

# Unix v6++中有趣的一次实验

---

汇报人：王家祺

指导教师：邓蓉

---

## 一. 实验内容

一句话：消除unix v6++中每个进程的用户相对虚实映射表，让内存映射更加合理且高效！

## 二. 实验工具+环境

Eclipse+unix v6++系统

关掉防火墙+任何杀毒工具（某60）

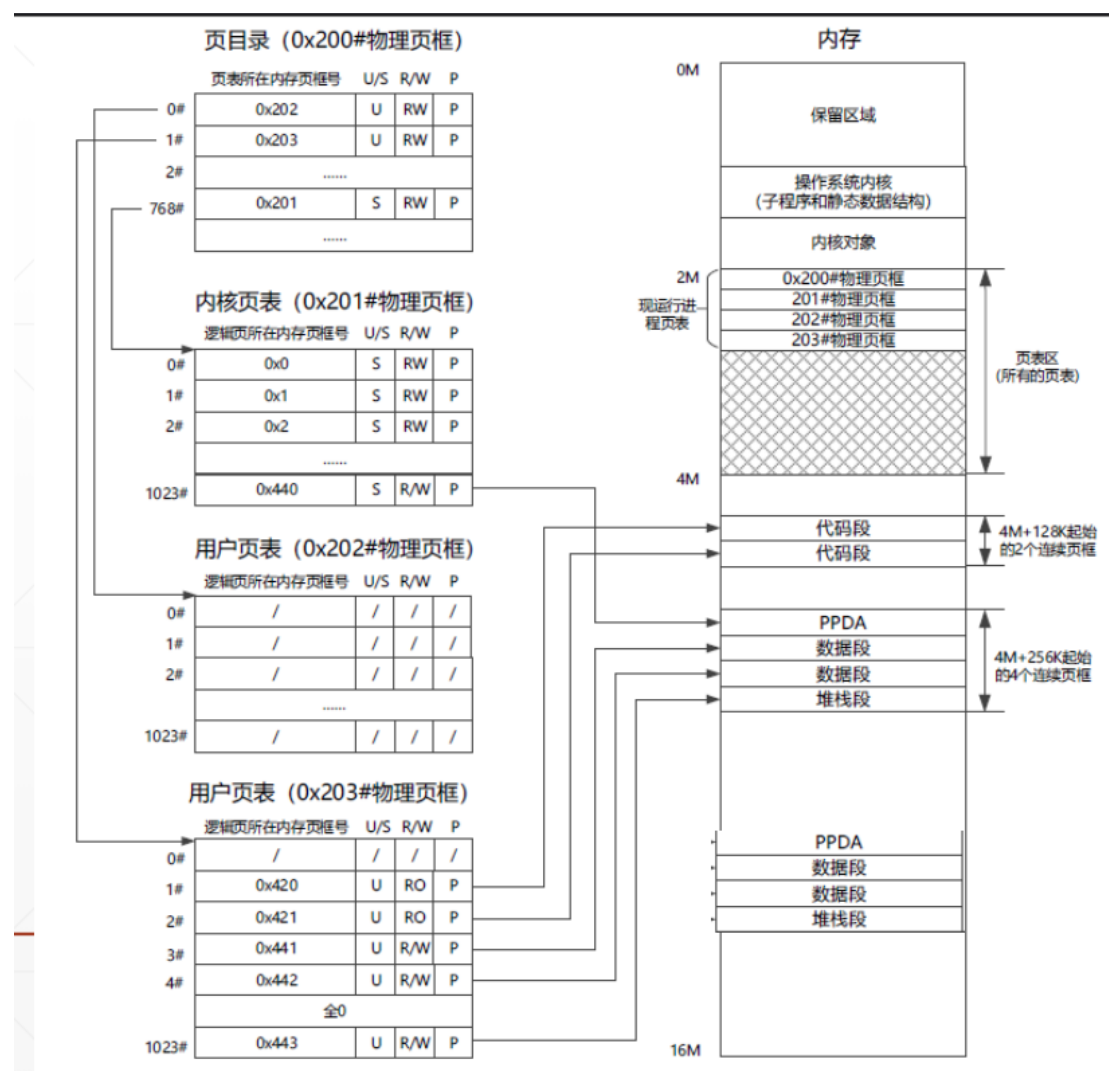
## 三. 什么是相对虚实映射表，为什么要消除？

unix v6++系统中，每个进程会有一个**MemoryDescriptor**对象，这个数据结构存储了一个很关键的信息：**该进程代码段、数据段、堆栈段的长度**。其目的主要是用来写系统页表，而且写的规则也很明确--根据代码段和数据段各自的相对基址号+(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。

