操作系统　第四章　进程管理

4.6 exec，exit 和 wait

同济大学计算机系

Part 1、Unix 系统加载应用程序

创建一个子进程，让它执行exec系统调用，承担执行应用程序的任务

```
int main( int argc, char *argv[] )
{
   ......
   int i ;
   if( i = fork( ))
         ......
   else
         exec ( “gcc” , arg1, arg2.....);
}
```

子进程要执行的程序

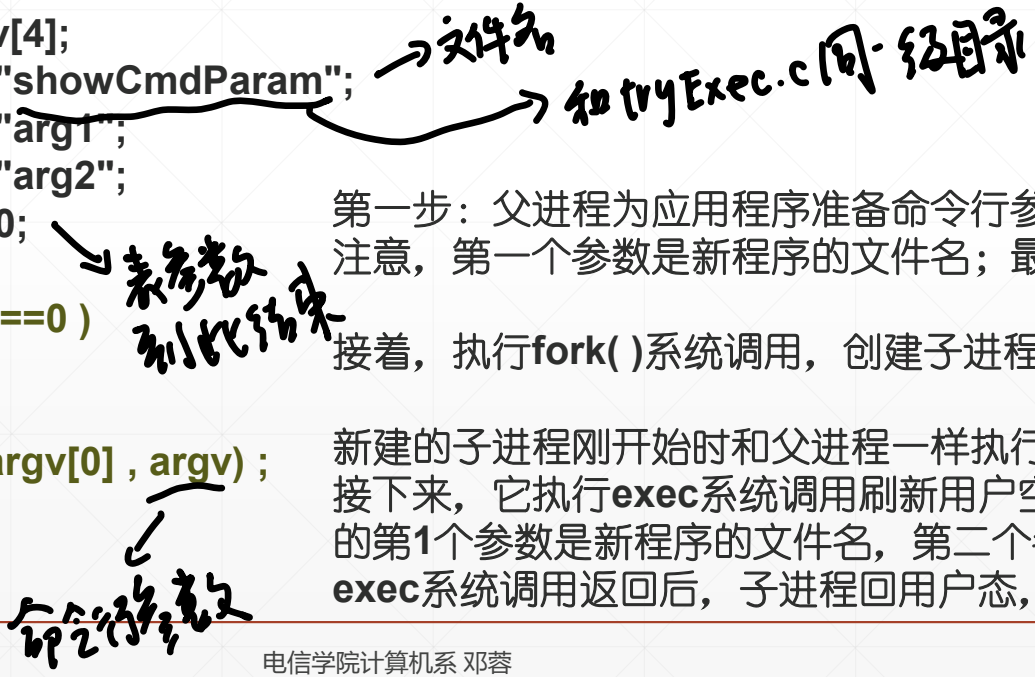
程序的命令行参数

gcc程序

```
int main( int argc, char *argv[] )
{
   ......
}
```

exec的使用方法

```
#include <stdio.h>
#include <stdlib.h>

int main1( )    // tryExec.c
{
    char* argv[4];
    argv[0] = "showCmdParam";
    argv[1] = "arg1";
    argv[2] = "arg2";
    argv[3] = 0;

    if ( fork( ) ==0 )
            ;
    else
        execv(argv[0] , argv) ;

    exit(0);
}
```



第一步：父进程为应用程序准备命令行参数。
注意，第一个参数是新程序的文件名；最后一个参数是**0**，表示命令行参数结束。

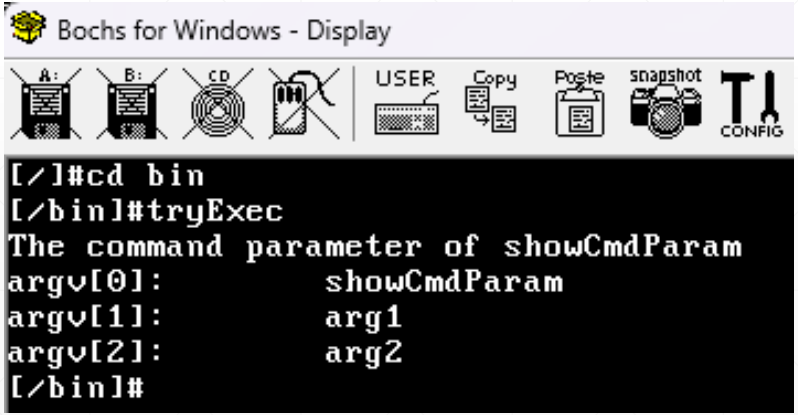
接着，执行**fork( )**系统调用，创建子进程。

新建的子进程刚开始时和父进程一样执行 **tryExec** 程序，是 **tryExec** 进程。
接下来，它执行**exec**系统调用刷新用户空间，装入新程序的图像。**exec** 系统调用的第**1**个参数是新程序的文件名，第二个参数是新程序的命令行参数。
exec系统调用返回后，子进程回用户态，从**main**函数的第一条指令开始执行新程序。

exec的使用方法

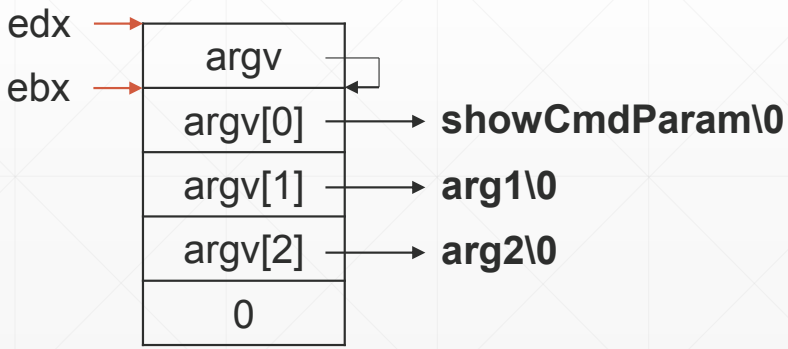
```
#include <stdio.h>
#include <stdlib.h>

int main1(int argc, char *argv[])   // showCmdParam.c
{
    int i;

    printf("The command parameter of showCmdParam\n");

    for(i = 0; i < argc; i++)
        printf("argv[%d]:\t%s\n", i, argv[i]);

    exit(0);
}
```

应用程序，main函数有2个入口参数，argc是命令行参数的数量，argv是命令行参数。本例，新程序输出命令行参数。



exec系统调用的钩子函数

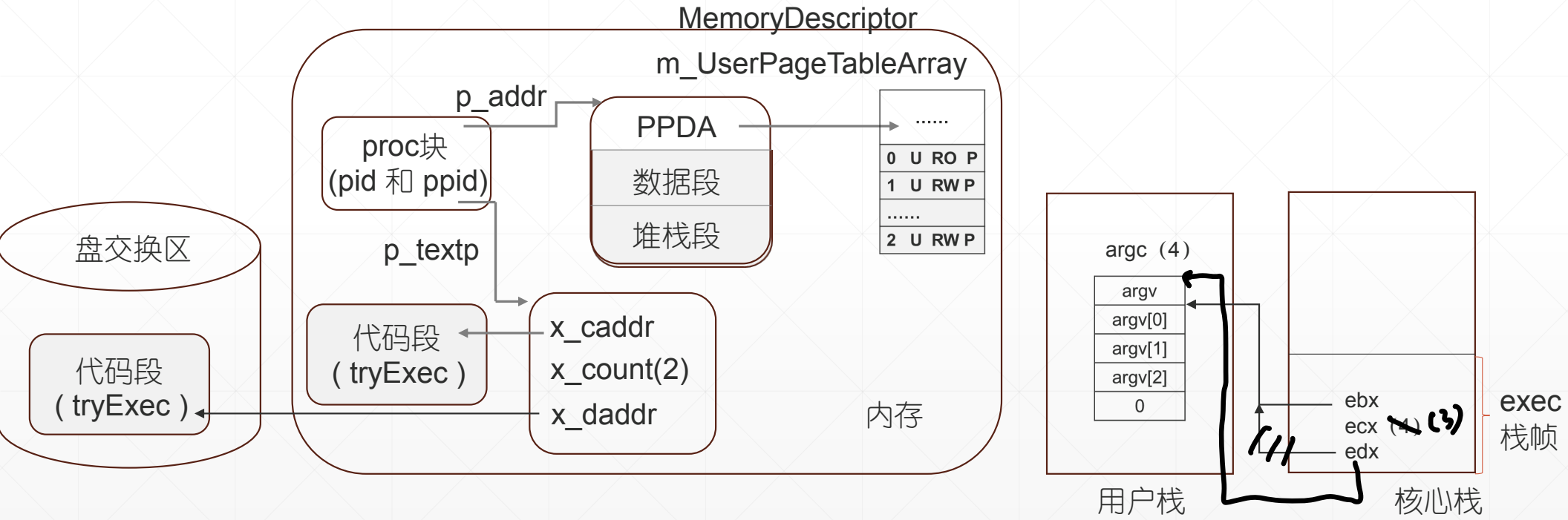
```
int execv(char *pathname, char *argv[])
{
    int res;
    int argc = 0;
    while(argv[argc] != 0)
        argc++;        // 清点命令行参数的数量

    __asm__ volatile ( "int $0x80":
            "=a"(res):"a"(11),"b"(pathname),"c"(argc),"d"(argv));
    if ( res >= 0 )
        return res;
    return -1;
}
```




```
#include <stdio.h>
#include <stdlib.h>

int main1( )    // tryExec.c
{
    char* argv[4];
    argv[0] = "showCmdParam";
    argv[1] = "arg1";
    argv[2] = "arg2";
    argv[3] = 0;

    if ( fork( ) ==0 )
            ;
    else
        execv(argv[0] , argv) ;

    exit(0);
}
```

