户页表2的顺序排列在逻辑内存的3G+2M~3G+2M-4K的前四个页框内。这四张页表开始位置的逻辑地址依次为： 0xC0200000（3G+2M）、0xC0201000（3G+2M+4K）、0xC0202000（3G+2M+8K）、 0xC0203000（3G+2M+12K）。

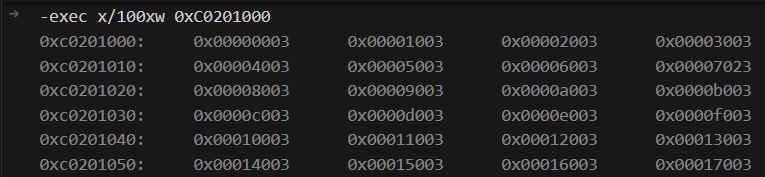
（1）目录页

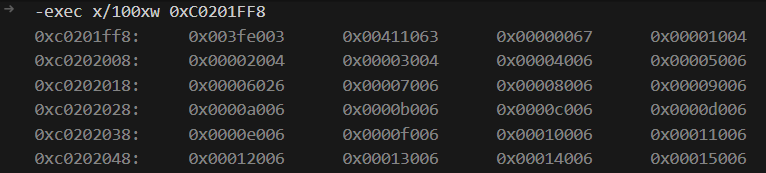




经检验，与实验三的结果完全一致

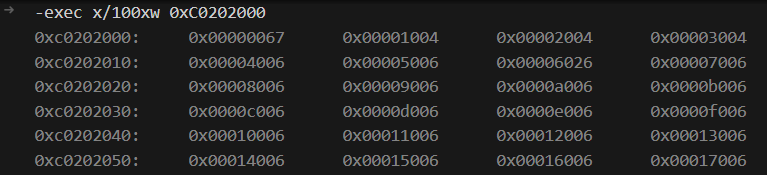
（2）内核页表

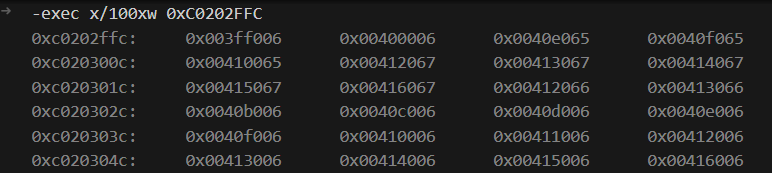




经检验，与实验三的结果完全一致

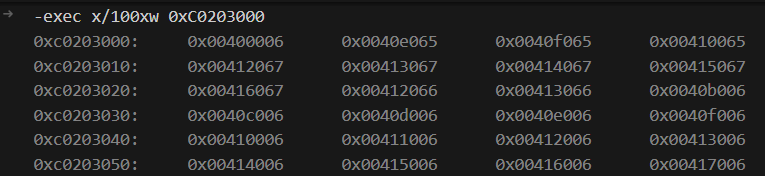
（3）用户页表1

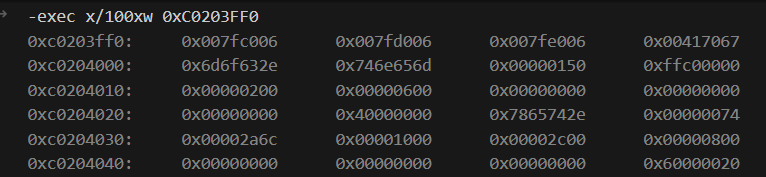




经检验，与实验三的结果完全一致

（4）用户页表2





经检验，与实验三的结果完全一致