exec 系统调用源码分析

同济大学计算机系 操作系统课程 2023-11-30

需要执行应用程序时，Unix 系统（其实是 shell 进程）fork 一个新进程，让新进程执行 exec 系统调用装入目标程序的图像，承担执行应用程序的任务。比如，执行命令行：

$ gcc –o testStack testStack.c

Unix 系统会新建一个进程，让它承担装入、运行目标程序 gcc 的任务。

具体而言，3 个步骤（1）shell 进程解析命令行，得到需要执行的应用程序和它的命令行参数（2）shell 进程执行 fork 系统调用创建一个子进程（3）新建的子进程执行 exec 系统调用，清除虚空间中的 shell 进程图像，装入目标程序 gcc，承担执行 gcc 程序的任务。

值得注意的是：exec 系统调用并没有重新创建进程的核心态图象，特别地：p_pid、p_ppid、p_tty、p_uid 和 p_gid 没变。进程还是原来的进程：一样的 p_pid、一样的 p_uid，标准输入输出（p_ttyp）也未曾改变。如果说，进程是一个演员，那么 exec 就是让其饰演一个全新的角色。UNIX 系统中，进程敬业，不会同时执行多个应用程序。

下面我们结合实例介绍 exec 系统调用的使用和实现过程。

1、exec 系统调用的使用

代码 1 是使用 exec 系统调用的应用程序 tryExec，代码 2 是进程转换图象后执行的应用程序 trivialProg。

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

int main1()
{
    int     pid;
    char *const argv[] = {"trivialProg\0", "arg1\0" , "arg2\0", "and arg3\0"};   // trivialProg 的 4 个命令行参数

    pid = fork( );   // 新建一个子进程
    if ( pid == 0 )
    // 子进程抛弃原先的进程图像 tryExec。装入新的应用程序，从 main 函数的入口开始执行 trivialProg
        execv ("./trivialProg\0", argv) < 0);
    else
        wait( );    // 父进程入睡，等待子进程终止

    printf("The end of tryExec.\n");
}
```

代码 1、应用程序 tryExec.c

父进程新建子进程，执行同一目录下的另一个应用程序 trivialProg

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

int main1(int argc, char *argv[])
{
    int i;
    for(i = 0; i < argc; i++)
        printf("argument %d:\t%s\n", i, argv[i]);
    exit(0);
}
```

代码 2、应用程序 trivialProg.c，输出命令行参数

argc 是命令行参数的数量，argv 是存放命令行参数的数组，argv[0]是应用程序的文件名。

执行程序 tryExec，输出如下：

```
$ ./ tryExec
argv[0]: trivialProg
argv[1]: arg1
argv[2]: arg2
argv[3]: and arg3
The end of tryExec.
```

2、库函数 execv

UNIX V6++中 exec 是 11#系统调用，负责执行这个系统调用的库函数是：

int execv(char *pathname, char *argv[])

其中，pathname 是字符串指针，指向可执行文件路径名，本例：“./trivialProg\0”。argv 是向 trivialProg 提供的命令行参数，这是一个字符串指针数组，每个元素指向一个命令行参数，后者是一个以’\0’结束的字符串。依惯例 argv[0]指向的字符串为可执行文件名，数组中的最后一个元素是(char *)0。图 1(a)是 execv( )函数的栈帧结构, (b)是本例 tryExec 程序的 execv( )函数栈帧。