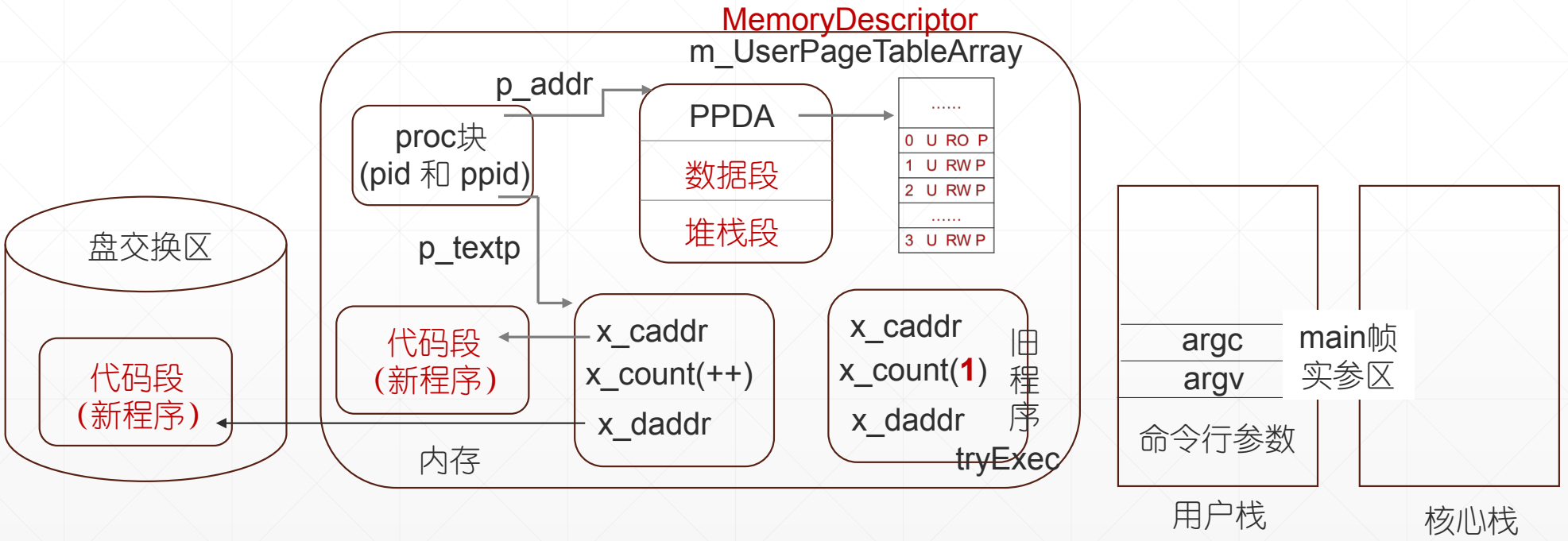
exec系统调用刚开始执行时，子进程的图像

假设 tryExec 程序一页代码，一页数据，一页堆栈

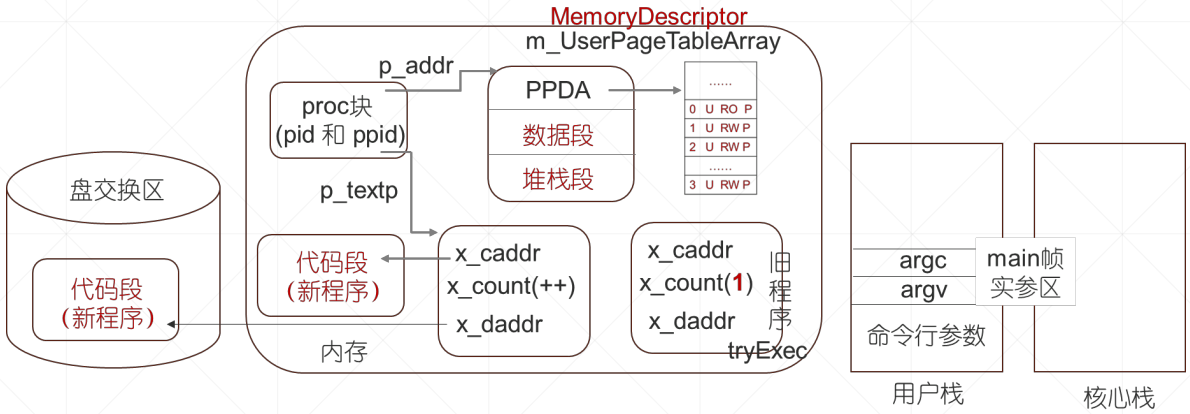


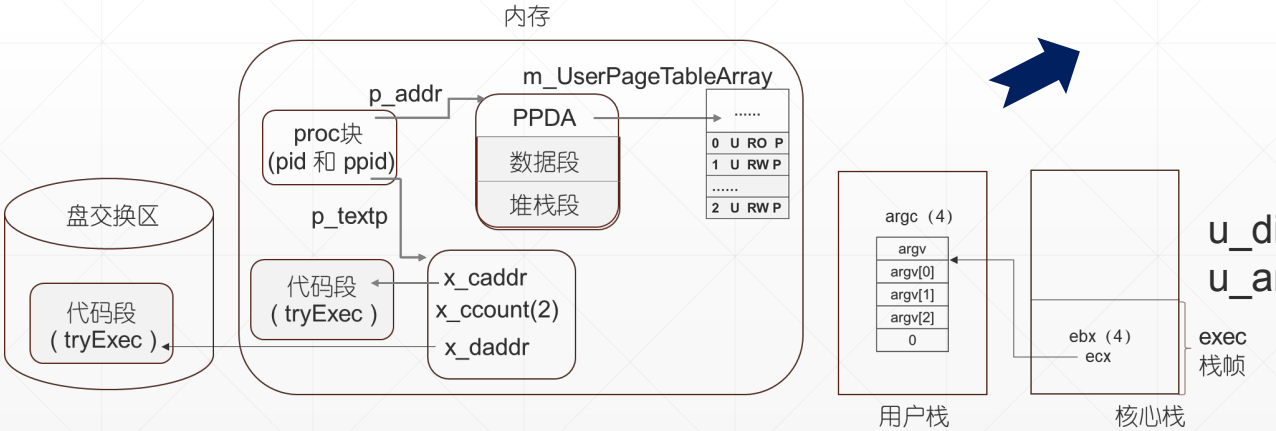
exec系统调用完成时，子进程的图像

假设 新程序 showCmdParam 1页代码，2页数据，1页堆栈



exec系统调用要做的工作





u_dirp → 新程序的文件名 // Trap.c

u_arg[4] → pt_context（硬件保护的系统调用现场）

↓

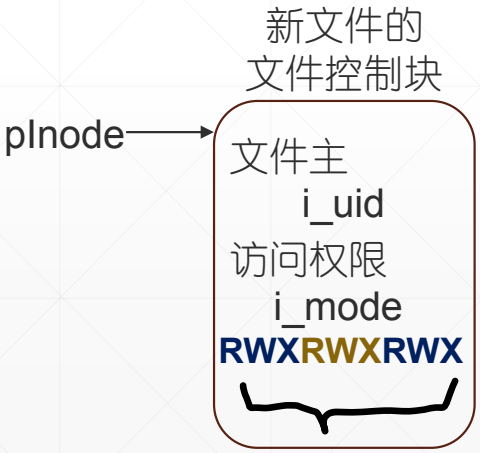
核心栈底部几个寄存器的值

第一步：新程序是磁盘上的普通文件，执行前需要确保文件存在 & 进程有执行权限

```
pInode = fileMgr.NameI(FileManager::NextChar, FileManager::OPEN);
if ( NULL == pInode )
{
        return;
}
```

u_dirp → 新程序的文件名

```
if ( fileMgr.Access(pInode, Inode::IEXEC) || (pInode->i_mode & Inode::IFMT) != 0 )
{
        fileMgr.m_InodeTable->IPut(pInode);
        ……
}
```



R：读权限
W：写权限
X：执行权限

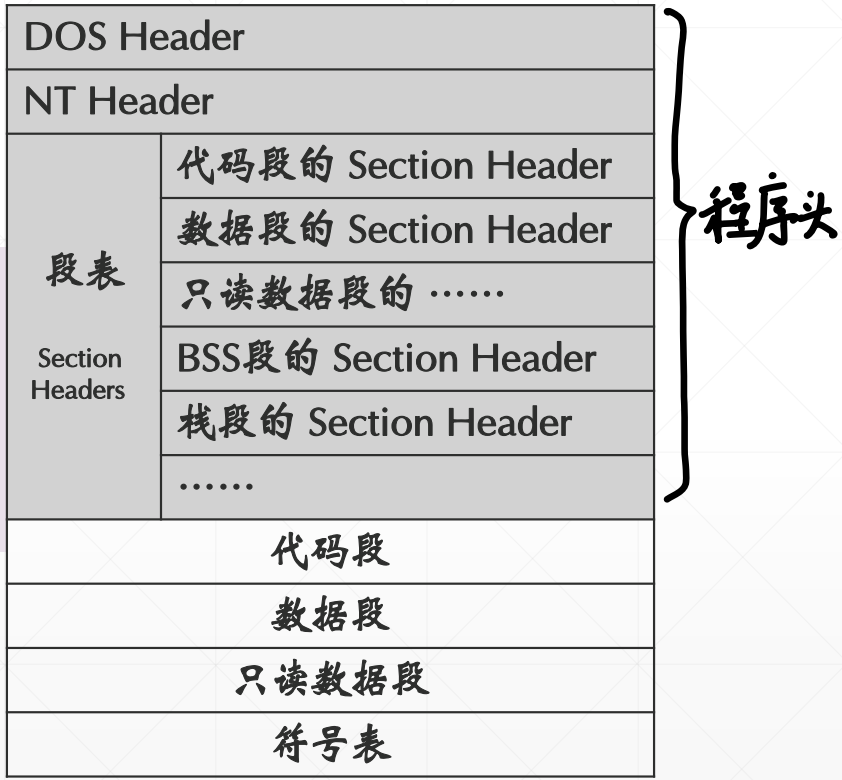
比较用户的三种情况

p_uid == i_uid ？ （**owner**）
Y：第1个X是0，无权限，失败，返回1
1，开始加载磁盘上的新程序
N：p_gid == i_gid ？ （同组用户）
Y：第2个X是0，无权限，失败
1，开始加载磁盘上的新程序
N：第3个X是0，无权限，失败 （**Nobody**）
1，开始加载磁盘上的新程序

