

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：

```
void MapToPageTable();
```

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)

所占页框数 = (对应部分的大小 + (页框大小 - 1)) / 页框大小

- (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_p
age)
            {
                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

```

/* 将相对地址映照表根据正文段和数据段在内存中的起始地址pText->x_caddr、p_addr, 建立用户态内存区的页表映射 */
// this->MapToPageTable();//对MapToPageTable()的调用, 替换为NMapToPageTable()的调用-修改处1
this->NMapToPageTable();

```

(2) Process::SStack 函数

```

// u.u_MemoryDescriptor.MapToPageTable();//对MapToPageTable()的调用, 替换为NMapToPageTable()的调用-修改处2
u.u_MemoryDescriptor.NMapToPageTable();

```

(3) Process::Expand 函数

```

//u.u_MemoryDescriptor.MapToPageTable();//对MapToPageTable()的调用, 替换为NMapToPageTable()的调用-修改处3
u.u_MemoryDescriptor.NMapToPageTable();

```

(4) ProcessManager::Swth()函数

```

// newu.u_MemoryDescriptor.MapToPageTable();//对MapToPageTable()的调用, 替换为NMapToPageTable()的调用-修改处4
newu.u_MemoryDescriptor.NMapToPageTable();

```

4.2.4 重新编译运行 UNIX V6++代码

```

问题 输出 调试控制台 终端 端口
[vesper_center_279@archlinux unix-v6pp-tongji]$ make all

```

```
问题 输出 调试控制台 终端 端口

[bin] > [info] 切换路径。
[bin/..] > [info 9] 创建文件夹: etc
[bin/..] > [info] 切换路径。
[bin/../etc] > [info 5] 上传成功: v6pp_splash.bmp
[bin/../etc] > [info] 切换路径。
[bin/../etc/..] > [info 5] 上传成功: Shell.exe
[bin/../etc/..] > bye!
cp target/img-workspace/c.img target/
build success (unix-v6pp-tongji).
```

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

```
QEMU - Press Ctrl+Alt+G to release grab

Machine View

welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#ls
Directory '/bin':
test   fork   mkdir   stack   showStack   rm       sigTest performance   trace   cp       fil
Text   procTest   date    forks   malloc   cat      sig      shutdown   echo    testSTDOUT
getppid copyfile   newsig   ls
[/bin]#fileText
The file 3 is created.
13 characters have been written into the file 3.
12 characters are read from file 3: Hello World!
[/bin]#showStack
result=3
[/bin]#

Process 2 is exiting
end sleep
Process 2 (Status:5) end wait
Process 1 finding dead son. They are Process 3 (Status:3) wait until child process Exit! Process 3
execing
Process 3 finding dead son. They are Process 4 (Status:3) wait until child process Exit! Process 4
is exiting
end sleep
Process 4 (Status:5) end wait
Process 3 is exiting
end sleep
Process 3 (Status:5) end wait
Process 1 finding dead son. They are Process 5 (Status:3) wait until child process Exit! Process 5
execing
Process 5 is exiting
end sleep
Process 5 (Status:5) end wait
```

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表的构造函数被调用前，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

<code>void Initialize()</code> ：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：	<code>void ProcessManager::SetupProcessZero()</code> ：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 <pre>55 // u.u_MemoryDescriptor.Initialize();</pre>
调用位置 2：	<code>int ProcessManager::NewProc()</code> ：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 <pre>89 /* 将父进程的用户态页表指针m_UserPageTableArray备份至pgTable */ 90 // PageTable* pgTable = u.u_MemoryDescriptor.m_UserPageTableArray; 91 // u.u_MemoryDescriptor.Initialize(); 92 /* 父进程的相对地址映照表拷贝给子进程，共两张页表的大小 */ 93 // if (NULL != pgTable) 94 // { 95 // u.u_MemoryDescriptor.Initialize(); 96 // Utility::MemCopy((unsigned long)pgTable, (unsigned long)u.u_MemoryDescriptor.m_UserPageTable 97 // sizeof(PageTable) * MemoryDescriptor::USER_SPACE_PAGE_TABLE_CNT); 98 // } 136 // u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;</pre>

<code>void Release()</code> ：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：	<code>void Process::Exit()</code> ：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 <pre>235 /* 释放内存资源 */ 236 { // u.u_MemoryDescriptor.Release();</pre>