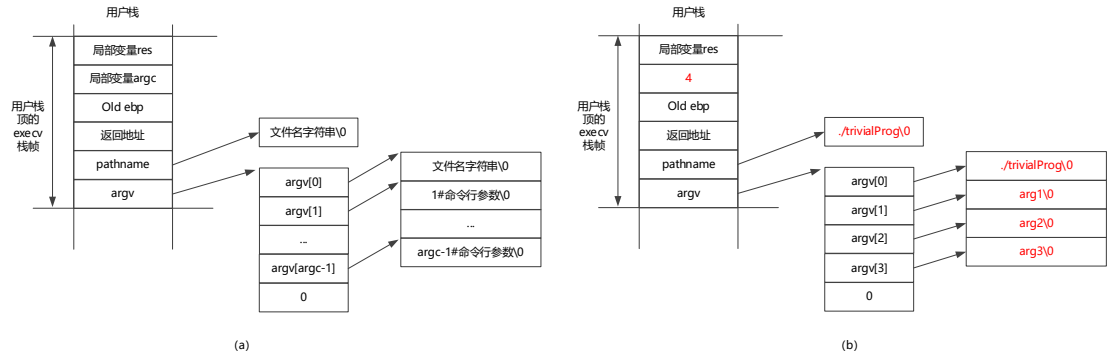


图 1、execv 栈帧

库函数 execv 是 exec 系统调用的钩子函数，定义在 src/lib/sys.c。第 1 个参数是可执行文件名，第 2 个参数是为它提供的命令行参数。

```
int execv(char *pathname, char *argv[])
{
    int res;

    /* 清点，得出命令行参数的个数 */
    int argc = 0;
    while(argv[argc] != 0)
        argc++;

    /* 执行 exec 系统调用。3 个入口参数分别是指向可执行文件名的指针，命令行参数的数量 和 指向命令行参数数组的指针。两根指针指向用户空间，系统调用执行期间，用来提取用户输入信息。*/
    __asm__ volatile ( "int $0x80":"=a"(res):"a"(11),"b"(pathname),"c"(argc),"d"(argv));

    if ( res >= 0 )
        return res;
    return -1;
}
```

3、exec( )

粗略而言，exec( )程序的执行过程大致有如下步骤，继续用我们上面提到的示例。

(1) 在文件系统中搜索可执行文件 trivialProg。

(2) 确定进程拥有执行 trivialProg 程序的权限。

(3) 解析（parse）可执行文件的程序头，获得代码段、数据段、BSS 段~~、初始堆~~和初始栈各自的长度。确保用户空间可以容纳所有逻辑段。

(4) 复制命令行参数至核心空间。

(5) 释放原进程图像的正文段，数据段和堆栈段。保留 PPDA 区。

(6) 重构正文段，引用 trivialProg 的 text 结构。如果没有其它进程执行该程序，为 trivialProg 的正文段分配 text 结构，内存空间和盘交换区空间。

(7) Expand( )扩大进程图像的可交换部分，按 PPDA 区、数据段、初始栈的长度总和为 trivialProg 申请内存空间。如果失败，PPDA 会被暂存在盘交换区，待 0#进程发现内存足以容纳新的可交换部分，将其换入。

