

项目二：修改 UnixV6 内核，去掉相对虚实地址映射表

学号：1951443

姓名：罗劲桐

一. 实验目的

1. 修改 UNIX V6++ 内核，去掉相对虚实地址映射表。（MemoryDescriptor.m_UserPageTableArray 保持为 NULL）

二. 实验设备及工具

Eclipse IDE 开发环境

Bochs-2.6 虚拟机

UNIX V6++操作系统

三. 预备知识

1. MemoryDescriptor 中 m_UserPageTableArray 为相对虚实地址映射表。每个进程在 fork、exec 后 EstablishUserPageTable 在内核地址创建相对虚实地址映射表，并在进程退出之前一直存在，直到 exit 释放相对虚实地址映射表。
2. 进程只有在运行过程中有对应的用户页表和页目录。进程调度上台时 MapToPageTable 将相对虚实地址映射表写到真正的用户页表位置；
3. 总体思想：

由于 UNIX V6++ 进程只使用两个用户页表，且代码段、数据段、堆栈段只对应第二页用户页表，因此相对虚实地址映射表与用户页表内容一一对应。而相对虚实地址映射表的内容只与 m_TextSize、m_DataSize、m_StackSize 和 m_TextStartAddress、m_DataStartAddress、USER_SPACE_SIZE 有关，所有项的位置都可以通过这六项计算得出。因此省略 EstablishUserPageTable 这一步，利用这六项数据直接在 MapToPageTable 建立用户页表。

4. 用户页表对应的物理地址：

由于 UNIX V6++ 进程的物理内存连续存放，代码段起始物理地址对应 u.u_procp->p_textp->x_caddr，可交换部分（ppda、代码段、数据段）起始物理地址 u.u_procp->p_addr（其中 ppda 不用写）

四. 实验内容（注：粘贴的代码段中已经删掉了注释的代码部分）

- （一） 修改 MemoryDescriptor 的函数

1. Initialize 函数

```
1. void MemoryDescriptor::Initialize()
2. {
3.     this->m_UserPageTableArray = NULL;
4. }
```

不需要分配相对虚实地址映射表的物理内存，直接赋值 NULL

2. Release 函数

因为没有分配相对虚实地址映射表，所以全部注释掉。

3. MapEntry 函数

```
1. unsigned int MemoryDescriptor::MapEntry(unsigned long virtualAddress, unsigned
   int size, unsigned long phyPageIdx, bool isReadWrite)
2. {
3.     //计算从 pagetable 的哪一个地址开始映射
4.     unsigned long cnt = ( size + (PageManager::PAGE_SIZE - 1) ) /
   PageManager::PAGE_SIZE;
5.     return phyPageIdx + cnt;
6. }
```

本函数作用是根据虚拟地址和物理地址在相对虚实地址映射表中创建 cnt 个表项，并读写权限设定为 isReadWrite。在去掉相对虚实地址映射表后，为了维持本函数的功能（返回下一个可用的物理地址），修改返回值为 phyPageIdx + cnt

4. EstablishUserPageTable 函数

```
1. bool MemoryDescriptor::EstablishUserPageTable( unsigned long textVirtualAddress,
   unsigned long textSize, unsigned long dataVirtualAddress, unsigned long dataSize,
   unsigned long stackSize )
2. {
3.     User& u = Kernel::Instance().GetUser();
4.
5.     /* 如果超出允许的用户程序最大 8M 的地址空间限制 */
6.     if ( textSize + dataSize + stackSize + PageManager::PAGE_SIZE >
   USER_SPACE_SIZE - textVirtualAddress)
7.     {
8.         u.u_error = User::ENOMEM;
9.         Diagnose::Write("u.u_error = %d\n",u.u_error);
10.        return false;
11.    }
12.
13.    this->m_TextSize=textSize;
14.    this->m_DataSize=dataSize;
15.    this->m_StackSize=stackSize;
16.
17.    /* 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 */
18.    this->MapToPageTable();
19.    return true;
20. }
```

