实验六：去除 UNIX V6++的相对虚实地址映射表

1. 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。

2. 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。

3. 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。

4. 实验内容

4.1. 实验准备

首先强调，由于此次实验涉及大幅度修改 UNIX V6++的内核源代码，请读者务必做好备份，修改过程中做好记录，养成良好的习惯，以免一旦某个环节失败，内核无法恢复运行！！！

开始本实验之前，请读者认真复习课堂学习到的关于进程相对虚实地址映射表的初始化和进程整个生命周期中对相对虚实地址映射表的使用情况。建议首先打开 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。

关于相对虚实地址映射表的使用，相信读者首先能想到的是：在进程上台的时候，会使用该进程的相对虚实地址映射表构建进程的物理页表，如果没有相对虚实地址映射表，这个构建过程该如何修改呢？让我们就从这里开始吧！

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。

这里给出以下几点修改建议：

（1）利用 MemoryDescriptor 类中记录的代码段，数据段，堆栈段各部分的长度及起始虚拟地址，来分别计算各部分在用户页表中的起始项和占用的项数；

（2）读懂 UNIX V6++原来的设计中，如何利用 MemoryDescriptor 类中记录的代码段，数据段，堆栈段各部分的长度及起始虚拟地址，来构建相对虚实地址映射表（MapEntry 函数和 EstablishUserPageTable 函数），你将得到很大的启发；

（3）建议读者不要在源代码上直接修改，而是重写一个构建页表的函数，并区别命名，例如，可命名为：

```
void NMapToPageTable();
```

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

bool MemoryDescriptor::EstablishUserPageTable;

void Process::Expand(unsigned int newSize);

void Process::SStack();

int ProcessManager::Swtch();

以上可能未列出全部，请读者在实践过程中认真查找，不要遗漏。

替换好之后重新编译 UNIX V6++，如果没有报错，则在远程桌面环境中重新运行 UNIX V6++。如果运行正确，恭喜你，页表构建的过程已经可以不再使用相对虚实地址映射表了。

但是任务还没有结束，请继续后面的步骤。

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。

上述操作大部分都很简单，可以直接去除，但有以下两个难点请读者务必注意：

（1）关于指向相对虚实地址映射映射表的指针

在 MemoryDescriptor 类中定义了一个指向进程相对虚实地址映射的指针 m_UserPageTableArray，该指针由 MemoryDescriptor::Initialize()函数完成初始化，在为进程分配两个连续页框的相对虚实地址映射表后，将起始地址赋值给 m_UserPageTableArray，而 MemoryDescriptor::Release()在撤销进程的相对虚实地址映射表后，将该指针赋值为 NULL。

所以，一旦取消 MemoryDescriptor::Initialize()，MemoryDescriptor::Release()后， m_UserPageTableArray 指针必须从代码中去除，并将原内核代码中所有使用该指针的位置全部注释掉，否则因为可能会存在空指针的情况而导致内核崩溃。

但是需要注意的是，m_UserPageTableArray 指针虽然在代码中消失，但是该指针原有的存储空间还要占满，否则可能会导致 User 结构的大小发生变化。内核动一发而牵全身，不知道哪里会埋着一颗雷，因为 User 结构大小变化而引爆，所以请读者在这里务必想办法将 m_UserPageTableArray 指针原有的存储空间占满。

（2）关于进程相对虚实地址映射映射表的构建

还有一个特别需要注意的是函数 MemoryDescriptor::EstablishUserPageTable()，该函数的目的是构建进程的相对虚实地址映射表，出现在进程的图像变化的很多函数中，比如：SStack，SBreak，Expand，Exec

等，涉及很多细节，读者在改动过程中，务必小心谨慎。原因在于，该函数在真正构造相对虚实地址映射表之前，还会判断进程的虚地址空间是否在允许的范围内，这个功能必须保留。

我们以进程堆栈扩展后重建相对虚实地址映射表为例，相应的 SStack 的代码如下：

```
void Process::SStack()
{
	User& u = Kernel::Instance().GetUser();
	MemoryDescriptor& md = u.u_MemoryDescriptor;
	unsigned int change = 4096;
	md.m_StackSize += change;
	unsigned int newSize = ProcessManager::USIZE + md.m_DataSize + md.m_StackSize;
	// 以下代码使用 EstablishUserPageTable 函数判断虚地址是否合理，在合理的前提下重建页表
	if ( false == u.u_MemoryDescriptor.EstablishUserPageTable(md.m_TextStartAddress,md.m_TextSize, md.m_DataStartAddress,
                                                              md.m_DataSize, md.m_StackSize) )
	{
		u.u_error = User::ENOMEM;
		return;
	}
	this->Expand(newSize);
	int dst = u.u_procp->p_addr + newSize;
	unsigned int count = md.m_StackSize - change;
	while(count--)
	{
		dst--;
		Utility::CopySeg(dst - change, dst);
	}
	u.u_MemoryDescriptor.NMapToPageTable();
}
```