(8) EstablishUserPageTable( )重建相对虚实地址映射表，并且将所有逻辑页面映射至分配给它的物理页框。

(9) 从磁盘读入可执行文件的代码段。

(10) 可交换部分，除 PPDA 之外的所有空间清 0。之后读可执行文件的数据段，为带初值的全局变量赋初值。bss 变量初值是 0。

(11) 在可交换部分的尾部构造 main 栈帧。

(12) 修改核心栈底 pt_regs 中保存的各寄存器值，它们是 trivialProg 开始执行时通用寄存器的初始值。最重要地：

regs->esp = 用户栈顶，regs->xss = 用户栈段描述子，

regs->eip 和 xcs 分别为应用程序的入口地址和用户代码段描述子，

regs->eax = main()函数的入口地址（虚地址），

regs 中其余寄存器清 0。应用程序将拥有一个“干净”的初始运行环境。

(13) u_signal 数组清 0，将所有的信号处理方式设置为默认，确保信号可以杀死运行中的

trivialProg。
(14) exec( )系统调用返回用户态。进程从 main( )函数的第一条指令开始执行 trivialProg 程序。main( )函数返回，进程终止。

```
void ProcessManager::Exec()
{
	Inode* pInode;
	Text* pText;
	User& u = Kernel::Instance().GetUser();
	FileManager& fileMgr = Kernel::Instance().GetFileManager();
	UserPageManager& userPgMgr = Kernel::Instance().GetUserPageManager();
	KernelPageManager& kernelPgMgr = Kernel::Instance().GetKernelPageManager();
	BufferManager& bufMgr = Kernel::Instance().GetBufferManager();

	/* (1) 在文件系统中搜索可执行文件 trivialProg */
	pInode = fileMgr.NameI(FileManager::NextChar, FileManager::OPEN);
	if ( NULL == pInode )  // 可执行文件 trivialProg 不存在
	{
		u.u_error = User :: ENOENT;
		return;   // exec 系统调用出错返回
	}

	/* exec 系统调用消耗系统资源过多，同时执行的进程数目不可超出限制 */
	while( this->ExeCnt >= NEXEC )
	{
		u.u_procp->Sleep((unsigned long)&ExeCnt, ProcessManager::EXPRI);
	}
	this->ExeCnt++;

	/* (2) 进程必需拥有 trivialProg 的执行权限，且 trivialProg 必需是普通文件（不可以是目录） */
	if ( fileMgr.Access(pInode, Inode::IEXEC) || (pInode->i_mode & Inode::IFMT) != 0 )
	{	// 没有执行权限。出错返回前，
		if ( this->ExeCnt >= NEXEC )
		{
			WakeUpAll((unsigned long)&ExeCnt);   // 唤醒等待执行 exec 系统调用的其它进程
		}
		this->ExeCnt--;
		fileMgr.m_InodeTable->IPut(pInode);   // 释放 trivialProg 的内存 Inode
		u.u_error = User :: EACCES;
		return;   // exec 系统调用出错返回
	}

	PEParser parser;
```

```
    /* (3) 解析（parse）可执行文件的程序头 */
    if ( parser.HeaderLoad(pInode)==false )   // 稍后介绍 PE 文件和 PE 程序头的格式
    {    // 可执行文件格式不对，不是期待的 PE 程序头
         fileMgr.m_InodeTable->IPut(pInode);
         u.u_error = User :: ENOEXEC
         return;   // exec 系统调用出错返回
    }

    /* 获得正文段的起始地址、长度 */
    u.u_MemoryDescriptor.m_TextStartAddress = parser.TextAddress;
    u.u_MemoryDescriptor.m_TextSize = parser.TextSize;

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

    /* 获得堆栈段初始长度 */
    u.u_MemoryDescriptor.m_StackSize = parser.StackSize;

    /* 确保用户空间可以容纳所有逻辑段 */
    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;
    }

    /* (4) 复制命令行参数至核心空间 */
     * 分配内存用于存放 trivialProg 的命令行参数 argc，argv[]。这些参数存放在进程的用户空间。
     * 需要把它们备份到 fakeStack，构造出 trivialProg 程序的 main 栈帧。
     */
    /* 页表区分配连续物理页框，用来构造 trivialProg 的 main 栈帧，存放命令行参数。fakeStack 是这
块空间的起始地址（物理地址）。分配粒度，8K 字节。*/
    int allocLength = (parser.StackSize + PageManager::PAGE_SIZE * 2 - 1) >> 13 << 13;
    unsigned long fakeStack = kernelPgMgr.AllocMemory(allocLength);

    int argc = u.u_arg[1];    // 命令行参数的个数
    char** argv = (char **)u.u_arg[2];    // 指向用户空间中存放的命令行参数

    unsigned int esp = MemoryDescriptor::USER_SPACE_SIZE;   /* esp，8M。是用户栈底的地
址 */
    unsigned long desAddress = fakeStack + allocLength + 0xC0000000;   /* fakeStack 空间
底部的线性地址 */
```

```
/* 复制 argv[]指针数组指向的命令行参数字符串 （压栈命令行参数）*/
for (int i = 0; i < argc; i++ )
{
	length = 0;
	while( NULL != argv[i][length] )  // 计算命令行参数 i 的长度，'\0' 字符不记录在内
	{
		length++;
	}
	// 将命令行参数 i，复制到 fakeStack。包括尾部的 \0 字符
	desAddress = desAddress - (length + 1);
	Utility::MemCopy((unsigned long)argv[i], desAddress, length + 1);

	esp = esp - (length + 1);   // esp 是命令行参数 i 的首地址
	argv[i] = (char *)esp;   // 暂存入用户空间：argv[i]单元
}
```

