同济大学计算机系

操作系统课程实验报告

学　　号 2154312

姓　　名 郑博远

专　　业 计算机科学与技术

授课老师 邓蓉老师

P05：UNIX V6++中去除相对虚实地址映射表

一、 完成的功能

1. 阅读实验文档，理解注释中出现的所有子程序（60 分） √；
2. 去除相对虚实地址映射表，m_UserPageTableArray 赋 NULL（90 分） √；
3. 去除相对虚实地址映射表的指针 m_UserPageTableArray（100 分） √。

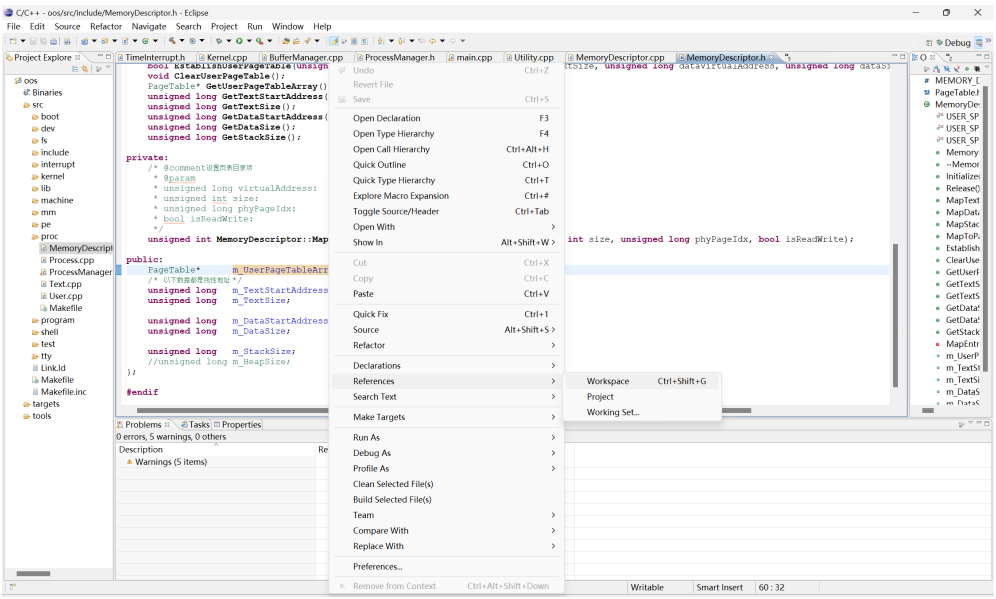
二、 去除相对虚实地址映射表，m_UserPageTableArray 赋 NULL

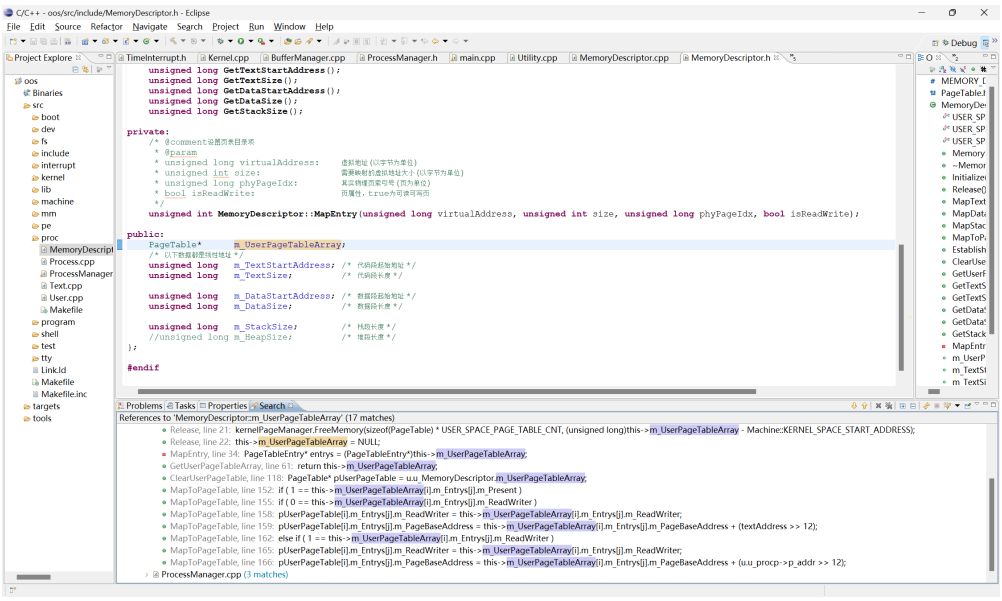
1. 首先，阐述去除相对虚实地址映射表的必要性：

UNIX V6++系统中，每个进程会有一个 MemoryDescriptor 对象，该数据结构存储着当前进程的关键信息，包括代码段、数据段、堆栈段的长度以及起始地址。其目的主要是用来写系统页表，即根据代码段和数据段各自的相对基地址号+（各自实际的物理地址>>12）就可以得到各自真实存储的内存页框。

