同济大学计算机系

操作系统 P05 实验报告

学　　号 2151769

姓　　名 吕博文

专　　业 计算机科学与技术

授课老师 邓蓉

实验五：去除相对虚实地址映射表相关实验

本实验实现了文档中的功能 1（阅读源码）、2（去除相对虚实地址映射表）、3（去除相对虚实地址映射表的指针。）

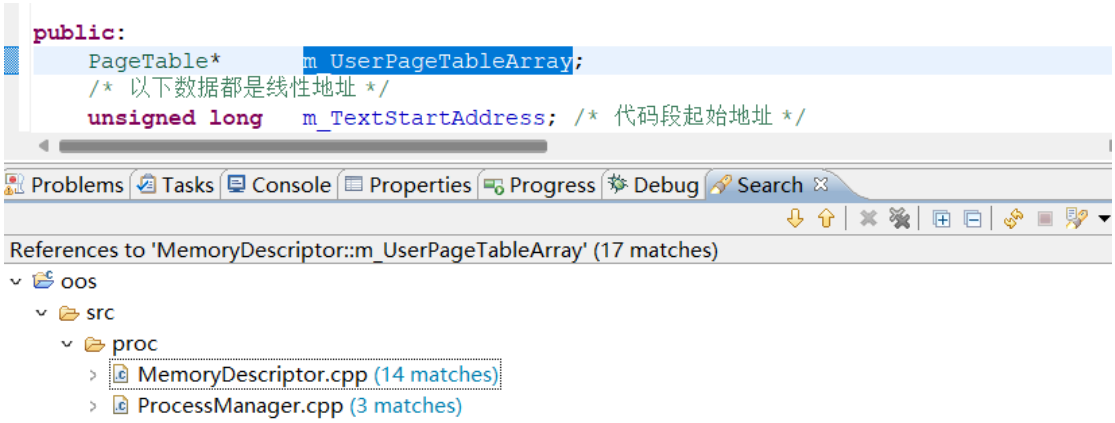
一、 去除相对虚实地址映射表。将内存描述符中的指 m_UserPageTableArray 赋 null。

1、 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 对象用来存储该进程代码段，数据段，堆栈段的相关信息，其中还有一个相对虚实地址映射表，每次进程的可交换部分或代码段在内存中的位置改变时，其程序地址和物理地址之间的地址转换需要改变，UNIX V6++系统借助该相对虚实地址映射表来每次更新系统页表，但是我们考虑到相对虚实地址映射表每次也是由 p_addr 和 x_caddr 来更新的，所以不如每次进程在内存中的位置改变时直接更新系统页表中的内容，从而省去相对虚实地址映射表这一中间过程。

2、 具体实验流程

首先，我们在 User 结构中找到 MemoryDescriptor 对象，继续找到相对虚实地址映射表 m_UserPageTableArray，全局寻找使用该对象的地方：



之后我们依次分析每一次调用该对象的代码含义，并做出相应的注释处理。

MemoryDescriptor.cpp:

```
//初始化分配内存函数
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)
    this->m_UserPageTableArray = NULL;
}
```

该函数为相对虚实地址映射表初始化分配内存函数，该表存储在页表区，所以物理地址和逻辑地址的差值为 3G，这里因为我们要去掉相对虚实地址映射表，所以直接赋值空指针即可。

```
//释放内存函数
void MemoryDescriptor::Release()
{
    KernelPageManager& kernelPageManager = Kernel::Instance().GetKernelPageManager();
    if ( this->m_UserPageTableArray )
    {
        kernelPageManager.FreeMemory(sizeof(PageTable) * USER_SPACE_PAGE_TABLE_CNT, (unsigned l
        this->m_UserPageTableArray = NULL;
    }
}
```

该函数为释放相对虚实地址映射表内存函数，因为我们在初始化时本来就没有分配内存，所以这里可以不改动。

```
//写相对虚实地址映射表
unsigned int MemoryDescriptor::MapEntry(unsigned long virtualAddress, unsigned int size, unsigne
{
    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;
//  }
    return phyPageIdx;
}
```