// 从用户空间，把 argv 数组复制过来，存入 fakeStack 正确的位置。

```
/* 后续存放的是 int 型数值，这里以 16 字节边界对齐 */
desAddress = desAddress & 0xFFFFFFF0;
esp = esp & 0xFFFFFFF0;

/* 压栈整数 0，作为 argv 数组的结束符 */
int endValue = 0;
desAddress -= sizeof(endValue);
esp -= sizeof(endValue);
Utility::MemCopy((unsigned long)&endValue, desAddress, sizeof(endValue));

/* 压栈 argv 数组，形成命令行参数指针数组 */
desAddress -= argc * sizeof(int);
esp -= argc * sizeof(int);
Utility::MemCopy((unsigned long)argv, desAddress, argc * sizeof(int));

/* 压栈命令行参数指针数组的起始地址，这是 main 函数的第 2 个参数 char ** argv */
endValue = esp;
desAddress -= sizeof(int);
esp -= sizeof(int);
Utility::MemCopy((unsigned long)&endValue, desAddress, sizeof(int));

/* 压栈命令行参数的数量，这是 main 函数的第 1 个参数 int argc */
desAddress -= sizeof(int);
esp -= sizeof(int);
Utility::MemCopy((unsigned long)&argc, desAddress, sizeof(int));   /* 完结 */
```

```
    /* (5) 释放原进程图像的正文段，数据段和堆栈段。保留 PPDA 区。*/
    if ( u.u_procp->p_textp != NULL )   // 释放正文段
    {
        u.u_procp->p_textp->XFree();
        u.u_procp->p_textp = NULL;
    }
    u.u_procp->Expand(ProcessManager::USIZE);   // 释放数据段和堆栈段。保留 PPDA 区

    /* (6) 重构正文段，引用 trivialProg 的 text 结构。*/
    pText = NULL;
    for ( int i = 0; i < ProcessManager::NTEXT; i++ )   // 可以与其它进程共享 trivialProg 的正文段 ？
    {
        if ( NULL == this->text[i].x_iptr )   // 顺带用 pText 记下找到的第一个空闲 text 结构
        {
            if ( NULL == pText )
            {
                pText = &(this->text[i]);
            }
        }
        else if ( pInode == this->text[i].x_iptr )   // 找到 trivialProg 的正文段，引用它
        {
            this->text[i].x_count++;
            this->text[i].x_ccount++;
            u.u_procp->p_textp = &(this->text[i]);   // Process 结构引用已有的 text 结构
            pText = NULL;
            break;
        }
    }

    int sharedText = 0;

    if ( NULL != pText )   // trivialProg 的正文段没找到。新建一个 text 结构
    {
        pInode->i_count++;
        pText->x_ccount = 1;
        pText->x_count = 1;
        pText->x_iptr = pInode;
        pText->x_size = u.u_MemoryDescriptor.m_TextSize;
        ，而具体正文段内容的读入需要等到建立页表映射之后，再从 mapAddress 地址起始的 exe 文件中读入 */
        pText->x_caddr = userPgMgr.AllocMemory(pText->x_size);   // 为正文段分配内存
        pText->x_daddr =          // 为正文段分配盘交换区
                Kernel::Instance().GetSwapperManager().AllocSwap(pText->x_size);