本函数已删除构造相对虚实地址映射表的部分，仅保留 OOM 内存溢出报错，保存

输入的 textSize、dataSize、stackSize 地址用于 MapToPageTable 函数构造用户页表。

5. ClearUserPageTable 函数

清空相对虚实地址映射表，因为相对虚实地址映射表不存在，全部注释掉。

6. MapToPageTable 函数

```
1. void MemoryDescriptor::MapToPageTable()
2. {
3.     User& u = Kernel::Instance().GetUser();
4.
5.     //if(u.u_MemoryDescriptor.m_UserPageTableArray == NULL)
6.     //    return;
7.
8.     PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray();
9.
10.    //各段长度和起始 entry
11.    unsigned long textEntryCnt = ( this->m_TextSize + (PageManager::PAGE_SIZE -
12.    1) ) / PageManager::PAGE_SIZE;
13.    unsigned long dataEntryCnt = ( this->m_DataSize + (PageManager::PAGE_SIZE -
14.    1) ) / PageManager::PAGE_SIZE;
15.    unsigned long stackEntryCnt = ( this->m_StackSize + (PageManager::PAGE_SIZE -
16.    1) ) / PageManager::PAGE_SIZE;
17.
18.    unsigned long textStartIdx = (this->m_TextStartAddress >> 12) - 1024;
19.    unsigned long dataStartIdx = (this->m_DataStartAddress >> 12) - 1024;
20.    unsigned long stackStartIdx = (((USER_SPACE_START_ADDRESS + USER_SPACE_SIZE -
21.    this->m_StackSize) & 0xFFFF000) >> 12) - 1024;
22.
23.    unsigned int textEntry = 0;
24.    unsigned int dataEntry = u.u_procp->p_addr >> 12;
25.    if ( u.u_procp->p_textp != NULL )
26.    {
27.        textEntry = u.u_procp->p_textp->x_caddr >> 12;
28.    }
29.
30.    for (unsigned int i = 0; i < Machine::USER_PAGE_TABLE_CNT; i++)
31.    {
32.        for ( unsigned int j = 0; j < PageTable::ENTRY_CNT_PER_PAGETABLE;
33.        j++ )
34.        {
35.            pUserPageTable[i].m_Entrys[j].m_Present = 0;    //先清 0
36.
37.            if ( 1 == i )
38.            {
39.                /* 只读属性表示正文段对应的页，以 pText->x_caddr 为内存
40.                起始地址 */
41.                if ( j >= textStartIdx && j < textStartIdx +
42.                textEntryCnt)
43.                {
44.                    pUserPageTable[i].m_Entrys[j].m_Present
45.                    = 1;
46.
47.                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 0;
```

```

39.     pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j - textStartIdx + textEntry;
40.     }
41.     /* 读写属性表示数据段对应的页，以 p_addr+1 为内存起始
地址 */
42.     else if ( j >= dataStartIdx && j < dataStartIdx +
dataEntryCnt)
43.     {
44.         pUserPageTable[i].m_Entrys[j].m_Present
= 1;
45.
46.         pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
47.         pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j - dataStartIdx + 1 +
dataEntry; //初始为 1, 预留 ppda 空间
48.     }
49.     /* 读写属性表示堆栈段对应的页，以
p_addr+dataEntryCnt+1 为内存起始地址 */
50.     else if ( j >= stackStartIdx && j < stackStartIdx
+ stackEntryCnt)
51.     {
52.         pUserPageTable[i].m_Entrys[j].m_Present
= 1;
53.
54.         pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
55.         pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j + dataEntryCnt -
stackStartIdx + 1 + dataEntry;
56.     }
57. }
58.
59. pUserPageTable[0].m_Entrys[0].m_Present = 1;
60. pUserPageTable[0].m_Entrys[0].m_ReadWriter = 1;
61. pUserPageTable[0].m_Entrys[0].m_PageBaseAddress = 0;
62.
63. FlushPageDirectory();
64. }

```

由于相对虚实地址映射表已去除，代码段、数据段、堆栈段地址对应的线性地址、物理地址 entry 需要计算，其余部分保留不变：

- 这三段长度不一定是 4K 的整数倍，通过向上取证计算 entry 个数

```

1. //各段长度和起始 entry
2. unsigned long textEntryCnt = ( this->m_TextSize + (PageManager::PAGE_SIZE -
1) )/ PageManager::PAGE_SIZE;
3. unsigned long dataEntryCnt = ( this->m_DataSize + (PageManager::PAGE_SIZE -
1) )/ PageManager::PAGE_SIZE;
4. unsigned long stackEntryCnt = ( this->m_StackSize + (PageManager::PAGE_SIZE -
1) )/ PageManager::PAGE_SIZE;