第二步：读程序头

```
PEParser parser;
if ( parser.HeaderLoad(pInode)==false )
{
    fileMgr.m_InodeTable->IPut(pInode);
    return;
}
```

可执行文件的格式（PE为例）



```
bool PEParser::HeaderLoad(Inode* p_inode)   // p_inode，可执行文件
{
    ImageDosHeader dos_header;
    User& u = Kernel::Instance().GetUser();
    KernelPageManager& kpm = Kernel::Instance().GetKernelPageManager();

    /*读取dos header*/
    u.u_IOParam.m_Base = (unsigned char*)&dos_header;
    u.u_IOParam.m_Offset = 0;
    u.u_IOParam.m_Count = 0x40;
    p_inode->ReadI();        //文件IO不会因为多次ReadI而增加。有缓存的!

    /*读取nt_Header*/
    //ntHeader = (ImageNTHeader*)(kpm.AllocMemory(ntHeader_size)+0xC0000000);
    u.u_IOParam.m_Base = (unsigned char*)(&this->ntHeader);
    u.u_IOParam.m_Offset = dos_header.e_lfanew;
    u.u_IOParam.m_Count = ntHeader_size;
    p_inode->ReadI();

    if ( ntHeader.Signature!=0x00004550 )
    {
        //kpm.FreeMemory(ntHeader_size, (unsigned long)ntHeader - 0xC0000000 );
        return false;
    }


    /* 原本V6++内核 : 读取Section tables至页表区。这是无奈之举，核心态用不了malloc！！
     * 希望内核用  new 和 free 函数申请动态数组。但现在的new操作符好像不对。先这么着。
     * sectionHeaders = new ImageSectionHeader;
     */
    //sectionHeaders = (ImageSectionHeader*)(kpm.AllocMemory(PageManager::PAGE_SIZE * 2) + 0xC0000000);
    u.u_IOParam.m_Base = (unsigned char*)sectionHeaders;
    u.u_IOParam.m_Offset = dos_header.e_lfanew + ntHeader_size;
    u.u_IOParam.m_Count = section_size * ntHeader.FileHeader.NumberOfSections;
    p_inode->ReadI();
```

一、从文件中读入DOS头

内存首地址
文件中的偏移量
DOS头的长度

二、从文件中读入NT头

三、每个可执行程序都有一个签名，表格式
Unix V6++可以识别PE格式的可执行文件

表明可执行程序格式(签名)

四、从文件中读入段表

4.1 页表区 [2M,4M] 分配 8k 字节连续的物理内存空间

在映射分配单元

4.2 多少个段

P	E	0	0

i386 是 Intel芯片，little endian 小端
所以，字符串“PE00”是整数0x00004550

```
struct ImageSectionHeader
{
    char    Name[8];
    union {
            unsigned long   PhysicalAddress;
            unsigned long   VirtualSize;
    } Misc;
    unsigned long   VirtualAddress;
    unsigned long   SizeOfRawData;
    unsigned long   PointerToRawData;
    unsigned long   PointerToRelocations;
    unsigned long   PointerToLinenumbers;
    unsigned short    NumberOfRelocations;
    unsigned short    NumberOfLinenumbers;
    unsigned long   Characteristics;
};
```

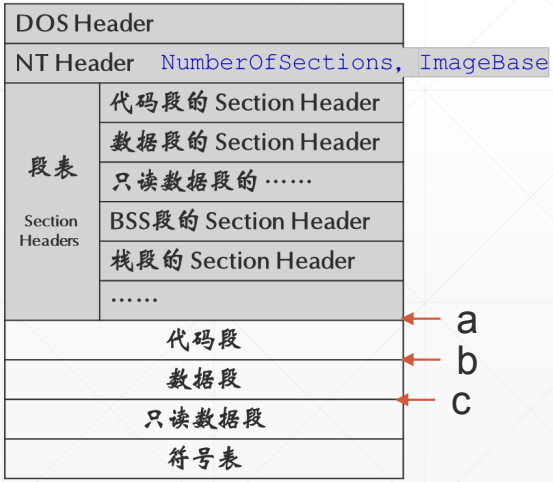
段长

段起始虚地址 = ImageBase+ VirtualAddress

文件中的起始偏移量 和 文件中的长度

```
class PEParser
{
public:
static const unsigned int TEXT_SECTION_IDX = 0;
        static const unsigned int DATA_SECTION_IDX = 1;
        static const unsigned int RDATA_SECTION_IDX = 2;
        static const unsigned int BSS_SECTION_IDX = 3;
        static const unsigned int IDATA_SECTION_IDX = 4;
```

可执行文件的格式（PE为例）



考试要考的哟

段号	VirtualAddress +ImageBase	VirtualSize	PointerToRawData	SizeOfRawData
0	0x401000	0x1000	a	0x1000
1	0x402000	0x2000	b	0x2000
2	0x404000	0x1000	c	**********
3	0x405000	0x1000	null	0
4	----------	0x1000	null	0

内核决定找原在哪

记在 PEParser 对象中

```
    /*
        * @comment 这里hardcode gcc的逻辑
        * section 顺序为 .text->.data->.rdata->.bss
        *
    */
    this->TextAddress =
        ntHeader.OptionalHeader.BaseOfCode + ntHeader.OptionalHeader.ImageBase;
    this->TextSize =
        ntHeader.OptionalHeader.BaseOfData - ntHeader.OptionalHeader.BaseOfCode;

    this->DataAddress =
        ntHeader.OptionalHeader.BaseOfData + ntHeader.OptionalHeader.ImageBase;
    this->DataSize = this->sectionHeaders[this->IDATA_SECTION_IDX].VirtualAddress - ntHeader.OptionalHeader.BaseOfData;

    StackSize = ntHeader.OptionalHeader.SizeOfStackCommit;
    HeapSize = ntHeader.OptionalHeader.SizeOfHeapCommit;

    EntryPointAddress = ntHeader.OptionalHeader.AddressOfEntryPoint +
                        ntHeader.OptionalHeader.ImageBase;

    return true;
}
```

代码段首地址

代码段长度

.....

数据段长度 = 数据段 + 只读数据段 + BSS

程序入口地址（main1 的第 1 条指令）

```
class PEParser
{
public:
    static const unsigned int TEXT_SECTION_IDX = 0;
    static const unsigned int DATA_SECTION_IDX = 1;
    static const unsigned int RDATA_SECTION_IDX = 2;
    static const unsigned int BSS_SECTION_IDX = 3;
    static const unsigned int IDATA_SECTION_IDX = 4;

    static const int ntHeader_size = 0xf8;
    static const int section_size = 0x28;
```

第三步：刷新进程的内存描述符

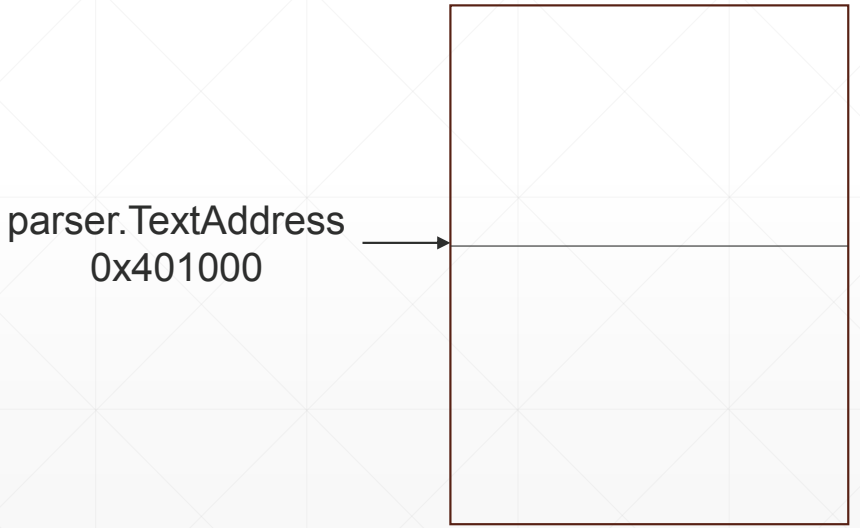
```
/* 获取分析PE头结构得到正文段的起始地址、长度 */
u.u_MemoryDescriptor.m_TextStartAddress = parser.TextAddress;
u.u_MemoryDescriptor.m_TextSize = parser.TextSize;

/* 数据段的起始地址、长度 */
u.u_MemoryDescriptor.m_DataStartAddress = parser.DataAddress;
u.u_MemoryDescriptor.m_DataSize = parser.DataSize;

/* 堆栈段初始化长度 */
u.u_MemoryDescriptor.m_StackSize = parser.StackSize;
```

```
class MemoryDescriptor
{
public:
    /* 用户空间大小 8M 0x0 - 0x800000 2 PageTable */
    static const unsigned int USER_SPACE_SIZE  = 0x800000;
    static const unsigned int USER_SPACE_PAGE_TABLE_CNT = 0x2;
    static const unsigned long USER_SPACE_START_ADDRESS = 0x0;
…
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 */
};
```