UNIX V6++中，每个进程都在页表区被分配一张占用两个连续物理页框相对虚实地址映射表。相对虚实地址映射表中记录着代码段、数据段中每一页相对于基地址的偏移量。当进程上台时，会用这两张相对虚实地址映射表的信息填写系统的页表。但是实际上由于 MemoryDescriptor 中已经记录了代码段、数据段和堆栈段的详细信息，完全可以直接对页表进行填写，因此可以去除相对虚实地址映射表。由于每个进程原本都要携带自己的相对虚实地址映射表，将其去除就可以省下内存中的许多空间。

2. 查找所有引用到 m_UserPageTableArray 对象的地方：





3. 修改 MemoryDescriptor.cpp 中对应代码段，并在对应部分添加注释：

```
void MemoryDescriptor::Initialize()
{
    KernelPageManager& kernelPageManager =
Kernel::Instance().GetKernelPageManager();

    /* m_UserPageTableArray需要把AllocMemory()返回的物理内存地址 +
0xC0000000 */
    // this->m_UserPageTableArray =
(PageTable*)(kernelPageManager.AllocMemory(sizeof(PageTable) *
USER_SPACE_PAGE_TABLE_CNT) + Machine::KERNEL_SPACE_START_ADDRESS);

    // 不再申请装相对虚实映射表的空间
    this->m_UserPageTableArray = NULL;
}
```

```
void MemoryDescriptor::Release()
{
    KernelPageManager& kernelPageManager =
Kernel::Instance().GetKernelPageManager();

    // 此处不再释放了，但是由于之前已经赋NULL，所以不修改其实也行

// if ( this->m_UserPageTableArray )
// {
//     kernelPageManager.FreeMemory(sizeof(PageTable) *
USER_SPACE_PAGE_TABLE_CNT, (unsigned long)this->m_UserPageTableArray - Machine::KERNEL_SPACE_START_ADDRESS);
//     this->m_UserPageTableArray = NULL;
// }
}
```

```
unsigned int MemoryDescriptor::MapEntry(unsigned long
virtualAddress, unsigned int size, unsigned long phyPageIdx, bool
isReadWrite)
{
    unsigned long address = virtualAddress -
USER_SPACE_START_ADDRESS;

// //计算从pagetable的哪一个地址开始映射
// unsigned long startIdx = address >> 12;
// unsigned long cnt = ( size + (PageManager::PAGE_SIZE - 1) )/
PageManager::PAGE_SIZE;
//
// PageTableEntry* entrys = (PageTableEntry*)this-
>m_UserPageTableArray;
// for ( unsigned int i = startIdx; i < startIdx + cnt; i++,
phyPageIdx++ )
// {
//     entrys[i].m_Present = 0x1;
//     entrys[i].m_ReadWriter = isReadWrite;
//     entrys[i].m_PageBaseAddress = phyPageIdx;
// }

    // 此处不需要再写页表了，返回一个无效的页框号
    return phyPageIdx;
}
```

```
PageTable* MemoryDescriptor::GetUserPageTableArray()
{
// return this->m_UserPageTableArray;

    // 没有相对虚实地址映射表了，其实不改也可以，因为initialize的时候赋NULL
    return NULL;
}
```

```
void MemoryDescriptor::ClearUserPageTable()
{
// User& u = Kernel::Instance().GetUser();
// PageTable* pUserPageTable =
u.u_MemoryDescriptor.m_UserPageTableArray;
//
// unsigned int i ;
// unsigned int j ;
//
// for (i = 0; i < Machine::USER_PAGE_TABLE_CNT; i++)
// {
//     for (j = 0; j < PageTable::ENTRY_CNT_PER_PAGETABLE; j++ )
//     {
//         pUserPageTable[i].m_Entrys[j].m_Present = 0;
//         pUserPageTable[i].m_Entrys[j].m_ReadWriter = 0;
//         pUserPageTable[i].m_Entrys[j].m_UserSupervisor = 1;
//         pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = 0;
//     }
// }

    // 不用再清空相对虚实地址映射表了（这里必须删，不然NULL指针出错了）
}
```

```
bool MemoryDescriptor::EstablishUserPageTable( unsigned long
textVirtualAddress, unsigned long textSize, unsigned long
dataVirtualAddress, unsigned long dataSize, unsigned long
stackSize )
{