**相对基址号**：是相对于代码段起始页框号的偏移量。其余有效PTE，base是相对于可交换部分起始页框号的偏移量。

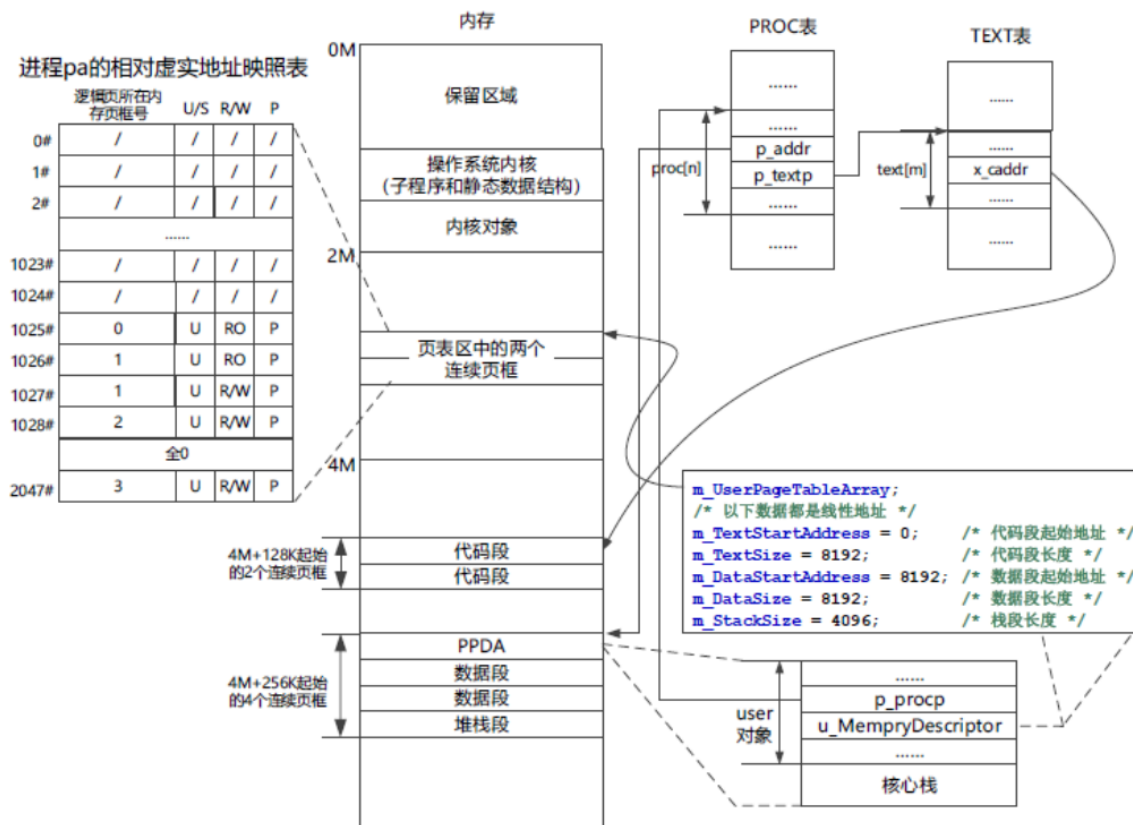
**各自的实际物理地址**：代码段（只可读）：`p_textp->x_caddr` 可交换部分（可读可写）：`p_addr`。