0x401000
0x1000
0x402000
0x2000
0x1000

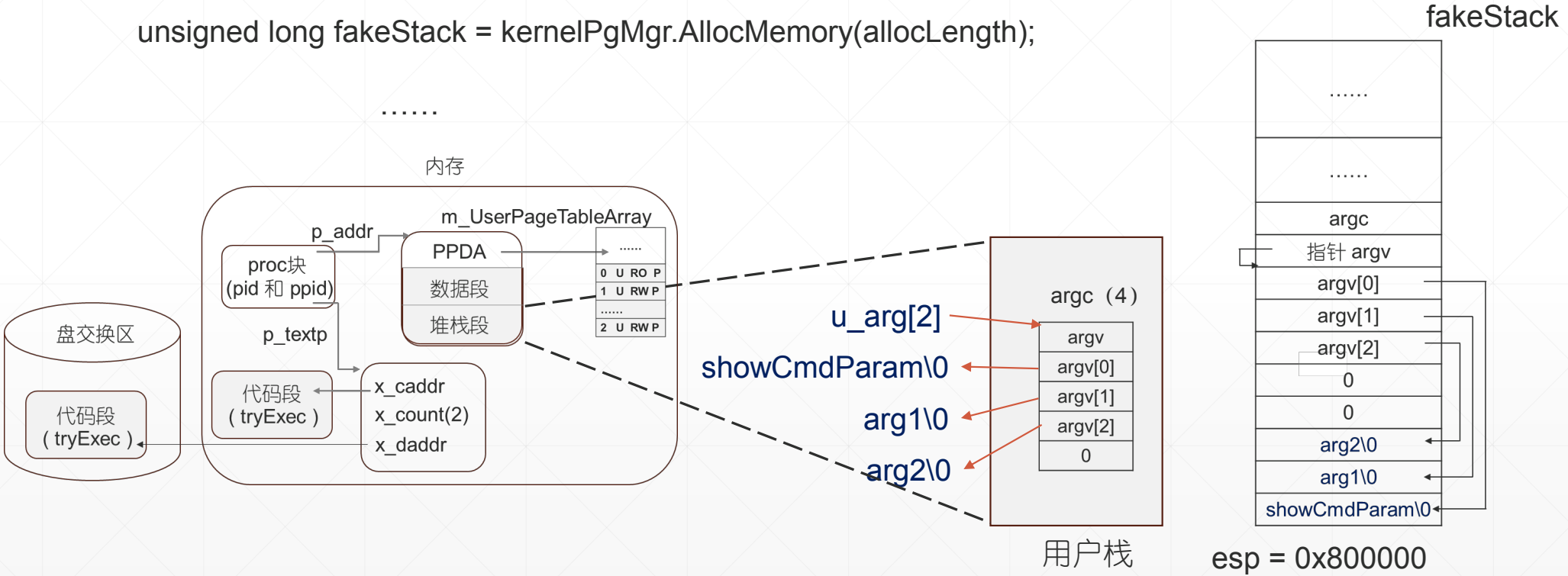
第四步：虚空间够大吗？

```
if ( parser.TextSize + parser.DataSize + parser.StackSize  + PageManager::PAGE_SIZE >
            MemoryDescriptor::USER_SPACE_SIZE - parser.TextAddress)
{
        fileMgr.m_InodeTable->IPut(pInode);
        u.u_error = User::ENOMEM;
        return;
}
```



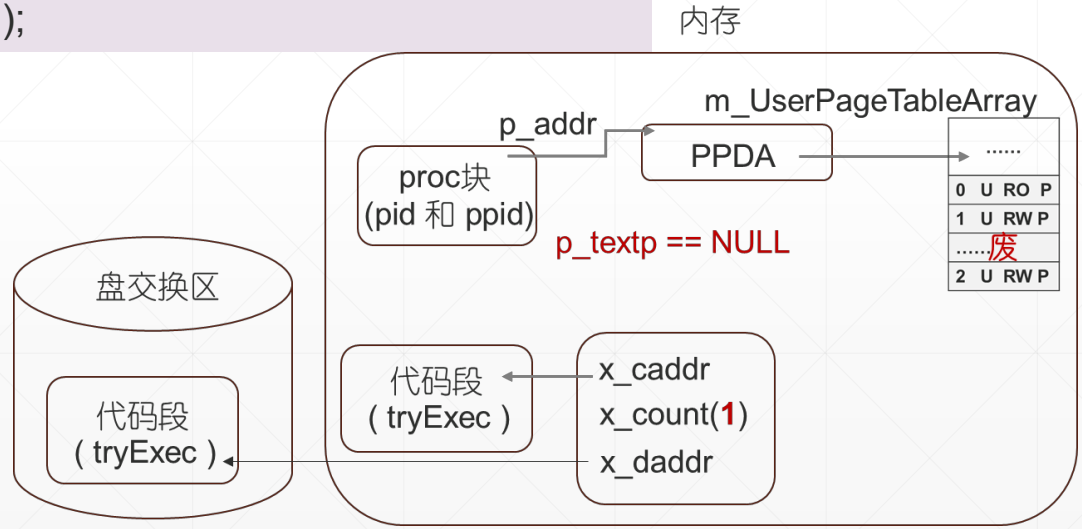
第5步：擦除用户空间之前，复制命令行参数

```
int allocLength = (parser.StackSize + PageManager::PAGE_SIZE * 2 - 1) >> 13 << 13;

unsigned long fakeStack = kernelPgMgr.AllocMemory(allocLength);
```



第6步：擦除用户空间

```
if ( u.u_procp->p_textp != NULL )
{
        u.u_procp->p_textp->XFree();
        u.u_procp->p_textp = NULL;
}
u.u_procp->Expand(ProcessManager::USIZE);
```



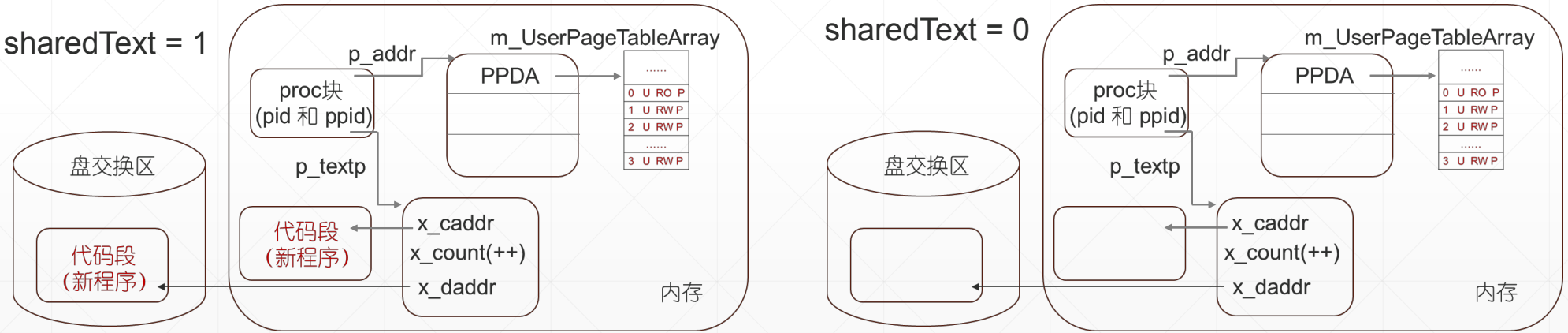
第七步：重建 Text 结构

```
for ( int i = 0; i < ProcessManager::NTEXT; i++ )
{
    if ( NULL == this->text[i].x_iptr )
    {
            if ( NULL == pText )
            {
                    pText = &(this->text[i]);     // 找到的第一个空闲Text结构，可以分配给新程序
            }
    }
    else if ( pInode == this->text[i].x_iptr )      //  Text数组中有新程序的代码段控制块，重用
    {
            this->text[i].x_count++;
            this->text[i].x_ccount++;
            u.u_procp->p_textp = &(this->text[i]);
            pText = NULL;      // 做个区分的标识
            break;
    }
}
```

```
int sharedText = 0;

if ( NULL != pText )
{       // 没有可复用的Text，把pText分配给新程序
        pInode->i_count++;
        pText->x_ccount = 1;
        pText->x_count = 1;
        pText->x_iptr = pInode;
        pText->x_size = u.u_MemoryDescriptor.m_TextSize;
        pText->x_caddr = userPgMgr.AllocMemory(pText->x_size);   // 为代码段分配内存
        pText->x_daddr = Kernel::Instance().GetSwapperManager().AllocSwap(pText->x_size); // 分配盘交换区

        u.u_procp->p_textp = pText;    //   连上 Process 结构
}
else
{       // 有可复用的Text
        pText = u.u_procp->p_textp;
        sharedText = 1;
}
```