该函数写相对虚实地址映射表，我们也是直接注释掉核心内容。

```
//得到相对虚实地址映射表
PageTable* MemoryDescriptor::GetUserPageTableArray()
{
    return this->m_UserPageTableArray;
}
```

我们没有对 m_UserPageTableArray 赋值过，所以这里也会直接返回空，不需要改动。

```
//清理相对虚实地址映射表
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;
//      }
//  }

}
```

清理相对虚实地址映射表，常用在进程图像在内存中位置改变后调用，

这里全部注释掉。

```
bool MemoryDescriptor::EstablishUserPageTable( unsigned long textVirtualAddress, unsigned long t
{
    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 - textVirtu
    {
        u.u_error = User::ENOMEM;
        Diagnose::Write("u.u_error = %d\n",u.u_error);
        return false;
    }
    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) &
//  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
    this->MapToPageTable();
    return true;
}
```

该函数是建立相对虚实地址映射表的过程，我们保留对代码段、数据段、

堆栈段信息的更新，对于相对虚实地址映射表的改变注释掉。

```
//根据相对虚实地址映射表写用户页表
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;
    }
    unsigned int text_index = 0,data_index = 1;//代码段和数据段的起始偏移量
    //计算代码段，数据段和堆栈段各自占多少个页框,向上取整
    unsigned int text_size = (m_TextSize + (PageManager::PAGE_SIZE - 1))
            /PageManager::PAGE_SIZE;
    unsigned int data_size  = (m_DataSize + (PageManager::PAGE_SIZE - 1))
            /PageManager::PAGE_SIZE;
    unsigned int stack_size = (m_StackSize + (PageManager::PAGE_SIZE - 1))
            /PageManager::PAGE_SIZE;
    unsigned int dataidx = 0;
    unsigned int textidx = 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] = 0;//先清空
            if(1 == i){
                //只刷新第二个用户页表
                if(1<=j&&j<=text_size){
                    //刷新代码段
                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 0;//代码段不可写
                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = textidx +
                            text_index + (textAddress>>12);
                    textidx++;
                }

                else if(j>text_size && j<=text_size+data_size){
                    //数据段
                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = dataidx +
                            data_index + (u.u_procp->p_addr >> 12);
                    dataidx++;
                }
                else if(j>=PageTable::ENTRY_CNT_PER_PAGETABLE - stack_size){
                    //堆栈段
                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = dataidx +
                            data_index + (u.u_procp->p_addr >>12);
                    dataidx++;
                }
            }
        }
    }

    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();
}
```

根据相对虚实地址映射表改写用户页表，因为本实验中我们要去除了相对虚实地址映射表，所以在该函数我们需要自行计算出需要更新的用户

页表的内容自行完成更新。

ProcessManager.cpp

```
    /* 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;
```

```
    /* 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_UserPage
    }
```

```
    u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;
    //Diagnose::Write("End NewProc()\n");
    return 0;
```

这三个地方不需要改动。

至此，有关相对虚实地址映射表的去除工作已经全部完成，我们选择生成

```
已复制    1 个文件。

**** Build Finished ****
```

运行

```
[/]#cd bin
[/bin]#ls
Directory '/bin':
cat      cat.exe cat1.exe        cp      cp.exe  cpfile.exe      date    date.exe
         echo    echo.exe        forks.exe       ls      ls.exe  malloc.exe
mkdir    mkdir.exe       newsig.exe      perf    perf.exe        rm      rm.exe
showStack.exe   shutdown        shutdown.exe    sig.exe sigTest.exe     stack.ex
e        test.exe        trace   trace.exe
[/bin]#showStack.exe
This is Process 3# speaking...
My parent process ID is 1
```