**系统页表**：不详细介绍了，课上也肯定讲过，我就来张大图。

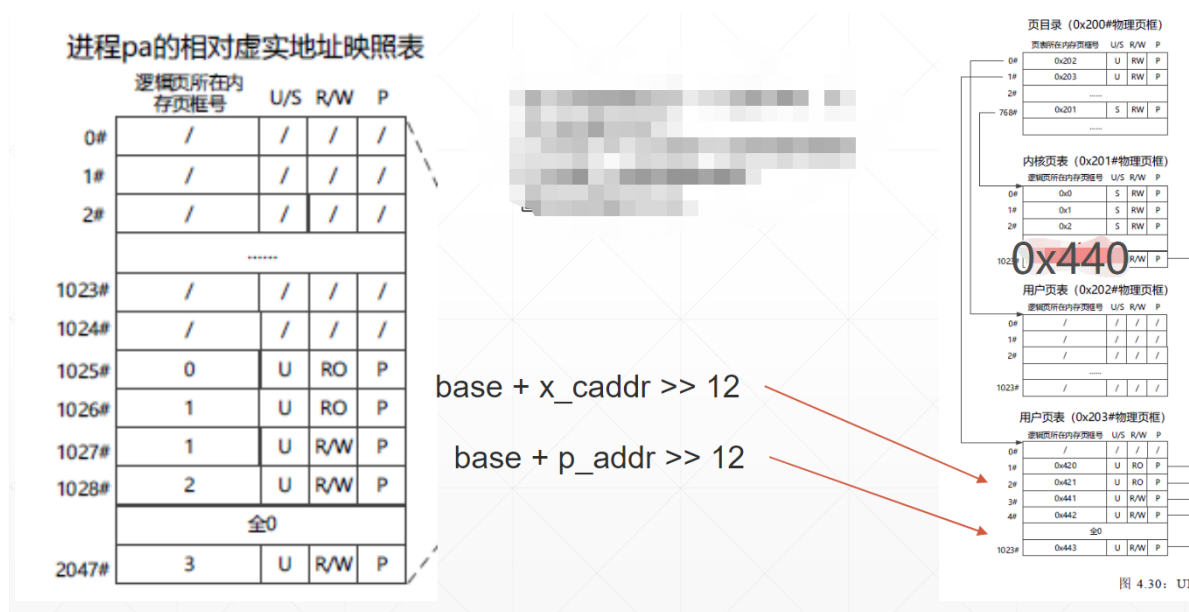


如同前面分析，似乎只需要前面所提到的东西我们就能写掉上图中的0x202和0x203页表。

但是呢，unix v6++有一个多此一举的数据结构：相对虚实映射表 (PageTable\* m\_UserPageTableArray) 他的做法是根据这个相对虚实映射表来写上图所示两张系统页表，下面放一张相对虚实映射表