```

- 计算各段对应的起始 entry，因为对应的 entry 都是在第二个用户页表中，所以减去 1024，换算到第二个用户页表对应的 entry 位置

```

1. unsigned long textStartIdx = (this->m_TextStartAddress >> 12) - 1024;

```

```

2. unsigned long dataStartIdx = (this->m_DataStartAddress >> 12) - 1024;
3. unsigned long stackStartIdx = (((USER_SPACE_START_ADDRESS + USER_SPACE_SIZE -
   this->m_StackSize) & 0xFFFFF000) >> 12) - 1024;

```

- 对应物理地址起始 entry，u.u_procp->p_textp->x_caddr 和 u.u_procp->p_addr 计算出

```

1. unsigned int textEntry = 0;
2. unsigned int dataEntry = u.u_procp->p_addr >> 12;
3. if ( u.u_procp->p_textp != NULL )
4. {
5.     textEntry = u.u_procp->p_textp->x_caddr >> 12;
6. }

```

- 计算 m_PageBaseAddress 时，代码段从 0 开始递增. 数据段和堆栈段共用一段连续物理空间，从 1 开始递增（第 0 段是 ppda 区）

```

1. pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j - textStartIdx + textEntry;
2. ...
3. pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j - dataStartIdx + 1 +
   dataEntry; //初始为 1, 预留 ppda 空间
4. ...
5. pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j + dataEntryCnt -
   stackStartIdx + 1 + dataEntry;

```

(二) 修改 ProcessManager 的函数

```

1. int ProcessManager::NewProc()
2. {
3.     //Diagnose::Write("Start NewProc()\n");
4.     Process* child = 0;
5.     for (int i = 0; i < ProcessManager::NPROC; i++)
6.     {
7.         if ( process[i].p_stat == Process::SNULL )
8.         {
9.             child = &process[i];
10.            break;
11.        }
12.    }
13.    if ( !child )
14.    {
15.        Utility::Panic("No Proc Entry!");
16.    }
17.
18.    User& u = Kernel::Instance().GetUser();
19.    Process* current = (Process*)u.u_procp;
20.    //Newproc 函数被分成两部分, clone 仅复制 process 结构内的数据
21.    current->Clone(*child);
22.
23.    /* 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
24.    设置过 */
25.    SaveU(u.u_rsav);
26.
27.    u.u_MemoryDescriptor.Initialize();
28.
29.    (省略中间部分的代码)
30.
31.    /*

```

```

32.    * 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；
33.    * 复制完成后才能恢复为先前备份的 pgTable。
34.    */
35.    //u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;
36.    //Diagnose::Write("End NewProc()\n");
37.    return 0;
38. }

```

fork 执行 NewProc 函数时，不需要再拷贝相对虚实地址映射表。因此不需要备份，直接给予进程初始化为 NULL (u.u_MemoryDescriptor.Initialize())

(三) 例：fork 的过程分析

命令行运行 fork，进入 ProcessManager::Fork()，在检测空闲 process 项后，运行 ProcessManager::NewProc() 函数创建子进程图像。返回后，进行返回值的处理。

```

414 void ProcessManager::Fork()
415 {
416     User& u = Kernel::Instance().GetUser();
417     Process* child = NULL;;
418
419     /* 寻找空闲的process项，作为子进程的进程控制块 */
420     for ( int i = 0; i < ProcessManager::NPROC; i++ )
421     {
422         if ( this->process[i].p_stat == Process::SNULL )
423         {
424             child = &this->process[i];
425             break;
426         }
427     }
428     if ( child == NULL )
429     {
430         /* 没有空闲process表项，返回 */
431         u.u_error = User::EAGAIN;
432         return;
433     }
434
435     if ( this->NewProc() ) /* 子进程返回1，父进程返回0 */
436     {
437         /* 子进程fork()系统调用返回0 */
438         u.u_ar0[User::EAX] = 0;
439         u.u_cstime = 0;
440         u.u_stime = 0;
441         ...