    User& u = Kernel::Instance().GetUser();

    /* 如果超出允许的用户程序最大8M的地址空间限制 */
    if ( textSize + dataSize + stackSize  +
PageManager::PAGE_SIZE > USER_SPACE_SIZE - textVirtualAddress)
    {
        u.u_error = User::ENOMEM;
        Diagnose::Write("u.u_error = %d\n",u.u_error);
        return false;
    }

    // 不建立相对虚实地址映射表了，存下这些信息直接填页表
    m_TextSize = textSize;
    m_DataSize = dataSize;
    m_StackSize = stackSize;

//  this->ClearUserPageTable();
//
//  /* 以相对起始地址phyPageIndex为0，为正文段建立相对地址映照表 */
//  unsigned int phyPageIndex = 0;
//  phyPageIndex = this->MapEntry(textVirtualAddress, textSize,
phyPageIndex, false);
//
//  /* 以相对起始地址phyPageIndex为1，ppda区占用1页4K大小物理内存，为数据
段建立相对地址映照表 */
//  phyPageIndex = 1;
//  phyPageIndex = this->MapEntry(dataVirtualAddress, dataSize,
phyPageIndex, true);
//
//  /* 紧跟着数据段之后，为堆栈段建立相对地址映照表 */
//  unsigned long stackStartAddress = (USER_SPACE_START_ADDRESS +
USER_SPACE_SIZE - stackSize) & 0xFFFFF000;
//  this->MapEntry(stackStartAddress, stackSize, phyPageIndex,
true);

    /* 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();
    return true;
}
```

```
// 这个函数是（原本用相对虚实地址映射表）填写页表的重点，需要大改
void MemoryDescriptor::MapToPageTable()
{
    User& u = Kernel::Instance().GetUser();
    PageTable* pUserPageTable =
Machine::Instance().GetUserPageTableArray();
    unsigned int textAddress = 0;
    if ( u.u_procp->p_textp != NULL )
```

```
    {
        textAddress = u.u_procp->p_textp->x_caddr;
    }

    // tstart_index对应代码段没有起始偏移量， dstart_index对应数据段和
p_addr偏移1个页框号
    unsigned int tstart_index = 0, dstart_index = 1;

    // 计算各个段对应的页框数
    unsigned int text_len  = (m_TextSize  +
(PageManager::PAGE_SIZE - 1)) / PageManager::PAGE_SIZE;
    unsigned int data_len  = (m_DataSize  +
(PageManager::PAGE_SIZE - 1)) / PageManager::PAGE_SIZE;
    unsigned int stack_len = (m_StackSize +
(PageManager::PAGE_SIZE - 1)) / PageManager::PAGE_SIZE;

    // 给数据段页框计数
    unsigned int data_index = 0;

    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

            // 只刷第2个用户页表
            if ( 1 == i )
            {
                /* 只读属性表示正文段页，以pText->x_caddr为内存起始地址 */
                if ( 1 <= j && j <= text_len )  // j==0 为runtime预留
                {
                    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 + tstart_index + (textAddress >> 12);
                }
                /* 读写属性表示数据段对应的页，以p_addr为内存起始地址 */
                else if ( j > text_len && j <= text_len + data_len )
                {
                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress
= (data_index++) + dstart_index + (u.u_procp->p_addr >> 12);
                }
                /* 堆栈段，从末尾 - stack_len开始  */
                else if ( j >= PageTable::ENTRY_CNT_PAGETABLE -
stack_len )
                {
                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress
= (data_index++) + dstart_index + (u.u_procp->p_addr >> 12);
                }
            }
        }
    }
```