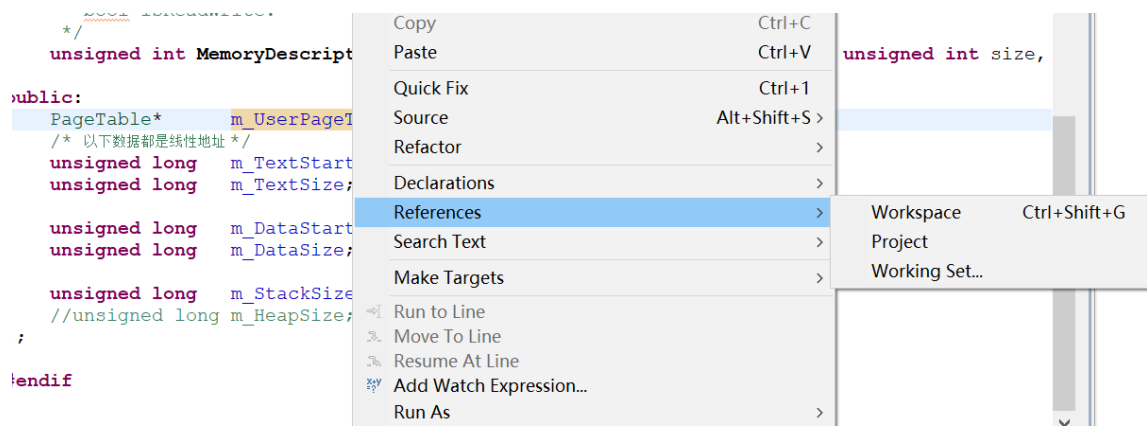


映射原理如下图，显然多此一举，不如我们前面所说的方法简洁。



## 四. 实验方法

首先查找所有引用到 m\_UserPageTableArray 对象的地方！查找方法：



18处match分布在 2 个源文件MemoryDescriptor.cpp和ProcessManager.cpp。

### MemoryDescriptor.cpp

```
1 // 初始化MemoryDescriptor对象的时候，分配2个页框装相对虚实地址映射表。计算其起始地址，赋值m_UserPageTableArray。
2 所以我们修改的时候，直接不要申请空间，只留4字节的指针。
3 void MemoryDescriptor::Initialize() {
4     KernelPageManager& kernelPageManager =
5         Kernel::Instance().GetKernelPageManager();
6     /* m_UserPageTableArray需要把AllocMemory()返回的物理内存地址 + 0xC0000000 */
7     // this->m_UserPageTableArray = (PageTable*)
8     (kernelPageManager.AllocMemory(
9         sizeof(PageTable) * USER_SPACE_PAGE_TABLE_CNT,
10         + Machine::KERNEL_SPACE_START_ADDRESS);
11     this->m_UserPageTableArray=NULL;
12 }
```

```
1 // 释放MemoryDescriptor对象的时候，释放相对虚实地址映射表。
2 所以这个动不动无所谓
3 void MemoryDescriptor::Release() {
4     KernelPageManager& kernelPageManager =
5         Kernel::Instance().GetKernelPageManager();
6     if (this->m_UserPageTableArray) {
7         kernelPageManager.FreeMemory(
8             sizeof(PageTable) * USER_SPACE_PAGE_TABLE_CNT,
9             (unsigned long) this->m_UserPageTableArray
10             - Machine::KERNEL_SPACE_START_ADDRESS);
11     this->m_UserPageTableArray = NULL;
12 }
13 }
```

```
1 //写页表，virtualAddress>>12开始；size 上取整，写这么多PTE。phyPageIdx是base，
2 isReadWrite是RO或RW。
3 *****
4     注意EstablishUserPageTable()函数调用它，需要理解这个函数功能，根本不需要它所以我直接注释掉里面内容只返回一个无效的phyPageIdx，但是是需要理解这个功能才知道后面怎么修改。
5     *****
6     *****
7
8 unsigned int MemoryDescriptor::MapEntry(unsigned long virtualAddress,
9     unsigned int size, unsigned long phyPageIdx, bool isReadWrite) {
10     unsigned long address = virtualAddress - USER_SPACE_START_ADDRESS;
```

```

11 // //计算从pagetable的哪一个地址开始映射
12 // unsigned long startIdx = address >> 12;
13 // unsigned long cnt = (size + (PageManager::PAGE_SIZE - 1))
14 //     / PageManager::PAGE_SIZE;
15 //
16 // PageTableEntry* entrys = (PageTableEntry*) this->m_UserPageTableArray;
17 // for (unsigned int i = startIdx; i < startIdx + cnt; i++, phyPageIdx++) {
18 //     entrys[i].m_Present = 0x1;
19 //     entrys[i].m_ReadWriter = isReadWrite;
20 //     entrys[i].m_PageBaseAddress = phyPageIdx;
21 // }
22     return phyPageIdx;
23 }
24

```

```

1 //也是注释掉里面的内容 用不到
2 PageTable* MemoryDescriptor::GetUserPageTableArray() {
3 // return this->m_UserPageTableArray;
4     return NULL;
5 }

```

```

1 //也是注释掉里面的内容 用不到
2 void MemoryDescriptor::ClearUserPageTable() {
3 // User& u = Kernel::Instance().GetUser();
4 // PageTable* pUserPageTable = u.u_MemoryDescriptor.m_UserPageTableArray;
5 //
6 // unsigned int i;
7 // unsigned int j;
8 //
9 // for (i = 0; i < Machine::USER_PAGE_TABLE_CNT; i++) {
10 //     for (j = 0; j < PageTable::ENTRY_CNT_PER_PAGETABLE; j++) {
11 //         pUserPageTable[i].m_Entrys[j].m_Present = 0;
12 //         pUserPageTable[i].m_Entrys[j].m_ReadWriter = 0;
13 //         pUserPageTable[i].m_Entrys[j].m_UserSupervisor = 1;
14 //         pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = 0;
15 //     }
16 // }
17 }