```

ProcessManager::NewProc() 复制 process 段后，进入到配置相对虚实地址映射表的部分，由于相对虚实地址映射表不再分配，此处 Initialize() 仅对 m_UserPageTableArray 置为空指针。其余（分配页表等）在子进程调度上台后在执行，此处相对虚实地址映射表仅执行这一条。这之后，执行复制可交换部分。

```

84     /* 这里必须先调用SaveU()保存现场到u区，因为有些进程并不一定
85     设置过 */
86     SaveU(u.u_rsav);
87
88 //  /* 将父进程的用户态页表指针m_UserPageTableArray备份至pgTable */
89 //  PageTable* pgTable = u.u_MemoryDescriptor.m_UserPageTableArray;
90 //  u.u_MemoryDescriptor.Initialize();
91 //  /* 父进程的相对地址映照表拷贝给予进程，共两张页表的大小 */
92 //  if ( NULL != pgTable ) /*针对0#进程没有虚实地址映照表*/
93 //  {
94 //      //u.u_MemoryDescriptor.Initialize();
95 //      Utility::MemCopy((unsigned long)pgTable, (unsigned long)u.u_MemoryDes
96 //  }
97

```

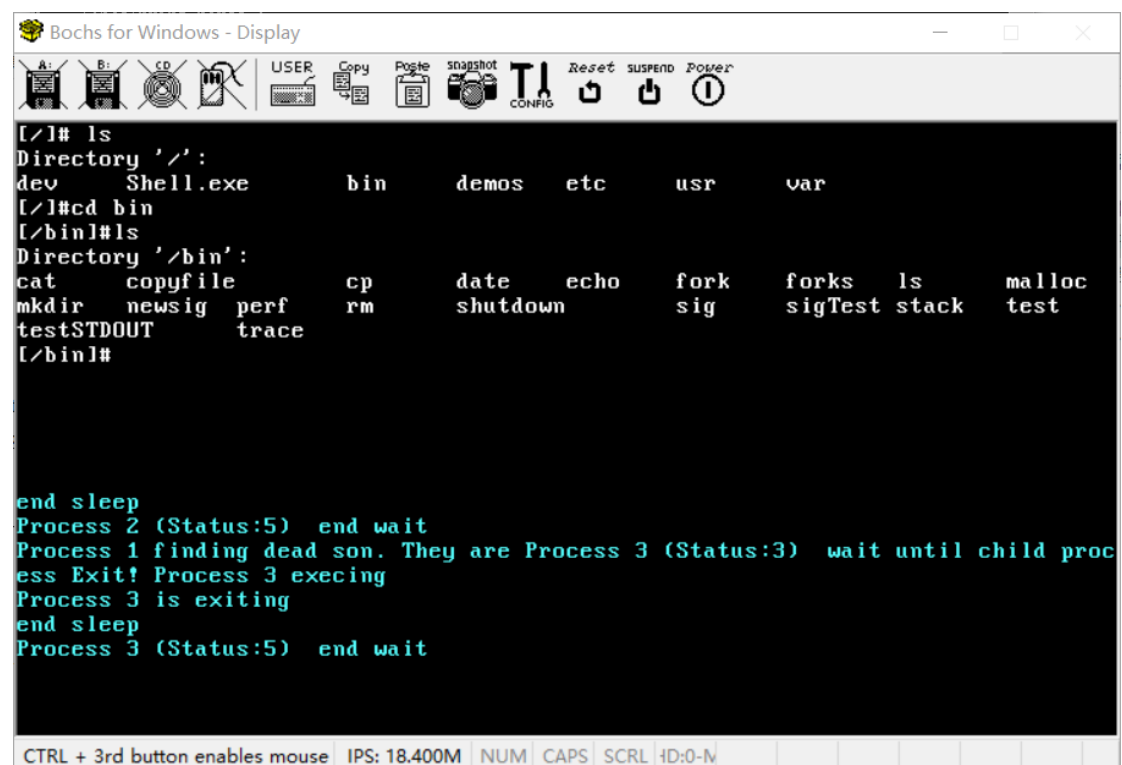
随后子进程调度上台时，执行 ProcessManager::Swch()，执行

MapToPageTable(), 将记录的 6 条数据转化为页表, 随后子进程返回用户态。

```
172
173 /* 恢复被保存进程的现场 */
174 X86Assembly::CLI();
175 SwtchUStruct(selected);
176
177 RetU();
178 X86Assembly::STI();
179
180 User& newu = Kernel::Instance().GetUser();
181
182 newu.u_MemoryDescriptor.MapToPageTable();
183
184 /*
185  * If the new process paused because it was
186  * swanned out, set the stack level to the last call
```