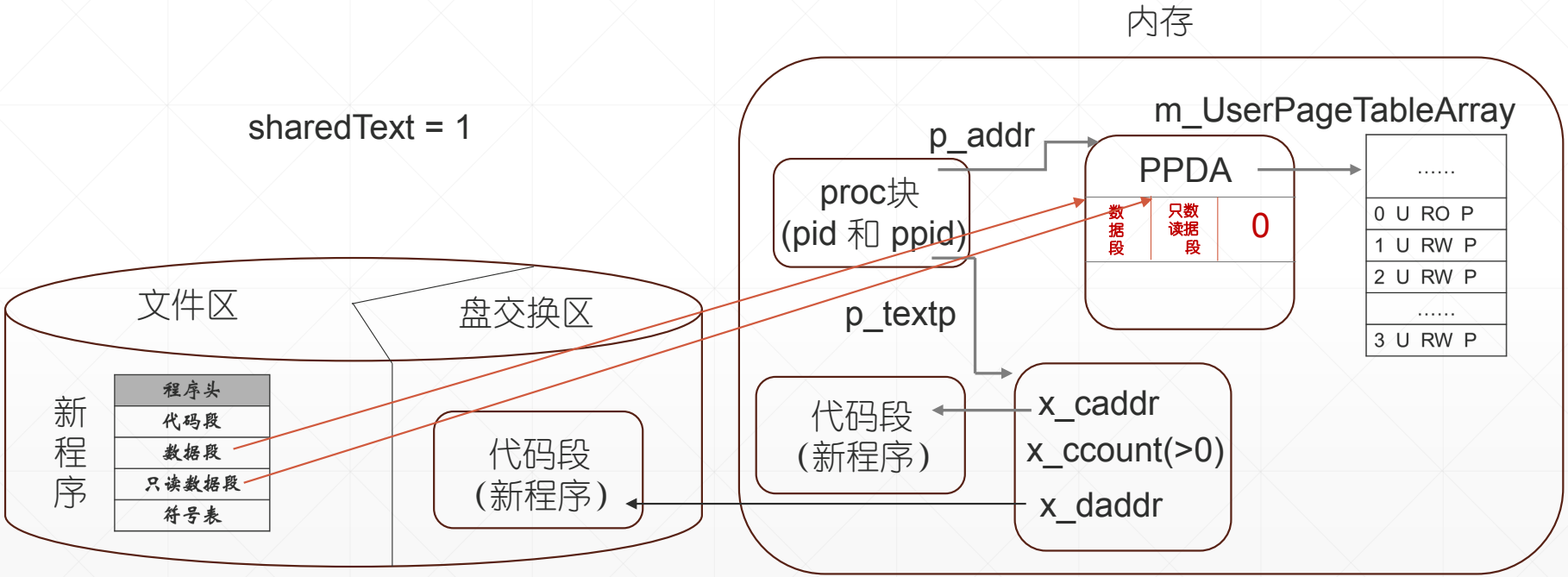
第八步：为可交换部分分配物理内存，刷新相对表，写系统页表

```
// 可交换部分的新尺寸                                          ( 数据段 + 只读数据段 + bss段 )
unsigned int newSize = ProcessManager::USIZE + u.u_MemoryDescriptor.m_DataSize
                                             + u.u_MemoryDescriptor.m_StackSize;

// 为可交换部分分配内存，把原来的PPDA区复制过来
u.u_procp->Expand(newSize);

// 根据新程序的尺寸重写相对虚实地址映照表，并加载到系统页表 */
u.u_MemoryDescriptor.EstablishUserPageTable(  parser.TextAddress, parser.TextSize,
                                              parser.DataAddress, parser.DataSize,
                                              parser.StackSize);
```


sharedText = 1
盘交换区
代码段
(新程序)
proc块
(pid 和 ppid)
p_addr
m_UserPageTableArray
PPDA
0 U RO P
1 U RW P
2 U RW P
3 U RW P
p_textp
代码段
(新程序)
x_caddr
x_count(++)
x_daddr
内存
sharedText = 0
盘交换区
proc块
(pid 和 ppid)
p_addr
m_UserPageTableArray
PPDA
0 U RO P
1 U RW P
2 U RW P
3 U RW P
p_textp
x_caddr
x_count(++)
x_daddr
内存

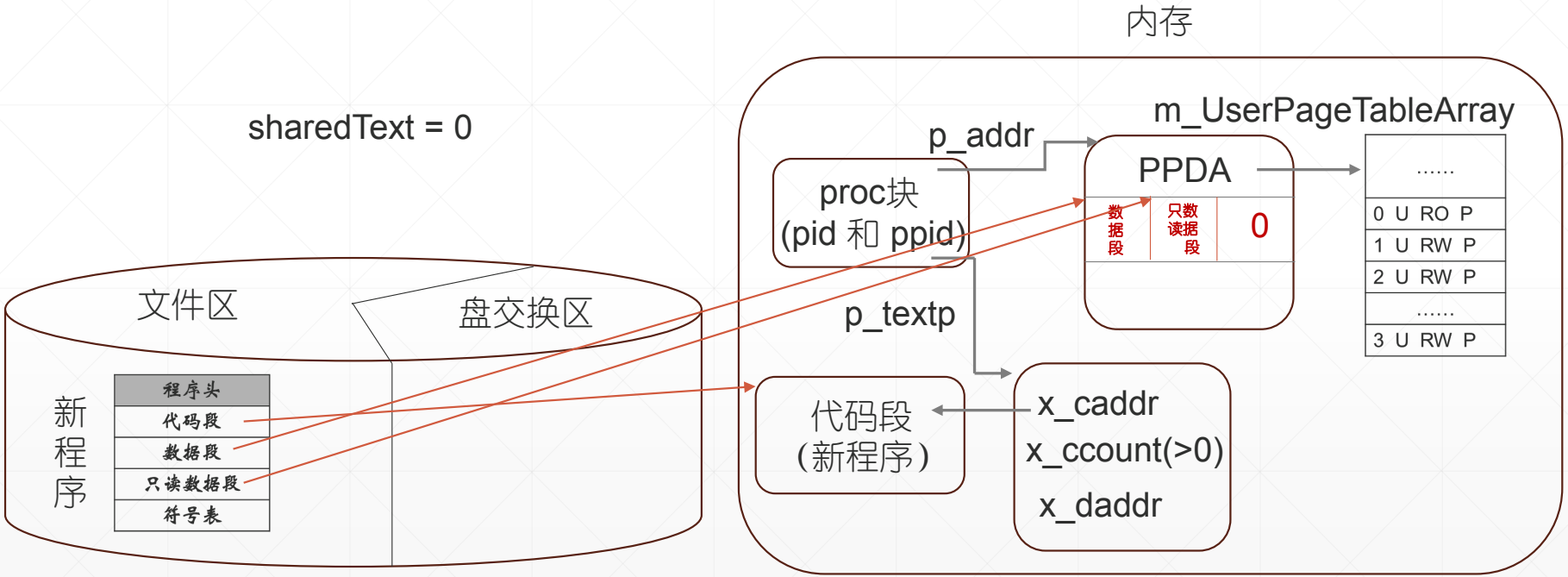
第九步、加载可执行程序代码，常量 和 全局变量的初值

parser.Relocate(pInode, sharedText); // 逐段加载



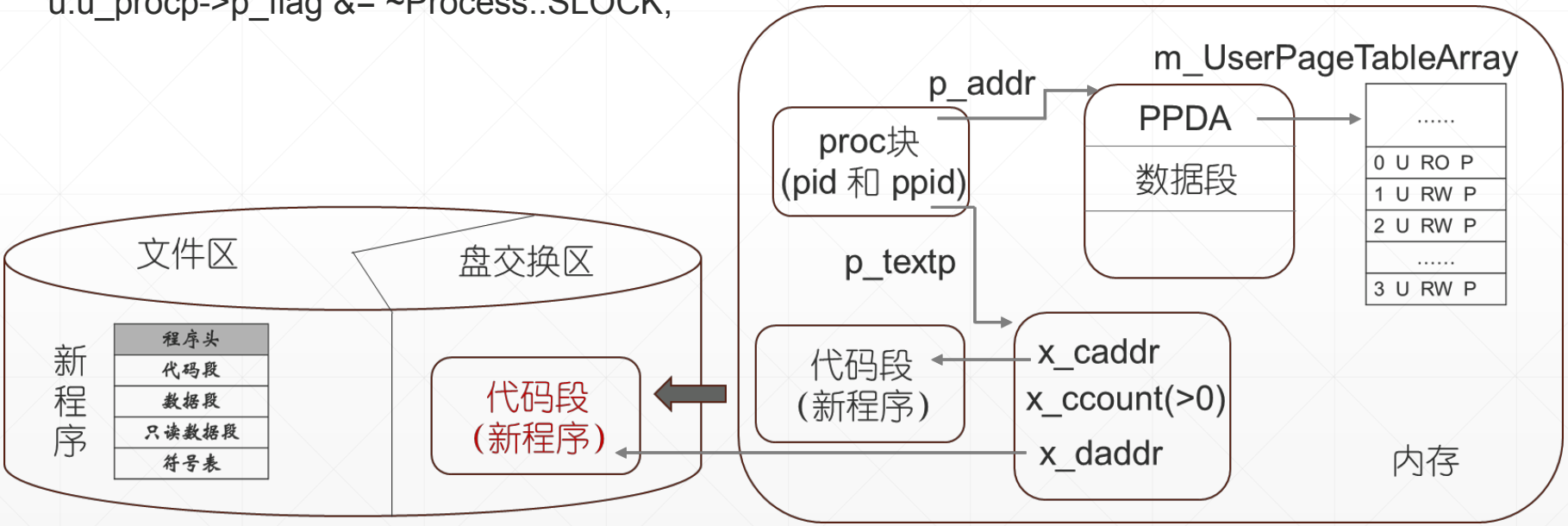
第九步、加载可执行程序代码，常量 和 全局变量的初值

parser.Relocate(pInode, sharedText); // 逐段加载



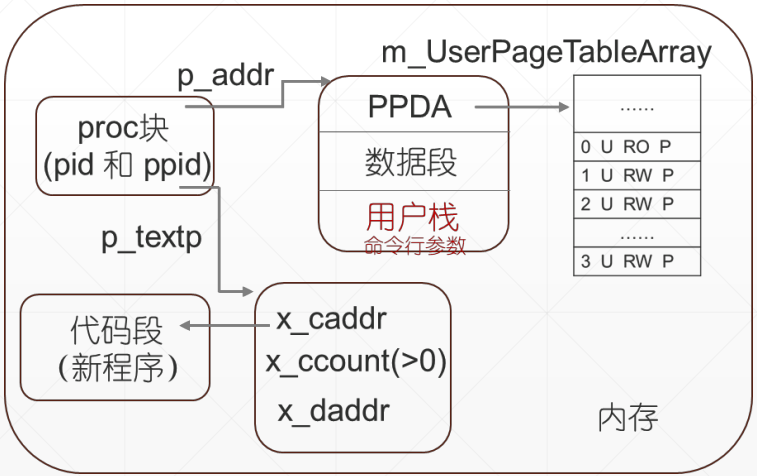
第十步、盘交换区为代码段留一份复本

```
if(sharedText == 0)
{
        u.u_procp->p_flag |= Process::SLOCK;
        bufMgr.Swap(pText->x_daddr, pText->x_caddr, pText->x_size, Buf::B_WRITE);
        u.u_procp->p_flag &= ~Process::SLOCK;
}
```