```

```

1 //已知虚空间分布，写相对虚实地址映射表
2 *****
3 *****
4     更新m_TextSize m_DataSize m_StackSize,注释掉写相对虚实地址的部分,调用this-
>MapToPageTable();
5 *****
6 *****
7 bool MemoryDescriptor::EstablishUserPageTable(unsigned long
textVirtualAddress,
8     unsigned long textSize, unsigned long dataVirtualAddress,
9     unsigned long dataSize, unsigned long stackSize) {
10     User& u = Kernel::Instance().GetUser();
11
12     /* 如果超出允许的用户程序最大8M的地址空间限制 */
13     if (textSize + dataSize + stackSize + PageManager::PAGE_SIZE
14         > USER_SPACE_SIZE - textVirtualAddress) {

```

```

15         u.u_error = User::ENOMEM;
16         Diagnose::write("u.u_error = %d\n", u.u_error);
17         return false;
18     }
19     m_TextSize=textSize;
20     m_DataSize=dataSize;
21     m_StackSize=stackSize;
22     // this->ClearUserPageTable();
23
24     // /* 以相对起始地址phyPageIndex为0, 为正文段建立相对地址映照表 */
25     // unsigned int phyPageIndex = 0;
26     // phyPageIndex = this->MapEntry(textVirtualAddress, textSize,
27     phyPageIndex,
28     //         false);
29     //
30     // /* 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
31     址映照表 */
32     // phyPageIndex = 1;
33     // phyPageIndex = this->MapEntry(dataVirtualAddress, dataSize,
34     phyPageIndex,
35     //         true);
36     //
37     // /* 紧跟着数据段之后, 为堆栈段建立相对地址映照表 */
38     // unsigned long stackStartAddress = (USER_SPACE_START_ADDRESS
39     //         + USER_SPACE_SIZE - stackSize) & 0xFFFFF000;
40     // this->MapEntry(stackStartAddress, stackSize, phyPageIndex, true);
41
42     /* 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
43     立用户态内存区的页表映射 */
44     this->MapToPageTable();
45     return true;
46 }

```

下面这个函数需要大改, 也是本次实验的核心地方, 思想前面已经介绍过, 放图, 不可复制。

```

1 void MemoryDescriptor::MapToPageTable() {
2     User& u = Kernel::Instance().GetUser();
3     PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray();
4     unsigned int textAddress = 0;
5     if (u.u_procp->p_textp != NULL) {
6         textAddress = u.u_procp->p_textp->x_caddr;
7     }
8     unsigned int tstart_index = 0, dstart_index = 1; //代码段和数据段起始偏移量
9     //下面计算代码段 数据段和堆栈段各自占多少个页框
10    unsigned int text_len = (m_TextSize + (PageManager::PAGE_SIZE - 1))
11        / PageManager::PAGE_SIZE;
12    unsigned int data_len = (m_DataSize + (PageManager::PAGE_SIZE - 1))
13        / PageManager::PAGE_SIZE;
14    unsigned int stack_len = (m_StackSize + (PageManager::PAGE_SIZE - 1))
15        / PageManager::PAGE_SIZE;
16    unsigned int dataidx=0; //data段的页框号计数
17    for (unsigned int i = 0; i < Machine::USER_PAGE_TABLE_CNT; i++) {
18        for (unsigned int j = 0; j < PageTable::ENTRY_CNT_PER_PAGETABLE; j++) {
19            pUserPageTable[i].m_Entrys[j].m_Present = 0; //先清0
20            if (1 == i) { //只刷第二个用户页表 第一个用户页表为空
21                if (1<=j&&j<= text_len) { //代码段 不可写
22                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
23                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 0;
24                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j-1
25                        + tstart_index + (textAddress >> 12);
26
27                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
28                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 0;
29                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = j-1
30                        + tstart_index + (textAddress >> 12);
31                }
32                else if (j > text_len && j <= text_len + data_len) { //数据段 可写
33                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
34                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
35                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress = dataidx+
36                        dstart_index
37                        + (u.u_procp->p_addr >> 12);
38                    dataidx++;
39                }
40                //堆栈段 stack开始 1024-stack_len
41                else if(j>=PageTable::ENTRY_CNT_PER_PAGETABLE-stack_len){
42                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
43                    pUserPageTable[i].m_Entrys[j].m_ReadWriter = 1;
44                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress =
45                        dataidx+dstart_index
46                        + (u.u_procp->p_addr >> 12);
47                    dataidx++;
48                }
49            }
50        }
51    }
52    FlushPageDirectory();
53 }
54
55 pUserPageTable[0].m_Entrys[0].m_Present = 1;
56 pUserPageTable[0].m_Entrys[0].m_ReadWriter = 1;
57 pUserPageTable[0].m_Entrys[0].m_PageBaseAddress = 0;
58
59 FlushPageDirectory();
60 }