五. 实验结果

● 成功初始化



The screenshot shows the Bochs for Windows - Display window. The command prompt displays the following output:

```
[/]# ls
Directory '/':
dev  Shell.exe  bin  demos  etc  usr  var

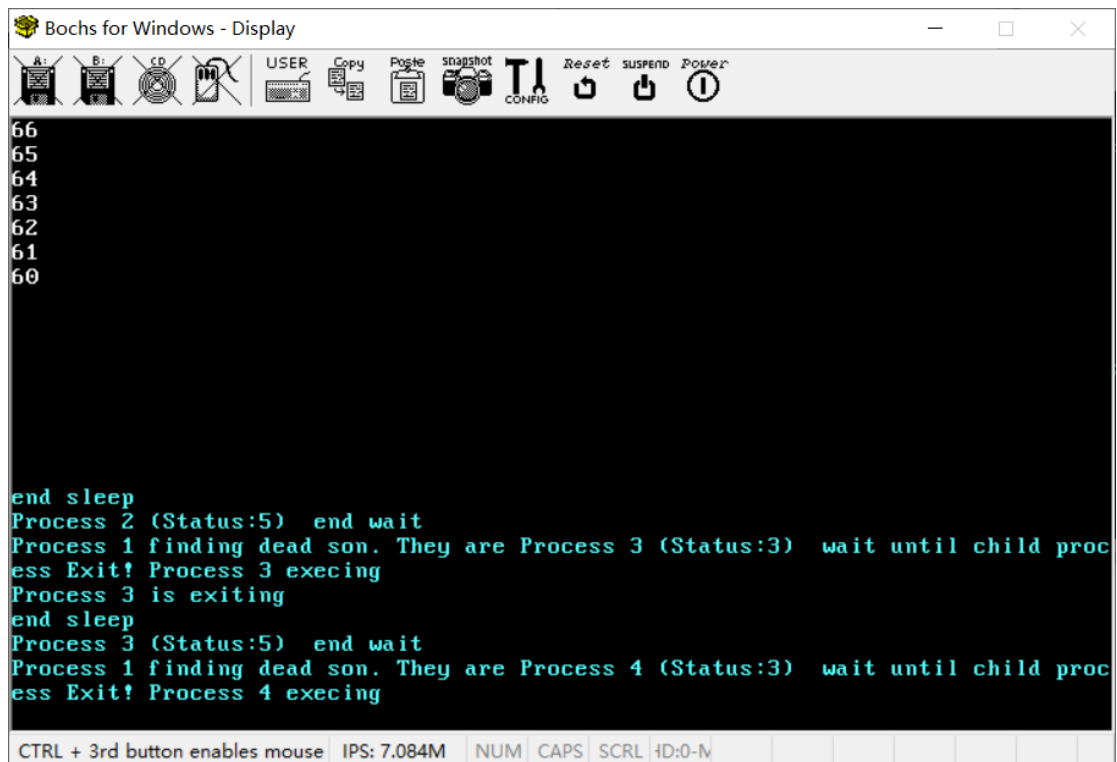
[/]#cd bin
[/bin]#ls
Directory '/bin':
cat  copyfile  cp  date  echo  fork  forks  ls  malloc
mkdir  newsig  perf  rm  shutdown  sig  sigTest  stack  test
testSTDOUT  trace

[/bin]#

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

The status bar at the bottom of the window shows: CTRL + 3rd button enables mouse | IPS: 18.400M | NUM | CAPS | SCRL | ID:0-N

运行 stack

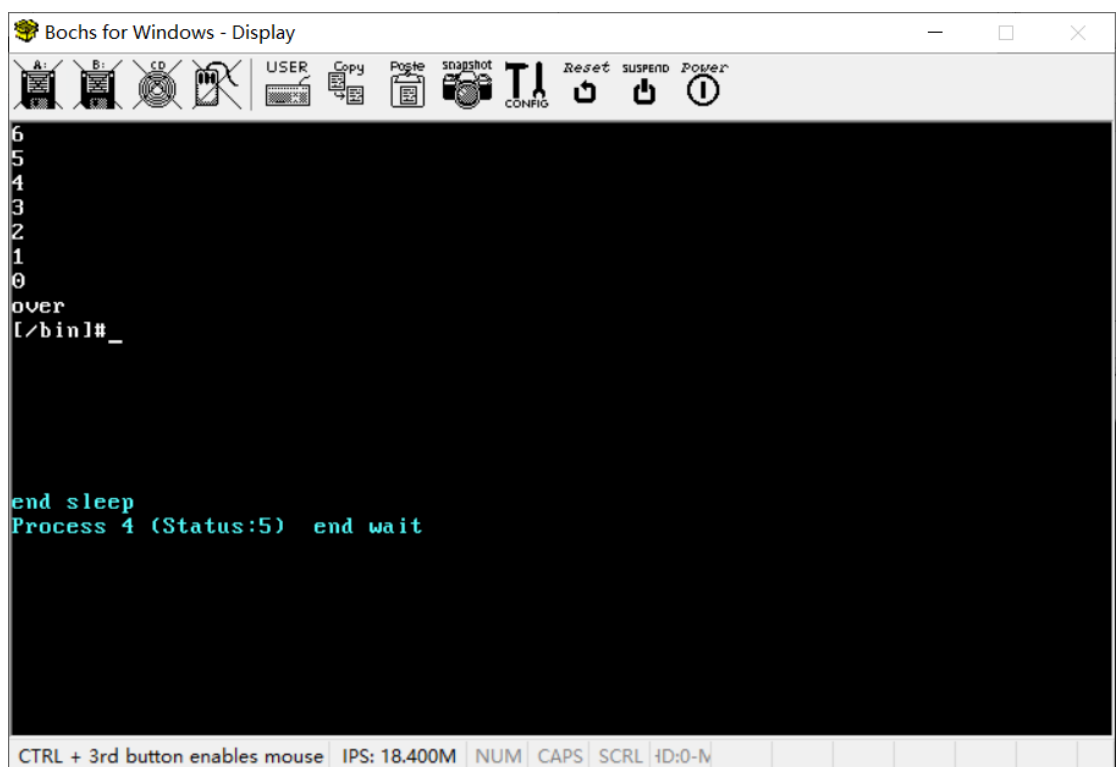


Bochs for Windows - Display

66
65
64
63
62
61
60

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

CTRL + 3rd button enables mouse IPS: 7.084M NUM CAPS SCRL ID:0-N



Bochs for Windows - Display

6
5
4
3
2
1
0
over
[/bin]#_