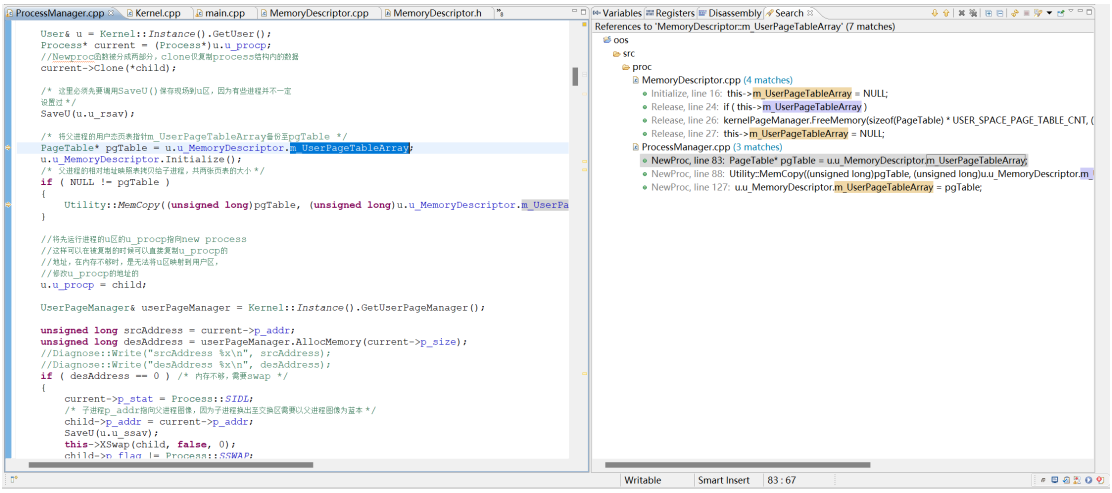
```
    pUserPageTable[0].m_Entrys[0].m_Present = 1;
    pUserPageTable[0].m_Entrys[0].m_ReadWriter = 1;
    pUserPageTable[0].m_Entrys[0].m_PageBaseAddress = 0;

    FlushPageDirectory();
}
```

4. 对于 ProcessManager.cpp 中的对应部分出现在 NewProc() 创建子进程调用中，分别是：

• 83 行：过程开始时为子进程申请两张虚实地址映射表；

• 88 行：将父进程的虚实地址映射表复制给子进程。拷贝进程图像期间，父进程的 m_UserPageTableArray 指向子进程的相对地址映照表。复制完成后恢复；

• 127 行：父进程的相对虚实映射表指针：u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;



由于已经修改了 MemoryDescriptor.cpp 的对应函数，此处可以不用再修改了。

三、 去除相对虚实地址映射表的指针 m_UserPageTableArray

1. 首先，直接废除 MemoryDescriptor::Initialize 函数：

```
// 去除m_UserPageTableArray指针后，这几个函数其实都可以去除
void MemoryDescriptor::Initialize()
{
// KernelPageManager& kernelPageManager =
Kernel::Instance().GetKernelPageManager();

    /* m_UserPageTableArray需要把AllocMemory()返回的物理内存地址 +
0xC0000000 */
```

```
    // this->m_UserPageTableArray =
(PageTable*)(kernelPageManager.AllocMemory(sizeof(PageTable) *
USER_SPACE_PAGE_TABLE_CNT) + Machine::KERNEL_SPACE_START_ADDRESS);

// this->m_UserPageTableArray = NULL;
}
```