```

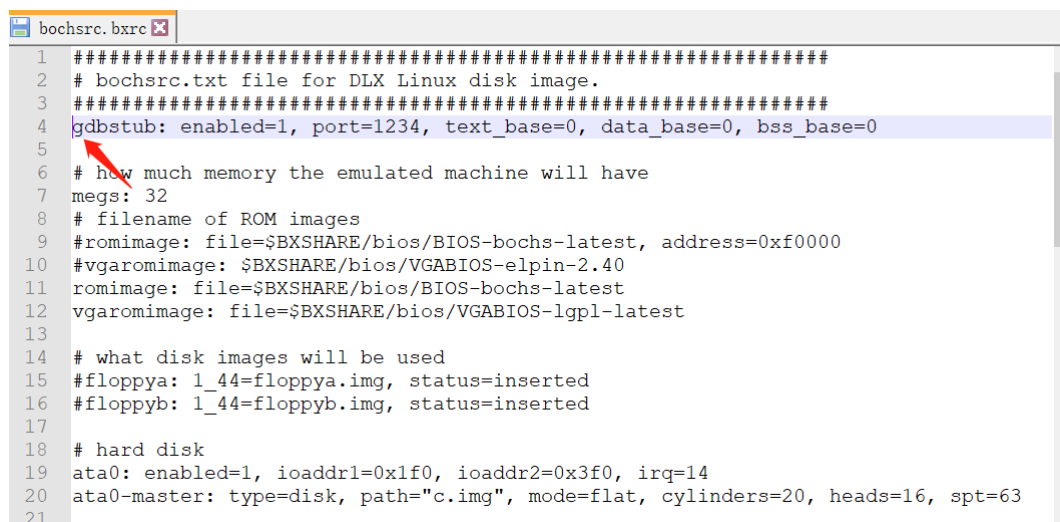
## ProcessManager.cpp

```
1 NewProc ()，创建子进程。
2 过程开始的时候，为子进程申请两张虚实地址映射表（84行）。
3 将父进程的虚实地址映射表复制给子进程（88行）。
4 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
  父进程的相对虚实映射表指针：u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;
  （127行）
5
6 *****
7 不需要改，当然想改也是有可改的地方的，留给大家自己尝试。
8 *****
```