<code>void MapTextEntrys(unsigned long textStartAddress, unsigned long textSize, unsigned long textPageIdxInPhyMemory)</code> ：完成对 user 结构中页表 Entry 的填充	
<code>void MapDataEntrys(unsigned long dataStartAddress, unsigned long dataSize, unsigned long dataPageIdxInPhyMemory)</code> ：完成对 user 结构中页表 Entry 的填充	
<code>void MapStackEntrys(unsigned long stackSize, unsigned long stackPageIdxInPhyMemory)</code> ：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

<code>void MapToPageTable()</code> ：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 <pre>97 // this->ClearUserPageTable();</pre>

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 <pre>334 // if (false == u.u_MemoryDescriptor.EstablishUserPageTable(md.m_TextStartAddress, 335 // md.m_TextSize, md.m_DataStartAddress, md.m_DataSize, md.m_StackSize)) 336 // { 337 // u.u_error = User::ENOMEM; 338 // return; 339 // } 340 if (md.m_TextSize + md.m_DataSize + md.m_StackSize + PageManager::PAGE_SIZE > md.USER_SPACE_SIZE - md.m_TextStartAddress) 341 { 342 u.u_error = User::ENOMEM; 343 Diagnose::Write("u.u_error = %d\n", u.u_error); 344 return; 345 }</pre>
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 <pre>374 // if (false == u.u_MemoryDescriptor.EstablishUserPageTable(md.m_TextStartAddress, 375 // md.m_TextSize, md.m_DataStartAddress, newSize, md.m_StackSize)) 376 // { 377 // // 系统调用出错时, 不可以用这种方式返回, 执行这条路径会导致 u.u_intflg == 1, u.u_error 被错误修改为 EINTR (4); 无论何故导致系统调用失败 378 // // aRetu(u.u_gsav); 379 // return; 380 // } 381 if (md.m_TextSize + md.m_DataSize + md.m_StackSize + PageManager::PAGE_SIZE > md.USER_SPACE_SIZE - md.m_TextStartAddress) 382 { 383 u.u_error = User::ENOMEM; 384 Diagnose::Write("u.u_error = %d\n", u.u_error); 385 return; 386 }</pre>
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 <pre>671 /* 根据正文段、数据段、堆栈段长度建立相对地址映射表, 并加载到页表中 */ 672 // u.u_MemoryDescriptor.EstablishUserPageTable(textAddr, textSize, dataAddr, dataSize, stackSize);</pre>

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:	<code>void ProcessManager::SetupProcessZero():</code> 创建 0#进程
	处理方法: 保留空间, 不删除, 但是设置为 NULL
	<div>54 <code>u.u_MemoryDescriptor.m_UserPageTableArray = NULL;</code></div>

4.3.3 重新编译运行 UNIX V6++代码

问题 输出 调试控制台 终端 端口

•

[vesper_center_279@archlinux unix-v6pp-tongji]\$ make all

问题 输出 调试控制台 终端 端口

[bin/..] > [info] 切换路径。
[bin/../etc] > [info 5] 上传成功: v6pp_splash.bmp
[bin/../etc] > [info] 切换路径。
[bin/../etc/..] > [info 5] 上传成功: Shell.exe
[bin/../etc/..] > bye!
cp target/img-workspace/c.img target/
build success (unix-v6pp-tongji).