第十一步：用户栈底构造main栈帧，释放fakeStack

```
Utility::MemCopy(fakeStack + allocLength - parser.StackSize | 0xC0000000,
                 MemoryDescriptor::USER_SPACE_SIZE - parser.StackSize, parser.StackSize);

kernelPgMgr.FreeMemory(allocLength, fakeStack);
```



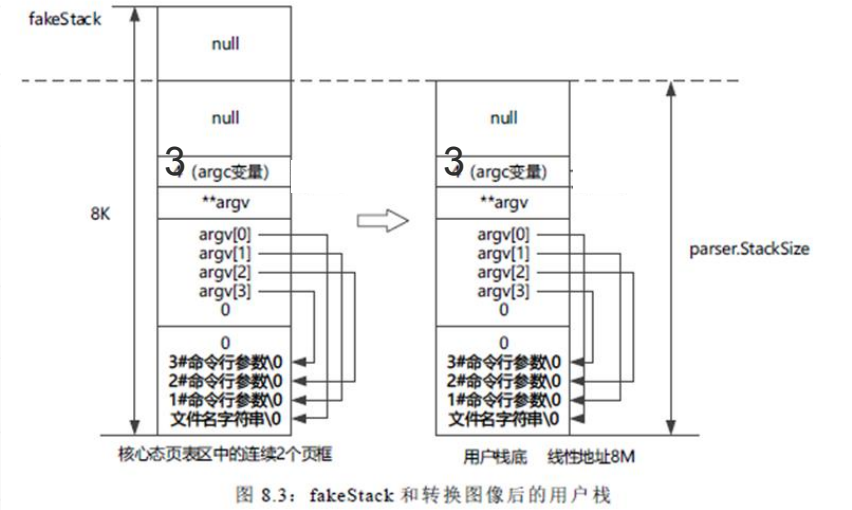
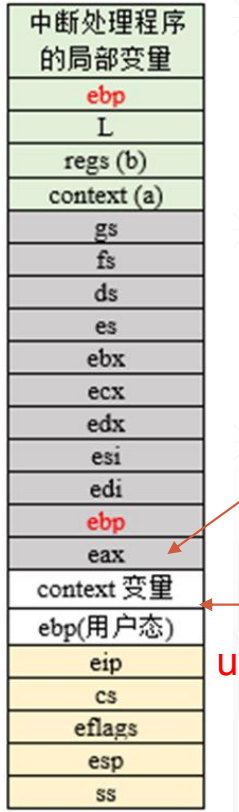


图 8.3：fakeStack 和转换图像后的用户栈

第十二步：初始化应用程序的执行环境




```
// 1、清0所有用户态通用寄存器
for (int i = User::EAX - 2; i < User::EAX - 7 ; i--)
{
        u.u_ar0[i] = 0;
}
u.u_ar0[2] = 0;
```

// 2、将exe程序的入口地址放入核心栈现场保护区中的EAX作为系统调用返回值，这个是runtime要用

```
u.u_ar0[User::EAX] = parser.EntryPointAddress;    // main1( )的入口地址
```

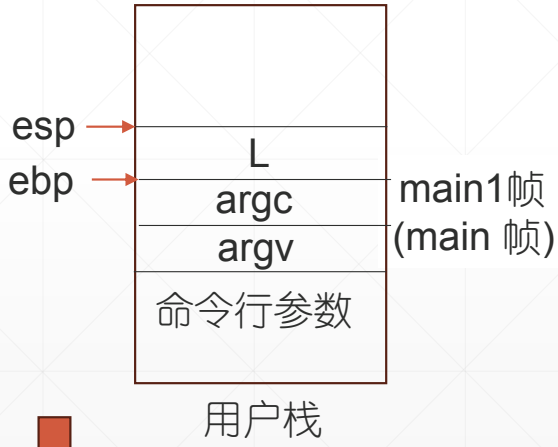
```
// 3、构造exec系统调用的返回环境
struct pt_context* pContext = (struct pt_context *)u.u_arg[4];
pContext->eip = 0x00000000;    // 0x00000000是runtime()的起始地址
pContext->xcs = Machine::USER_CODE_SEGMENT_SELECTOR;
pContext->eflags = 0x200;    // 此项是否篡改无关紧要，因为IRET会开中断
pContext->esp = esp;
pContext->xss = Machine::USER_DATA_SEGMENT_SELECTOR;
```

第十三步：返回用户态，驱动main1函数

eip == 0


```
extern "C" void runtime()
{
    __asm__ __volatile__("  leave;                          \
                            movl %%esp, %%ebp;              \
                            call *%%eax;                    \
                    L:      movl $1, %%eax;                 \
                            movl $0, %%ebx;                 \
                            int $0x80"::);
}
```

```
push %ebp
mov %esp，%ebp
mov %ebp，%esp
pop %ebp
```



开始执行main函数

作业1：改tryExec 使其可以提供任意多的命令行参数

期末要考！

作业2：大实验 作业3：可选

Relocate( )

一、从文件加载逻辑段前，设置起始段号 i，处理代码段的PTE

```
unsigned int PEParser::Relocate(Inode* p_inode, int sharedText)
{
        ……
    PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray();
    unsigned int textBegin = this->TextAddress >> 12 , textLength = this->TextSize >> 12;    // 代码段起始页号
    PageTableEntry* pointer = (PageTableEntry *)pUserPageTable;   // 系统用户页表的指针

    if(sharedText == 1)
        i = 1;      // 代码段可以共享，从1#段，数据段，开始加载
    else
    {       // 否则需要加载代码段
        i = 0;
        for (i0 = textBegin; i0 < textBegin + textLength; i0++)
            pointer[i0].m_ReadWriter = 1;   // 把代码段的PTE设为可写（RW）

        FlushPageDirectory();
    }
```

class PEParser
中定义的段号

```
static const unsigned int TEXT_SECTION_IDX = 0;
static const unsigned int DATA_SECTION_IDX = 1;
static const unsigned int RDATA_SECTION_IDX = 2;
static const unsigned int BSS_SECTION_IDX = 3;
static const unsigned int IDATA_SECTION_IDX = 4;
```

二、清0分配给应用程序的页面

```
for (; i <= this->BSS_SECTION_IDX; i++ )    // 逐段
{
    ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
    int beginVM = sectionHeader->VirtualAddress + ntHeader.OptionalHeader.ImageBase;   // 逻辑段的首地址
    int size = ((sectionHeader->Misc.VirtualSize + PageManager::PAGE_SIZE - 1)>>12)<<12;  // 逻辑段的长度，4K对齐

     int j;
     for (j=0; j<size; j++)  // 清0
    {
            unsigned char* b =(unsigned char*)(j + beginVM);
            *b = 0;
    }
}
```

三、加载应用程序

```
if(sharedText == 1)
        i = 1;
else
        i = 0;

for ( ; i < this->BSS_SECTION_IDX; i++ )   // 依次读入代码、数据 和 只读数据
{
    ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
    srcAddress = sectionHeader->PointerToRawData;    // 文件中的起始偏移量
    desAddress = this->ntHeader.OptionalHeader.ImageBase + sectionHeader->VirtualAddress;  // 内存中的首地址

    u.u_IOParam.m_Base = (unsigned char*)desAddress;
    u.u_IOParam.m_Offset = srcAddress;
    u.u_IOParam.m_Count = sectionHeader->Misc.VirtualSize;
    p_inode->ReadI();

    cnt += sectionHeader->Misc.VirtualSize;
}
```

四、将代码段页面改回只读

```
if(sharedText == 0)
{   //将正文段页面改回只读
        for (i0 = 0; i0 < textLength; i0++)
            pointer[i0].m_ReadWriter = 0;

        FlushPageDirectory();
}

// 释放暂存程序头的2个页框
KernelPageManager& kpm = Kernel::Instance().GetKernelPageManager();
kpm.FreeMemory(PageManager::PAGE_SIZE * 2, (unsigned long)this->sectionHeaders - 0xC0000000 );
return     cnt;
}
```

Part 2、进程终止 与 exit、wait 父子进程同步

- 进程的正常终止 和 异常终止
- exit系统调用（1#系统调用）
- wait系统调用（7#系统调用）

一、进程的正常终止 和 异常终止

- 应用程序执行完毕，进程正常终止
  - 执行exit（n）系统调用
  - main( )返回　exit（0）

思考题：怎样接收main( )的返回值?