正常运行。

二、 去除相对虚实地址映射表的指针。注释掉 m_UserPageTableArray，直接用包中的 cmd。依次执行 clean 命令，all 命令，run 命令

选择将相对虚实地址映射表注释掉：

```
public:
    //PageTable*        m_UserPageTableArray;
    /* 以下数据都是线性地址 */
    unsigned long   m_TextStartAddress; /* 代码段起始地址 */
    unsigned long   m_TextSize;         /* 代码段长度 */

    unsigned long   m_DataStartAddress; /* 数据段起始地址 */
    unsigned long   m_DataSize;         /* 数据段长度 */

    unsigned long   m_StackSize;        /* 栈段长度 */
    //unsigned long m_HeapSize;         /* 堆段长度 */
};
```

同时将调用到 m_UserPageTableArray 的地方全部注释掉

```
    /* 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_Us
//  }

    /*
     * 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。
     */
    //u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;
    //Diagnose::Write("End NewProc()\n");
//得到相对虚实地址映射表
PageTable* MemoryDescriptor::GetUserPageTableArray()
{
    //return this->m_UserPageTableArray;
    return NULL;
}
```

之后进入命令行执行命令。

```
C:\UnixV6\oos\tools>clean
del     ..\targets\objs\*.o
del     ..\targets\objs\*.exe
del     ..\targets\objs\*.bin
del     ..\targets\objs\*.sym
del     ..\targets\objs\*.asm
del     ..\targets\img\*.bin
del     ..\targets\img\*.sym
del     ..\targets\img\*.asm
del     "..\targets\UNIXV6++"\c.img
```

```
C:\UnixV6\oos\tools>all
make --directory=boot
make[1]: Entering directory `C:/UnixV6/oos/src/boot'
nasm -f bin boot.s -o ..\..\targets\objs\boot.bin
nasm -f elf sector2.s -o ..\..\targets\objs\sector2.bin
g++ -Wall -O0 -g -nostartfiles -nostdlib -fno-builtin -fno-r
objs\support.o
```

```
已复制         1 个文件。
cd ..\tools\MakeImage\bin\Debug && build.exe c.img boot.bin kernel.bin programs
copy ..\tools\MakeImage\bin\Debug\c.img "..\targets\UNIXV6++"\c.img
已复制         1 个文件。
```

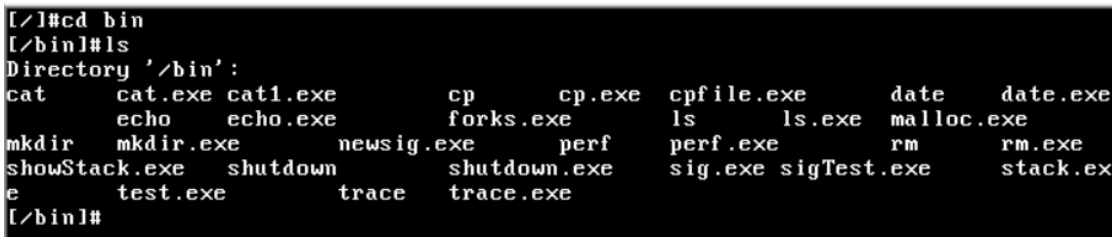
```
C:\UnixV6\oos\tools>run

C:\UnixV6\oos\tools>pushd .

C:\UnixV6\oos\tools>cd "C:\UnixV6\oos\targets\UNIXV6++"   && start run.bat

C:\UnixV6\oos\targets\UNIXV6++>popd

C:\UnixV6\oos\tools>
```


[/]#cd bin
[/bin]#ls
Directory '/bin':
cat cat.exe cat1.exe cp cp.exe cpfile.exe date date.exe
echo echo.exe forks.exe ls ls.exe malloc.exe
mkdir mkdir.exe newsig.exe perf perf.exe rm rm.exe
showStack.exe shutdown shutdown.exe sig.exe sigTest.exe stack.ex
e test.exe trace trace.exe
[/bin]#