4.3.4 UNIX V6++运行测试

QEMU - Press Ctrl+Alt+G to release grab

Machine View

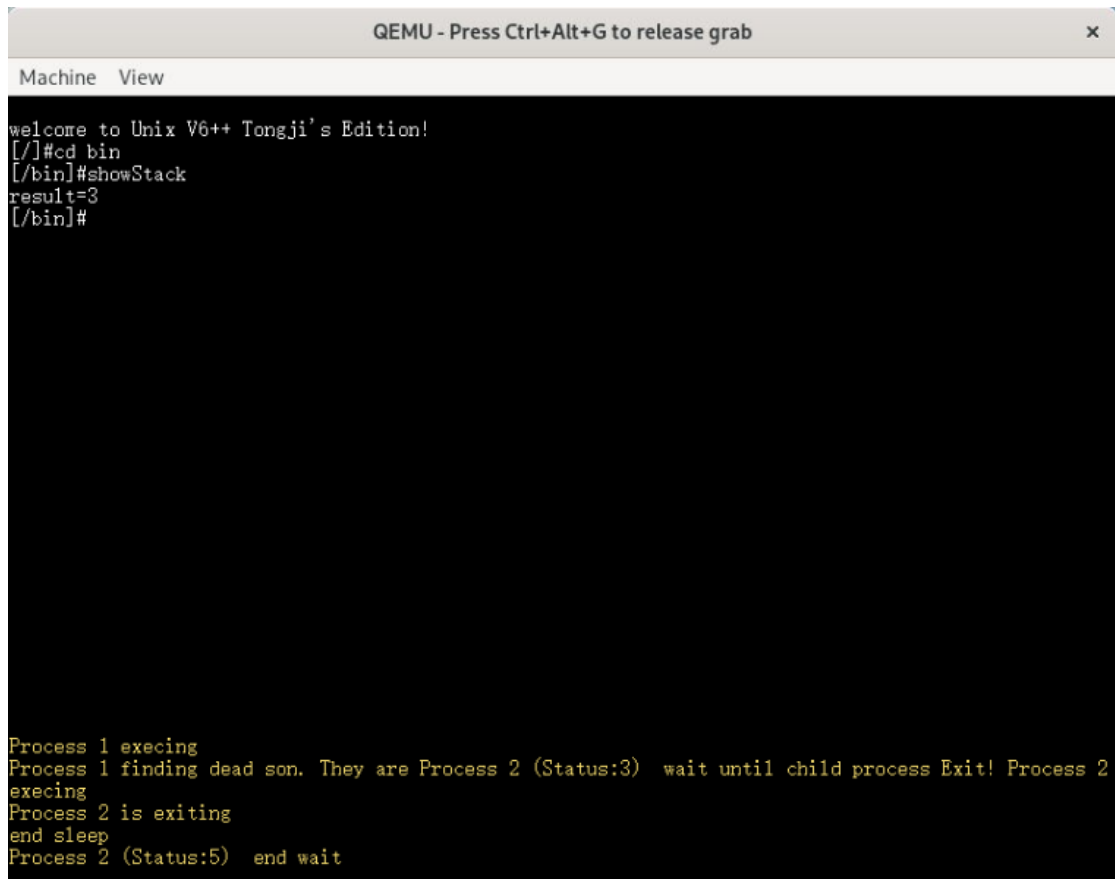
welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#ls
Directory '/bin':
test fork mkdir stack showStack rm sigTest performance trace cp file
Text procTest date forks malloc cat sig shutdown echo testSTDOUT
getppid copyfile newsig ls
[/bin]#getppid
[/bin]#getppid
This is Process 3# speaking...
My parent process ID is: 1#
[/bin]#showStack
result=3
[/bin]#fileText
The file 3 is created.
13 characters have been written into the file 3.
12 characters are read from file 3: Hello World!
[/bin]#

Process 3 is exiting
end sleep
Process 3 (Status:5) end wait
Process 1 finding dead son. They are Process 4 (Status:3) wait until child process Exit! Process 4
execing
Process 4 is exiting
end sleep
Process 4 (Status:5) end wait
Process 1 finding dead son. They are Process 5 (Status:3) wait until child process Exit! Process 5
execing
Process 5 finding dead son. They are Process 6 (Status:3) wait until child process Exit! Process 6
is exiting
end sleep
Process 6 (Status:5) end wait
Process 5 is exiting
end sleep
Process 5 (Status:5) end wait

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



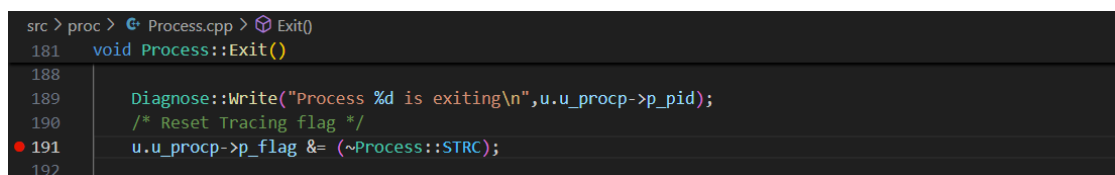
```
QEMU - Press Ctrl+Alt+G to release grab
Machine View
welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#showStack
result=3
[/bin]#

Process 1 execing
Process 1 finding dead son. They are Process 2 (Status:3) wait until child process Exit! Process 2
execing
Process 2 is exiting
end sleep
Process 2 (Status:5) end wait
```

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



```
src > proc > Process.cpp > Exit()
181 void Process::Exit()
188
189 Diagnose::Write("Process %d is exiting\n",u.u_procp->p_pid);
190 /* Reset Tracing flag */
191 u.u_procp->p_flag &= (~Process::STRC);
192
```