做到 return(n) → exit(n)

```
extern "C" void runtime()
{
    __asm__ __volatile__("        leave;                          \
                          movl %%esp, %%ebp;    \
                              call *%%eax;                        \
                          L:  movl $1, %%eax;                     \
                              movl $0, %%ebx;                     \
                              int $0x80"::);
}
```

- 应用程序无法继续执行或没必要继续执行时，内核 或 该APP的用户会向它发信号，杀死执行该程序的进程。
  - 执行非法指令
  - 非法内存访问
  - 运算错误，比如浮点运算结果溢出
  - ctrl+c，kill -9　用户终止运行中的程序

- 无论正常终止，还是异常终止，进程执行内核函数Exit( )终止自己。
  - 每个终止的进程，有一个终止码供父进程 或 系统采集
    - 正常终止的进程，终止码是 n<<8，n是exit系统调用的参数
    - 异常终止的进程，终止码是收到的信号

二、Unix V6++ 的 exit 系统调用

```
int exit(int status)    // 用户空间的钩子函数
{
        int res;
        __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(1),"b"(status));
        if ( res >= 0 )
                return res;
        return -1;
}

int SystemCall::Sys_Rexit()    // exit系统调用的入口
{
        User& u = Kernel::Instance().GetUser();

        // u.u_arg[0] =  u.u_arg[0] << 8;
        u.u_procp->Exit();

        return 0; /* GCC likes it ! */
}
```

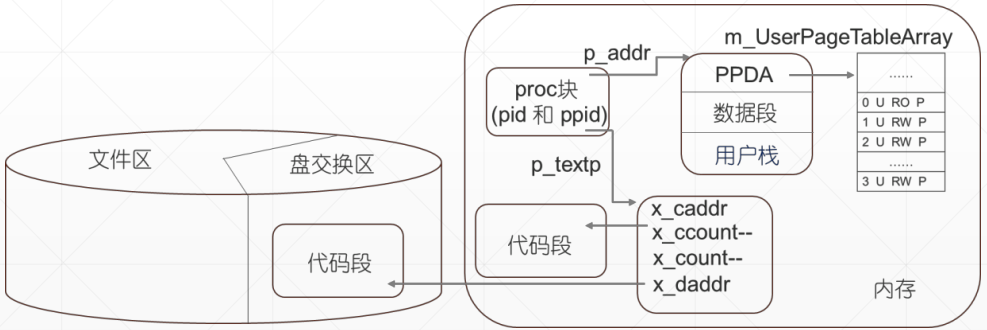
u_arg[0] = 终止码

Exit( )函数

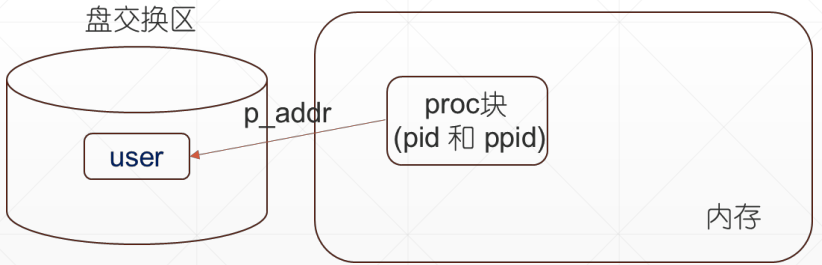
- 进程执行内核函数 Exit( )，完成 运行→终止 的状态变迁。
- 需要执行的主要操作：
  - 在盘交换区上申请一个扇区（512字节），启动 IO，将 user 结构复制到该扇区。
  - 释放资源
    - 关闭所有打开文件
    - 当前工作目录，引用计数--
    - 释放相对虚实地址映射表
    - 释放可交换部分
    - 代码段引用计数减 1，减至0，释放其占用的内存和盘交换区空间
  - 唤醒父进程
  - 将子进程的ppid改为1#进程
  - 唤醒 1#进程

Exit( )执行后的终止进程

p_stat = SZOMB
u_arg[0] = 终止码
u_utime = 用户态执行时长
u_stime = 核心态执行时长
u_cutime = 子进程用户态执行时长
u_cstime = 子进程核心态执行时长



proc块中的 ppid 供父进程找到自己



三、 Unix V6++ 的 wait 系统调用

- 父进程执行 wait 系统调用回收子进程PCB。
- wait 系统调用执行时，如果子进程已终止，立即回收其PCB。否则，父进程入睡等待，直至子进程终止。
- wait系统调用的返回值是子进程的pid号。

wait系统调用的钩子函数

```
int wait(int* status)
{
    int res;
    __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(7),"b"(status));
    if ( res >= 0 )
        return res;
    return -1;
}
```

系统调用的第一个参数是父进程用户空间的一根指针，
这个单元是一个整数，用来存放子进程的终止码。

四、wait、exit系统调用的使用方法，例1

```
#include <stdio.h>
#include <sys.h>
int main1(int argc, char* argv[])
{
   int i, j;
   if(fork())
   {
     i = wait(&j);
     printf("It is parent process. \n");
     printf("The finished child process is %d. \n", i);
     printf("The exit status is %d. \n", j);
   }
   else
   {
     printf("It is child process. \n");
     sleep( 2 );
   }
}
```

已知， Unix V6++系统，只有这一个程序在执行。
父进程pid==2 ， 子进程pid==3。

请写出这个程序的输出。

可以用wait、exit系统调用确保任务之间的前驱后继关系

```
int main1(int argc, char* argv[])
{
   int i, j;
   if(fork())
   {
     i = wait(&j);
     后继任务。。。
   }
   else
   {
     前驱任务。。。
   }
}
```

五、源代码 Exit( )

```
void Process::Exit()
{ ……
    /* 1、信号处理方式设置为1，不再响应任何信号 */
    for ( i = 0; i < User::NSIG; i++ )
    {
        u.u_signal[i] = 1;
    }

    /* 2、关闭所有的打开文件 */
    for ( i = 0; i < OpenFiles::NOFILES; i++ )
    {
        File* pFile = NULL;
        if ( (pFile = u.u_ofiles.GetF(i)) != NULL )
        {
            fileTable.CloseF(pFile);    //释放File结构（引用计数--）
            u.u_ofiles.SetF(i, NULL);   //打开文件表相关元素（fd）置null
        }
    }
```

```
/*   访问不存在的fd会产生error code，清除u.u_error避免影响后续程序执行流程 */
u.u_error = User::NOERROR;

/* 3、当前工作目录的引用计数-- */
inodeTable.IPut(u.u_cdir);

/* 4、释放代码段（引用计数减 1，减至 0 释放占用的内存空间和盘交换区空间） */
if ( u.u_procp->p_textp != NULL )
{
    u.u_procp->p_textp->XFree();
    u.u_procp->p_textp = NULL;
}
/* 5、user结构写入盘交换区 */
/* 1、盘交换区申请一个扇区（512字节），扇区号是blkno*/
int blkno = swapperMgr.AllocSwap(BufferManager::BUFFER_SIZE);
    ……
/* 2.1 内存申请一个缓存块（512字节），用来同步磁盘扇区blkno。磁盘IO要用到。起始地址pBuf->b_addr */
Buf* pBuf = bufMgr.GetBlk(DeviceManager::ROOTDEV, blkno);
/* 2.2  把user结构写进这个缓存块    */
Utility::DWordCopy((int *)&u, (int *)pBuf->b_addr, BufferManager::BUFFER_SIZE / sizeof(int));
/* 3 把这个缓存块同步写入磁盘。进程睡眠等待IO完成
bufMgr.Bwrite(pBuf);
```

```
/* 6、释放相对虚实地址映射表 */
u.u_MemoryDescriptor.Release();

/* 7、释放可交换部分 */
Process* current = u.u_procp;
UserPageManager& userPageMgr = Kernel::Instance().GetUserPageManager();
userPageMgr.FreeMemory(current->p_size, current->p_addr);

/* 8、p_addr指向磁盘上的user结构。置终止状态 */
current->p_addr = blkno;
current->p_stat = Process::SZOMB;
```

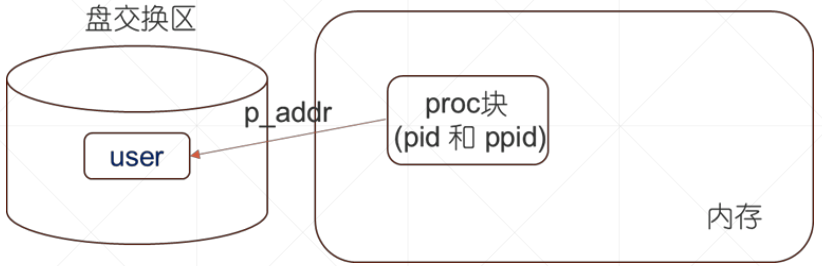
```
/* 9、唤醒父进程 */
for ( i = 0; i < ProcessManager::NPROC; i++ )
{
        if ( procMgr.process[i].p_pid == current->p_ppid )  // 父进程PID
        {
                procMgr.WakeUpAll((unsigned long)&procMgr.process[i]);
                break;
        }
}

/* 10、没找到父进程 */
if ( ProcessManager::NPROC == i )
        current->p_ppid = 1;

/* 11、无论是否找到父进程，唤醒1#进程 */
procMgr.WakeUpAll((unsigned long)&procMgr.process[1]);
```

```
    /* 12、将自己的子进程传给1#进程 */
    for ( i = 0; i < ProcessManager::NPROC; i++ )
    {
            if ( current->p_pid == procMgr.process[i].p_ppid )    // 找到终止进程的子进程
            {
                    Diagnose::Write("……",……);  // 内核使用的格式化输出函数，相当于应用程序用的printf
                    procMgr.process[i].p_ppid = 1;    //  把它们的父进程改成1#进程
                    if ( procMgr.process[i].p_stat == Process::SSTOP )
                    {
                            procMgr.process[i].SetRun();
                    }
            }
    }

    procMgr.Swtch( );    // 13、放弃CPU
}
```