## 五. 实验结果

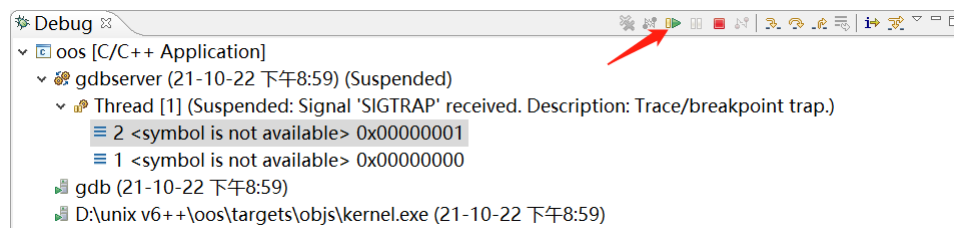
大功告成，开始测试。

注意打开注释，然后先运行unix v6++\oos\tools里面的OOS Command Prompt，运行命令run，最后在eclipse里面点debug。

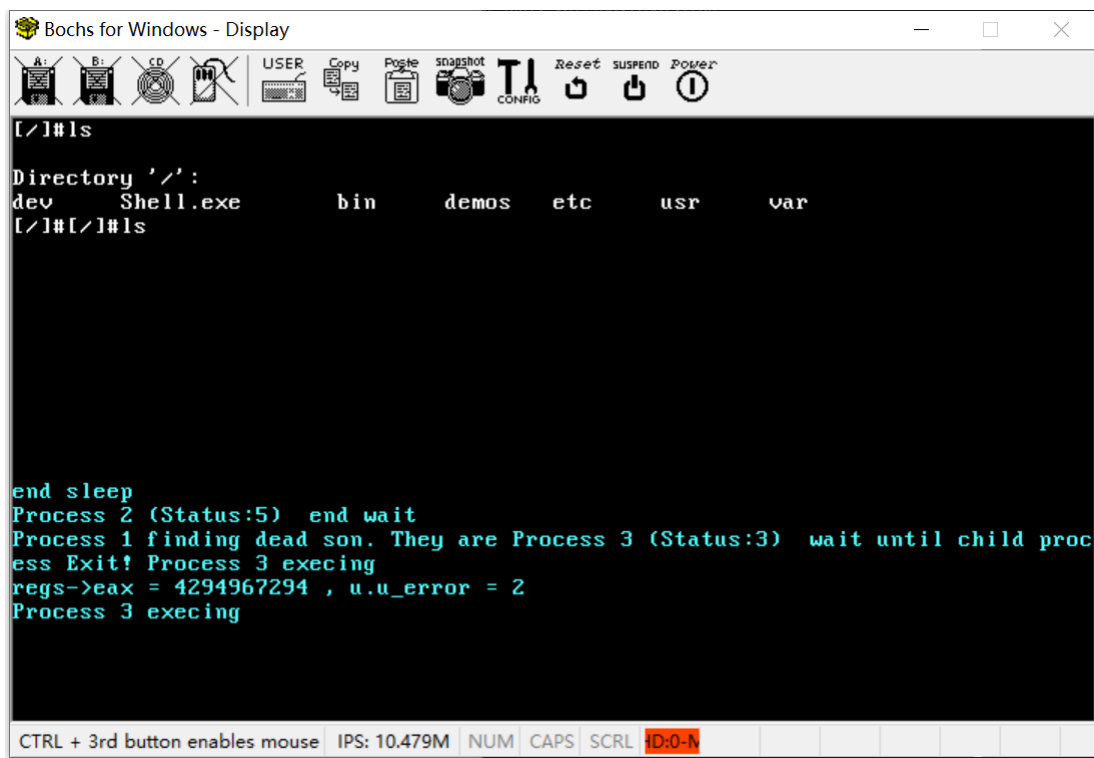


```
bochsrc.bxrc
1 #####
2 # bochsrc.txt file for DLX Linux disk image.
3 #####
4 gdbstub: enabled=1, port=1234, text_base=0, data_base=0, bss_base=0
5
6 # how much memory the emulated machine will have
7 megs: 32
8 # filename of ROM images
9 #romimage: file=$BXS SHARE/bios/BIOS-bochs-latest, address=0xf0000
10 #vgaromimage: $BXS SHARE/bios/VGABIOS-elpin-2.40
11 romimage: file=$BXS SHARE/bios/BIOS-bochs-latest
12 vgaromimage: file=$BXS SHARE/bios/VGABIOS-lgpl-latest
13
14 # what disk images will be used
15 #floppya: 1_44=floppya.img, status=inserted
16 #floppyb: 1_44=floppyb.img, status=inserted
17
18 # hard disk
19 ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
20 ata0-master: type=disk, path="c.img", mode=flat, cylinders=20, heads=16, spt=63
21
```

调试的时候一直点resume即可，点不动为止，就可以进入到系统进行各种各样的测试。







## 六. 实验收获

修改源码确实很好玩！