```

```
            u.u_procp->p_textp = pText;       // Process 结构引用新的 text 结构
        }
        else
        {
            pText = u.u_procp->p_textp;
            sharedText = 1;     // 标识后续无需从磁盘读入 trivialProg 的正文段
        }

        /* (7) 为可交换部分分配内存 */
        unsigned int newSize = ProcessManager::USIZE + u.u_MemoryDescriptor.m_DataSize +
u.u_MemoryDescriptor.m_StackSize;     // 计算可交换部分的新长度
        u.u_procp->Expand(newSize);       // 为可交换部分分配内存。低地址端保留原先的 PPDA 区

        /* (8) 重建相对虚实地址映射表，并且将所有逻辑页面映射至分配给它的物理页框 */
        u.u_MemoryDescriptor.EstablishUserPageTable(parser.TextAddress, parser.TextSize,
parser.DataAddress, parser.DataSize, parser.StackSize);

        /* (9) (10) 从可执行文件 trivialProg 中依次读入 .text 段、.data 段 和 rdata 段 */
        parser.Relocate(pInode, sharedText);

        /* .text 段在 swap 分区上留复本 */
        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;
        }

        /* (11) 在可交换部分的尾部构造 main 栈帧。
        /* 具体操作：将 fakeStack 中构造的 main 栈帧复制到新的用户栈 */
        Utility::MemCopy(fakeStack + allocLength – (MemoryDescriptor::USER_SPACE_SIZE –
esp) | 0xC0000000,  esp,  MemoryDescriptor::USER_SPACE_SIZE - esp);
        /* 释放 fakeStack 占用的内存 */
        kernelPgMgr.FreeMemory(allocLength, fakeStack);

        /* 释放 Inode，减少 ExeCnt 计数值，唤醒等待执行 exec 系统调用的其它进程 */
        fileMgr.m_InodeTable->IPut(pInode);
        if ( this->ExeCnt >= NEXEC )
        {
            WakeUpAll((unsigned long)&ExeCnt);
        }
        this->ExeCnt--;

        /* (12) 用默认的方式处理信号  */
```

```
    for (int i = 0; i < u.NSIG ; i++)
    {
        u.u_signal[i] = 0;
    }

    /* (13) 设置用户态寄存器组，让应用程序有一个干净的执行环境 */
    // 清 0 所有的用户态通用寄存器，不包括 EAX
    for (int i = User::EAX - 4; i < User::EAX - 4*7 ; i = i - 4)
    {
        u.u_ar0[i] = 0;
    }

    u.u_ar0[User::EAX] = parser.EntryPointAddress;    // EAX 是程序的入口地址

    /* 构造出 Exec()系统调用的退出环境，使进程返回用户态时，从头开始执行应用程序 */
    struct pt_context* pContext = (struct pt_context *)u.u_arg[4];
    pContext->eip = 0x00000000;   // 返回用户态后执行线性地址为 0 的函数 runtime()
    pContext->xcs = Machine::USER_CODE_SEGMENT_SELECTOR;
    pContext->eflags = 0x200;  // IF 为 1
    pContext->esp = esp;       // 用户栈顶，当前指向 main 栈帧的实参区
    pContext->xss = Machine::USER_DATA_SEGMENT_SELECTOR;
}
```