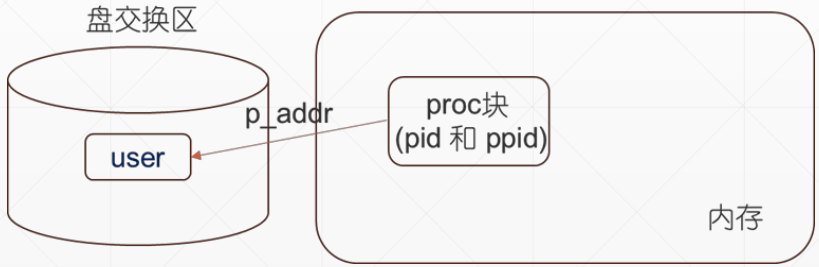
六、源代码　Wait( )

```
void ProcessManager::Wait()
{
   while(true)
   {
      for ( i = 0; i < NPROC; i++ )
      {
         if ( u.u_procp->p_pid == process[i].p_ppid )
         {
            hasChild = true;   /* 找子进程 */
            if( Process::SZOMB == process[i].p_stat )     /* 看它有没有终止 */
            {     处理终止的子进程;        return;      }
         }
      }
      if (true == hasChild)   // 有子进程，但尚未终止
      {
         Diagnose::Write("wait until child process Exit! ");
         u.u_procp->Sleep((unsigned long)u.u_procp, ProcessManager::PWAIT);    // 入睡，等待子进程终止
         continue;     /* 唤醒后，再进for循环，找到终止子进程 */
      }
      else
      {
         u.u_error = User::ECHILD;
         break;      /* Get out of while loop */
      }
   }
}
```

```
if( Process::SZOMB == process[i].p_stat )      // 处理终止的子进程
{
    /* 准备wait()系统调用的返回值：子进程的pid */
    u.u_ar0[User::EAX] = process[i].p_pid;
    /* 释放 Process结构 */
    process[i].p_stat = Process::SNULL;
    process[i].p_pid = 0;
    process[i].p_ppid = -1;
    process[i].p_sig = 0;
    process[i].p_flag = 0;

    /* 启动IO，读入swap分区上的user结构，放在pBuf 管理的内存块 */
    Buf* pBuf = bufMgr.Bread(DeviceManager::ROOTDEV, process[i].p_addr);
    /* 释放盘交换区上user结构占用的空间 */
    swapperMgr.FreeSwap(BufferManager::BUFFER_SIZE, process[i].p_addr);

    User* pUser = (User *)pBuf->b_addr;    // 内存块的首地址，就是读入的子进程user结构的首地址 pUser
```

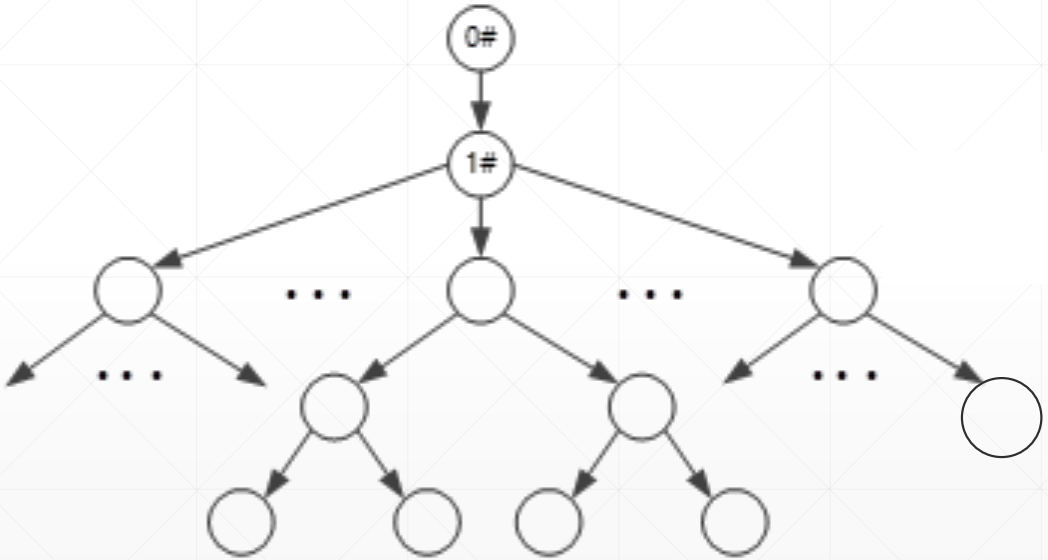


```
                /* 终止子进程的时间累加到父进程上，在字段u_c*time里 */
                u.u_cstime += pUser->u_cstime + pUser->u_stime;
                u.u_cutime += pUser->u_cutime + pUser->u_utime;

               /* pInt指向父进程wait系统调用传入的用户指针，所指变量 j，用来存放子进程的终止码 status */
                int* pInt = (int *)u.u_arg[0];     /* pInt是父进程 wait系统调用的参数，指向其用户空间单元 j，这里
i=wait(&j);                                                放子进程 终止码 */
                *pInt = pUser->u_arg[0];      /* 子进程的终止码 直接写 父进程的 j 变量*/

                /* 释放pBuf 管理的内存块 */
                bufMgr.Brelse(pBuf);
                return;
        }
```

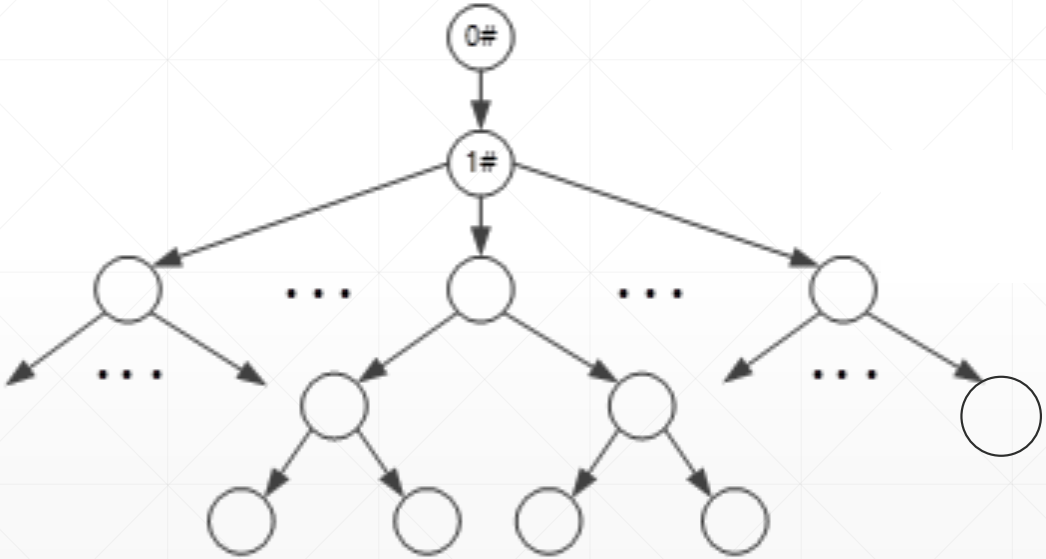


Part 3　Unix 进程树 和 shell

- 在Unix系统中，除0#进程外，所有进程都是fork创建出来的。
- 执行fork的是父进程，被创建的是子进程。
  - 0#进程创建1#进程。 1#进程，是整个Unix系统的监控进程。
  - 1#进程为每个加电的tty创建一个进程，这个进程先初始化终端；然后执行login程序，接受用户输入的用户名和口令字；最终会执行shell程序，变成shell进程，为使用这个终端的用户提供命令行界面服务。
  - shell进程解析命令行，为命令行中出现的每个应用程序创建一个进程。这个进程负责执行这个应用程序。
  - 进程之间的父子关系，绘成进程树。



- 每个进程承担一项任务。任务完成，进程终止。
- 0#进程是内核的服务器进程，永不终止。
- 1#进程是系统监控进程。永不终止。
  - 监控终端运行状态。shell进程终止后，创建新进程等待新用户。
  - 回收孤儿进程的PCB
- shell进程为用户提供命令行界面服务，用户logout，shell进程终止。
- 其余进程，应用程序执行完毕，进程终止。



简化的 shell进程 代码框架

```
main( )
{
  ……
  while( )
  {
        输出 $，睡眠等待用户输入命令行：command arg1 arg2 ……
        如果输入的是 “logout”，shell进程就exit
        while((i=fork( ))==-1);
        if( ! i)
            exec(“command”，arg1，arg2，……);
        else if ( 命令行中没有后台命令符号& ) {
                child = wait(&terminationStatus);
                if（terminationStatus & 0xFF ！= 0）
                      按需，根据terminationStatus & 0xFF 的值
                            printf出来，段错误核心转储之类的信息
              } //shell进程不会睡眠等待负责执行后台作业的进程终止
  }
}
```

shell进程终止后，后台进程会继续运行。它们的父进程是1# 进程。后台进程终止后，1# 进程回收其PCB 和 它的子进程的 PCB。

wait、exit系统调用的使用方法，例2

```
#include <stdio.h>
#include <sys.h>
int main1(int argc, char* argv[])
{
   int i, j;
   if(fork())
   {
       printf("father. \n");
       if(fork())
           i = wait(&j);
       else
       {  printf("second child. \n");   exit(3);   }
   }
   else
   {
       sleep( 2 );
       printf("first child. \n");
   }
}
```

已知， Unix V6++系统，只有这一个程序在执行。父进程pid==2 ，子进程1 pid==3，子进程2 pid==4。

问，（1）这个程序的输出是什么？
（2）子进程1 和 子进程2 的PCB分别是哪个进程回收的？
（3）父进程终止后，PCB是哪个进程回收的？