(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)

```
.vscode > launch.json > Launch Targets > {} V6PP - build and debug kernel
3   "configurations": [
4   {
9   //下面这个是调试内核
10  "program": "${workspaceFolder}/target/objs/kernel.exe",
11  //下面这个是调试子程序,要调试哪个子程序就将其名称改在"apps-elf/"后面
12  // 例如, 要调试showStack程序
13  // "program": "${workspaceFolder}/target/objs/apps-elf/showStack",
14  // "program": "${workspaceFolder}/target/objs/apps-elf/getppid",
15  // "program": "${workspaceFolder}/target/objs/Shell.elf.exe",
16  //      check this  ^^^^^^^^^^^
17  }
```

(3) 重新编译

```
问题 输出 调试控制台 终端 端口
[vesper_center_279@archlinux unix-v6pp-tongji]$ make all

问题 输出 调试控制台 终端 端口
[bin] > [info] 切换路径。
[bin/..] > [info 9] 创建文件夹: etc
[bin/..] > [info] 切换路径。
[bin/../etc] > [info 5] 上传成功: v6pp_splash.bmp
[bin/../etc] > [info] 切换路径。
[bin/../etc/..] > [info 5] 上传成功: Shell.exe
[bin/../etc/..] > bye!
cp target/img-workspace/c.img target/
build success (unix-v6pp-tongji).
```

(4) 运行 UNIX V6++的调试模式并调试停止在断点处

```
src > proc > Process.cpp > Exit()
181 void Process::Exit()
188
189 Diagnose::Write("Process %d is exiting\n",u.u_procp->p_pid);
190 /* Reset Tracing flag */
191 u.u_procp->p_flag &= (~Process::STRC);
```

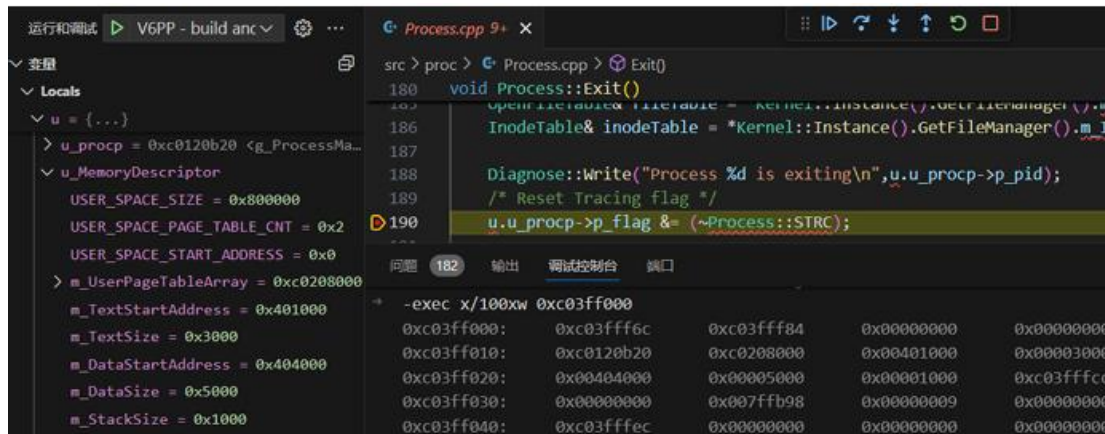
4.4.3 问题一

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：

```
u_MemoryDescriptor
  USER_SPACE_SIZE = 0x800000
  USER_SPACE_PAGE_TABLE_CNT = 0x2
  USER_SPACE_START_ADDRESS = 0x0
  m_UserPageTableArray = 0x0
    ENTRY_CNT_PER_PAGETABLE = 0x400
    SIZE_PER_PAGETABLE_MAP = 0x400000
  m_Entrys
    m_TextStartAddress = 0x401000
    m_TextSize = 0x3000
    m_DataStartAddress = 0x404000
    m_DataSize = 0x5000
    m_StackSize = 0x1000
```

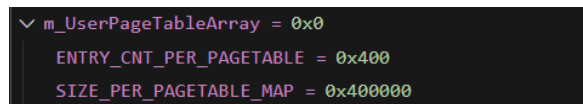
实验三中的结果如下（这里是当时实验报告的截图）：



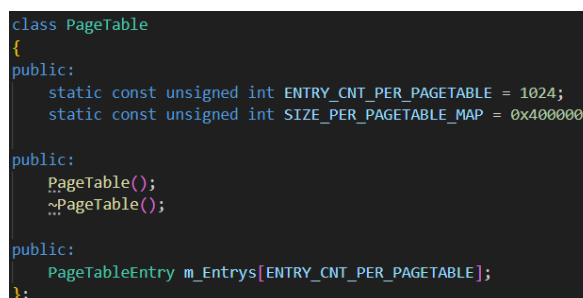
可以发现，除了 `m_UserPageTableArray` 变量的值不同外，没有其他区别。其中，`m_UserPageTableArray` 变量的值不同在于本次实验中 `m_UserPageTableArray` 设置成了 NULL，所以是 0x0，而第三次实验并未这样设置，所以不是 0x0，是 0xc0208000。