```
end sleep  
Process 4 (Status:5) end wait
```

CTRL + 3rd button enables mouse IPS: 18.400M NUM CAPS SCRL ID:0-N

六. 去除相对虚实地址映射表的必要性

相对虚实地址映射表在数据上是冗余的，它和用户页表同样占据 2 个连续的物理页框，表达的内容完全一致，唯一的区别在于用户页表在每个新的进程上台后重写（所有进程使用的页表位于同一块内核空间），但相对虚实地址映射表始

终保存在内核空间中直到进程退出。

另外，由于 UNIX V6++物理内存的存储是连续的，因此用户页表的 entry 很容易由 `m_TextSize`、`m_DataSize`、`m_StackSize` 和 `m_TextStartAddress`、`m_DataStartAddress`、`USER_SPACE_SIZE` 计算得到。

因此可以在每个进程调度上台时只重写页表。

这样去除相对虚实地址映射表有两个好处：释放内核内存空间，每个进程减少内存开销 $2*4K$ （两个物理页框）；减少访存次数，在进程 `fork`、`exec` 后减少写相对虚实地址映射表的访存，在进程调度上台时减少读相对虚实地址映射表的访存。

七. 设计文档：重新规划进程的核心态地址空间

（一）主要思想

修改 UNIX V6++内核，改为 Linux 式页目录、页表分配机制：进程 `fork`、`exec` 后在内核中直接构造新进程的页目录、用户页表，同时保留老进程的页目录、用户页表；在进程退出后，回收页目录、用户页表。在进程调度上台时，只需切换 `cr3` 寄存器就可以切换用户空间管理结构，无需重写页表。

（二）User 类的修改

- 新建 `u_PageDirectory`，`u_UserPageTable` 指针，记录每个进程对应的页目录、页表指针。进程上台时，修改 `cr3` 寄存器和 `Machine` 类。
- 新进程 `fork`、`exec` 时，应分配新的页目录、页表，并写道 `User` 类中。新进程调度上台时，应用 `User` 类写 `Machine` 类，修改用户内存映射。