2. 去除 NewProc()函数中，对 m_UserPageTableArray 的使用。详见下方注释：

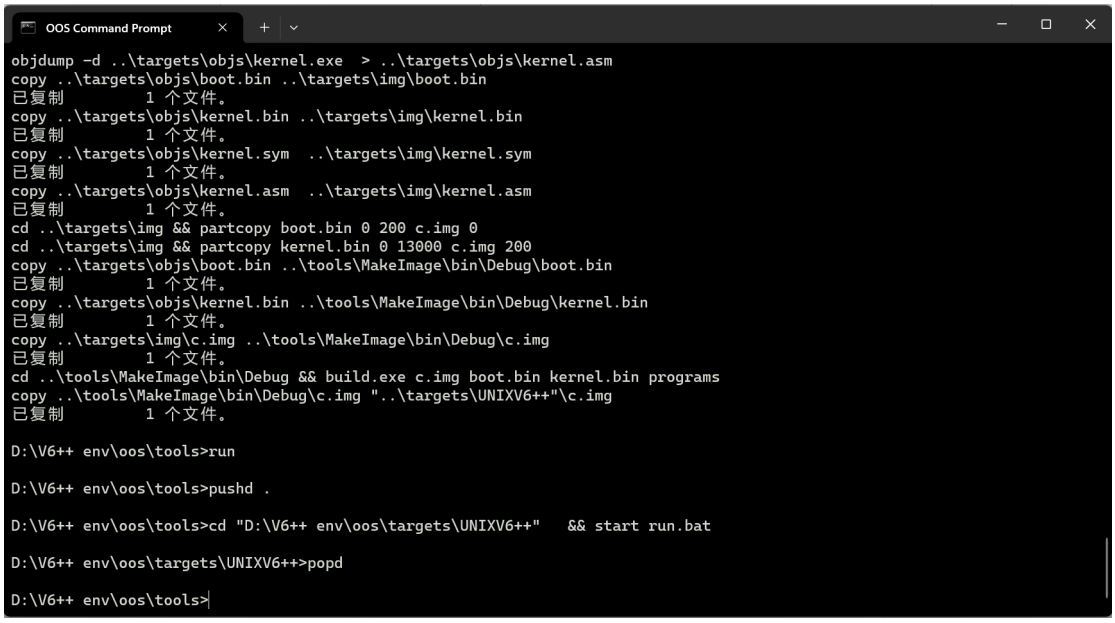
```
int ProcessManager::NewProc()
{
    //Diagnose::Write("Start NewProc()\n");
    Process* child = 0;
    for (int i = 0; i < ProcessManager::NPROC; i++ )
    {
        if ( process[i].p_stat == Process::SNULL )
        {
            child = &process[i];
            break;
        }
    }
    if ( !child )
    {
        Utility::Panic("No Proc Entry!");
    }

    User& u = Kernel::Instance().GetUser();
    Process* current = (Process*)u.u_procp;
    //Newproc函数被分成两部分，clone仅复制process结构内的数据
    current->Clone(*child);

    /* 这里必须先要调用SaveU()保存现场到u区，因为有些进程并不一定
    设置过 */
    SaveU(u.u_rsav);

    // 弃用相对虚实地址映射表，免去这里的拷贝过程，删除多余的pgTable指针

// /* 将父进程的用户态页表指针m_UserPageTableArray备份至pgTable */
// PageTable* pgTable = u.u_MemoryDescriptor.m_UserPageTableArray;
// u.u_MemoryDescriptor.Initialize();
// /* 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 */
// if ( NULL != pgTable )
// {
//     Utility::MemCopy((unsigned long)pgTable, (unsigned
long)u.u_MemoryDescriptor.m_UserPageTableArray, sizeof(PageTable) *
MemoryDescriptor::USER_SPACE_PAGE_TABLE_CNT);
// }

    //将先运行进程的u区的u_procp指向new process
    //这样可以在被复制的时候可以直接复制u_procp的
    //地址，在内存不够时，是无法将u区映射到用户区，
    //修改u_procp的地址的
    u.u_procp = child;

    UserPageManager& userPageManager =
```

```
Kernel::Instance().GetUserPageManager();

    unsigned long srcAddress = current->p_addr;
    unsigned long desAddress = userPageManager.AllocMemory(current-
>p_size);
    //Diagnose::Write("srcAddress %x\n", srcAddress);
    //Diagnose::Write("desAddress %x\n", desAddress);
    if ( desAddress == 0 ) /* 内存不够，需要swap */
    {
        current->p_stat = Process::SIDL;
        /* 子进程p_addr指向父进程图像，因为子进程换出至交换区需要以父进程图像为
蓝本 */
        child->p_addr = current->p_addr;
        SaveU(u.u_ssav);
        this->XSwap(child, false, 0);
        child->p_flag |= Process::SSWAP;
        current->p_stat = Process::SRUN;
    }
    else
    {
        int n = current->p_size;
        child->p_addr = desAddress;
        while (n--)
        {
            Utility::CopySeg(srcAddress++, desAddress++);
        }
    }
    u.u_procp = current;
    /*
     * 拷贝进程图像期间，父进程的m_UserPageTableArray指向子进程的相对地址映照
表;
     * 复制完成后才能恢复为先前备份的pgTable。
     */

    // 这里也不用恢复备份的pgTable了

// u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;

    //Diagnose::Write("End NewProc()\n");
    return 0;
}
```

3. 最后，直接在头文件中去除 m_UserPageTableArray 的指针。

四、 实验结果

1. 依次执行 clean、all、run 指令，如下：



2. 运行修改后的 UNIX V6++系统，能够正常地执行进程：