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 $3G+2M \sim 3G+2M+4K$ 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

-exec x/100xw 0xC0200000				
0xc0200000:	0x00202027	0x00203027	0x00000000	0x00000000
0xc0200010:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200020:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200030:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200040:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200050:	0x00000000	0x00000000	0x00000000	0x00000000

-exec x/100xw 0xC0200C00				
0xc0200c00:	0x00201023	0x00000000	0x00000000	0x00000000
0xc0200c10:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200c20:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200c30:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200c40:	0x00000000	0x00000000	0x00000000	0x00000000
0xc0200c50:	0x00000000	0x00000000	0x00000000	0x00000000

经检验，与实验三的结果完全一致

(2) 内核页表

-exec x/100xw 0xC0201000				
0xc0201000:	0x00000003	0x00001003	0x00002003	0x00003003
0xc0201010:	0x00004003	0x00005003	0x00006003	0x00007023
0xc0201020:	0x00008003	0x00009003	0x0000a003	0x0000b003
0xc0201030:	0x0000c003	0x0000d003	0x0000e003	0x0000f003
0xc0201040:	0x00010003	0x00011003	0x00012003	0x00013003
0xc0201050:	0x00014003	0x00015003	0x00016003	0x00017003

-exec x/100xw 0xC0201FF8				
0xc0201ff8:	0x003fe003	0x00411063	0x00000067	0x00001004
0xc0202008:	0x00002004	0x00003004	0x00004006	0x00005006
0xc0202018:	0x00006026	0x00007006	0x00008006	0x00009006
0xc0202028:	0x0000a006	0x0000b006	0x0000c006	0x0000d006
0xc0202038:	0x0000e006	0x0000f006	0x00010006	0x00011006
0xc0202048:	0x00012006	0x00013006	0x00014006	0x00015006

经检验，与实验三的结果完全一致

(3) 用户页表 1

→ -exec x/100xw 0xC0202000				
0xc0202000:	0x00000067	0x00001004	0x00002004	0x00003004
0xc0202010:	0x00004006	0x00005006	0x00006026	0x00007006
0xc0202020:	0x00008006	0x00009006	0x0000a006	0x0000b006
0xc0202030:	0x0000c006	0x0000d006	0x0000e006	0x0000f006
0xc0202040:	0x00010006	0x00011006	0x00012006	0x00013006
0xc0202050:	0x00014006	0x00015006	0x00016006	0x00017006

→ -exec x/100xw 0xC0202FFC				
0xc0202ffc:	0x003ff006	0x00400006	0x0040e065	0x0040f065
0xc020300c:	0x00410065	0x00412067	0x00413067	0x00414067
0xc020301c:	0x00415067	0x00416067	0x00412066	0x00413066
0xc020302c:	0x0040b006	0x0040c006	0x0040d006	0x0040e006
0xc020303c:	0x0040f006	0x00410006	0x00411006	0x00412006
0xc020304c:	0x00413006	0x00414006	0x00415006	0x00416006

经检验，与实验三的结果完全一致

(4) 用户页表 2

→ -exec x/100xw 0xC0203000				
0xc0203000:	0x00400006	0x0040e065	0x0040f065	0x00410065
0xc0203010:	0x00412067	0x00413067	0x00414067	0x00415067
0xc0203020:	0x00416067	0x00412066	0x00413066	0x0040b006
0xc0203030:	0x0040c006	0x0040d006	0x0040e006	0x0040f006
0xc0203040:	0x00410006	0x00411006	0x00412006	0x00413006
0xc0203050:	0x00414006	0x00415006	0x00416006	0x00417006

→ -exec x/100xw 0xC0203FF0				
0xc0203ff0:	0x007fc006	0x007fd006	0x007fe006	0x00417067
0xc0204000:	0x6d6f632e	0x746e656d	0x00000150	0xffc00000
0xc0204010:	0x00000200	0x00000600	0x00000000	0x00000000
0xc0204020:	0x00000000	0x40000000	0x7865742e	0x00000074
0xc0204030:	0x00002a6c	0x00001000	0x00002c00	0x00000800
0xc0204040:	0x00000000	0x00000000	0x00000000	0x60000020

经检验，与实验三的结果完全一致