```
1. void MemoryDescriptor::MapToPageTable()  
2. {  
3.     User& u = Kernel::Instance().GetUser();  
4.  
5.     //if(u.u_MemoryDescriptor.m_UserPageTableArray == NULL)  
6.     //     return;  
7.  
8.     PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray();  
9.     .....
```

例如在 `MemoryDescriptor::MapToPageTable()` 函数中，上面这步获取用户页表指针的部分应该修改。当前进程若没有分配页目录、页表，需要报错；若已有页目录、页表，直接写入页表即可。

（三）Machine 类的修改

- 页目录指针 `m_PageDirectory`、用户页表指针 `m_UserPageTable` 在进程调度

上台时应该修改。（但由于内核空间是共用的， m_KernelPageTable 只需要复制内核对应的唯一一个内核页表即可）

```
1.   PageDirectory* m_PageDirectory;
2.   PageTable*      m_KernelPageTable;
3.   PageTable*      m_UserPageTable;
```

- 页目录、核心态页表、用户态页表在物理内存中的起始地址由于进程改变而改变，这部分数据作废

```
1.   /* 页目录、核心态页表、用户态页表在物理内存中的起始地址 */
2.   static const unsigned long PAGE_DIRECTORY_BASE_ADDRESS = 0x200000;
3.   static const unsigned long KERNEL_PAGE_TABLE_BASE_ADDRESS = 0x201000;
4.   static const unsigned long USER_PAGE_TABLE_BASE_ADDRESS = 0x202000;
```

- 修改 InitPageDirectory、InitUserPageTable 函数

```
1.   PageDirectory* pPageDirectory = (PageDirectory*)(PAGE_DIRECTORY_BASE_ADDRESS +
    KERNEL_SPACE_START_ADDRESS);
```

例如以上这句代码，PAGE_DIRECTORY_BASE_ADDRESS 对应不再固定为 0x200# 页表，改为由 KernelPageManager.AllocMemory 分配空间。函数 InitUserPageTable 同理。

（四）ProcessManager 类的修改

- NewProc 函数是 fork 过程中新建进程的方法，原先的 fork 过程子进程和父进程使用同一物理空间的同样的页表，因此不需要复制页表。现在由于页目录、页表与进程一一对应，页目录、页表的拷贝是必须的了。

```
1.   u.u_MemoryDescriptor.Initialize();
2.   /* 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 */
3.   if ( NULL != pgTable ) /*针对 0#进程没有虚实地址映射表*/
4.   {
5.       //u.u_MemoryDescriptor.Initialize();
6.       Utility::MemCopy((unsigned long)pgTable, (unsigned
    long)u.u_MemoryDescriptor.m_UserPageTableArray, sizeof(PageTable) *
    MemoryDescriptor::USER_SPACE_PAGE_TABLE_CNT);
7.   }
```

因此在原先分配和拷贝相对虚实地址映射表的位置，改为分配和拷贝页目录、页表。

- Swtch 函数是进程切换时的方法，需要切换 cr3 寄存器和 Machine 类的用户页表。如下，MapToPageTable() 函数没有意义了，可以去掉

```
1.   User& newu = Kernel::Instance().GetUser();
2.
3.   newu.u_MemoryDescriptor.MapToPageTable();
```

- Exec 函数是执行新的应用程序的方法，需要利用原先的页目录、用户页表写新的页目录、用户页表，并写 cr3 寄存器。所以应该在

EstablishUserPageTable 函数前重新分配用户页表（去除多余的页表），修改对应的页目录，最后 EstablishUserPageTable 写页表。

```
1.  /* 根据正文段、数据段、堆栈段长度建立相对地址映照表，并加载到页表中 */
2.  u.u_MemoryDescriptor.EstablishUserPageTable(parser.TextAddress,
        parser.TextSize, parser.DataAddress, parser.DataSize, parser.StackSize);
```

（五）修改的好处

进程上台后，不需要重复地刷新页表，只用修改 cr3 寄存器，进程切换效率提高。同时，新的页目录和页表相比之前的相对虚实地址映射表并没有增大很多，对内存的消耗增加不大。

（六）可能增加的功能

Linux 系统有写时拷贝的机制，在 fork 之后，用户内存空间并不直接复制一份，而是等待即将执行的子进程（或后续上台的父进程）修改内存时再复制，修改前可以认为内存是只读的。页表和页目录同理：当需要增减页表项时，再复制页表。这么做的好处主要在于，当 fork 后紧接着执行 exec 时，原先复制的内存随即被清除，现在使用写时拷贝的机制，节省了一次复制的时间。