上述代码中高亮显示的部分，通过调用 MemoryDescriptor::EstablishUserPageTable()函数，在判断进程的虚地址空间合理的前提下，为进程重建相对虚实地址映射表。建议可以将这部分高亮显示的代码改成下面的样子：

```
if ( md.m_TextSize + md.m_DataSize + md.m_StackSize + PageManager::PAGE_SIZE> md.USER_SPACE_SIZE - md.m_TextStartAddress)
{
    u.u_error = User::ENOMEM;
    Diagnose::Write("u.u_error = %d\n",u.u_error);
    return;
}
```

即：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。

这里我们只给出了一个例子，类似这样的情况在很多地方都出现，请读者认真查找，根据不同情况具体分析 MemoryDescriptor::EstablishUserPageTable()的替代方法。

这部分实验是本次实验的攻坚阶段，上面我们只是给出了两个具有代表性的问题，实际的修改过程中，请读者务必耐心细致，并随时做好备份和记录，灵活运用各种调试手段帮助查找问题。

MemoryDescriptor 类中所有与相对虚实地址映射表有关的数据成员和成员函数都去除之后，重新编译 UNIX V6++，如果没有报错，则在远程桌面环境中重新运行。如果运行正确，再次恭喜你，已经看到了胜利的曙光，离最后的成功又迈进了一大步。

4.4. 调试验证新的页表系统

本次实验的最后一步，通过调试程序，并观察内存状态，来确定修改过的页表系统工作正确。打开实验三，重新运行程序 showStack，首先确定程序的运行结果是正确的。

接着，在调试模式下，让程序和实验三中停在相同的位置。观察以下内容：

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

（2）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？

（3）0x200~0x203 四个物理页框中的页表内容与实验三中是否一致？

如果上述内容都显示新的页表系统工作正常，那么正在庆祝胜利的时刻到了。相对虚实地址映射在你的系统里除了保留了一个指针的占位之外，什么也没有留下了。

5. 实验报告要求

本次实验报告需完成以下内容：

（1）（3 分）完成实验 4.2，编写一段正确的用户物理页表生成代码。要求在实验报告中给出算法说明，修改后的完整代码和注释，编译通过，内核能启动，并能正确运行。

（2）（3 分）完成实验 4.3，去除其他与相对虚实地址映射表相关的变量和函数。要求在报告中列出去除的每一个变量与函数，相应的使用位置所在的函数，该函数的作用及处理办法和理由，如果不能直接删除的，给出修改方案和修改后的代码（可参照下表中的 Initialize()函数的示例整理）。全部修改完成后，编译通过，内核能启动，并能正确运行。如果内核无法正常启动，将视报告中记录的修改情况酌情给分。

<table>
<tr><td colspan="2"><code>void Initialize()</code>：初始化相对虚实地址映射表</td></tr>
<tr><td>调用位置 1：</td><td><code>void ProcessManager::SetupProcessZero()</code>：创建 0#进程</td></tr>
<tr><td></td><td>处理方式：直接删除，0#进程创建时不再需要初始化相对虚实地址映射表
<code>57 // u.u_MemoryDescriptor.Initialize();</code></td></tr>
<tr><td>调用位置 2：</td><td><code>int ProcessManager::NewProc()</code>：创建子进程</td></tr>
<tr><td></td><td>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>
90 /*
91 // 将父进程的用户态页表指针m_UserPageTableArray备份至pgTable
92 PageTable* pgTable = u.u_MemoryDescriptor.m_UserPageTableArray;
93 u.u_MemoryDescriptor.Initialize();
94 // 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
95 if (NULL != pgTable)
96 {
97 Utility::MemCopy((unsigned long)pgTable, (unsigned long)u.u_MemoryDescriptor.m_UserPageTableArray, si
98 }
99 */
</pre></td></tr>
</table>

（3）（2 分）完成实验 4.4，将新的页表系统运行结果与实验三的结果比较分析。