4、exec 系统调用返回用户态后，执行 runtime( )函数驱动应用程序。进程从 main( ) 的入口地址开始执行

```
extern "C" void runtime()
{
    /*
    1. 销毁 runtime 的 stack Frame
    2. esp 中指向用户栈中 argc 位置，而 ebp 尚未正确初始化
    3. eax 中存放可执行程序 EntryPoint
    4~6. exit(0)结束进程
    */
    __asm__ __volatile__("  leave;     \        1
                            movl %%esp, %%ebp;     \       2
                            call *%%eax;           \       3  调用 main 函数
                            movl $1, %%eax;    \       4 main 函数返回后执行 1#系统调用 exit
                            movl $0, %%ebx;    \          exit 系统调用的入口参数是 0
                            int $0x80"::);
}
```

5、可执行文件的格式 和 程序头

程序是磁盘上的可执行文件。通常，可执行文件的开头是一个叫做程序头的数据结构，记录着程序执行需要多少存储空间，此外还包括程序入口地址等等与运行相关的重要信息。之后是程序代码和变量初值。

目前，UNIX V6++操作系统可执行的程序是 PE（Portable Executable）格式的。一个 PE 可执行文件一般由 5 部分组成，它们是：

（1）PE 文件 DOS 头（IMAGE_DOS_HEADER）；

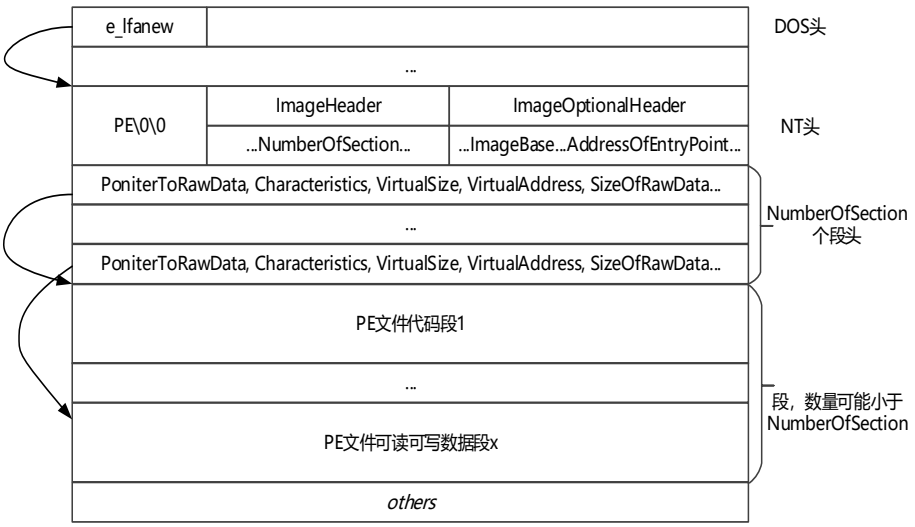
（2）PE 文件 NT 头（IMAGE_NT_HEADERS）；

（3）PE 文件段表（SECTION TABLEs）；

（4）程序代码段和数据段，可能有多个；

（5）符号表和其它与调试相关的信息。

PE 文件格式如下图所示。Unix V6++不支持动态链接，应用程序静态链接而成。本节仅介绍与静态链接程序装入有关的文件头部信息。



PE 文件格式

5.1 DOS 程序头

DOS 头是 PE 文件的第一部分，其中的 e_lfanew 是一个整数，其值通常为 0x3Ch，表示 NT 头在 PE 文件中的偏移量。解析 PE 文件时，DOS 头不用，仅使用 e_lfanew 寻找 NT 头。与该部分对应的是源代码 PEParser.h 文件中的 struct ImageDosHeader。

5.2 NT 程序头

NT 程序头是 PE 文件的第二部分，包含 3 个主要成员，刻画可执行文件结构和进程虚地址空间布局：

```
struct ImageNTHeader
{
    unsigned long Signature;
    ImageFileHeader FileHeader;
    ImageOptionalHeader32 OptionalHeader;
};
```

NT 程序头（ImageNTHeader 结构）

（1）Signature：可执行文件魔数（Magic Number），标识可执行文件的格式。PE 文件的 Signature 是字符串“PE\0\0”的 ASCII 码，换算成整数值是 0x00004550。

（2）**FileHeader**：ImageFileHeader 结构，定义于 src/include/PEParser.h。其中最重要的字段是 **NumberOfSections**，表示**程序包含的段数**。

（3）OptionalHeader：ImageOptionalHeader32 结构，定义于 src/include/ PEParser.h。，描述进程用户空间布局，重要数据成员包括：

① ImageBase：映像文件装入内存时，在线性地址空间中的起始地址
② BaseOfCode：代码起始地址（线性）
③ BaseOfData：数据起始地址（线性）
④ AddressOfEntryPoint：程序入口的线性地址，也就是 main( )函数的入口地址
⑤ SizeOfCode：代码段长度
⑥ SizeOfInitializedData：数据段长度
⑦ SizeOfUninitializedData：bss 段长度

Unix V6++系统中，每个 PE 程序包含唯一代码段。这是因为链接多个程序模块时，gcc

编译器会将所有程序模块中标记为 text 的代码段相邻放置，形成单个代码段。相类似，数据段、bss 段也是唯一的，分别包括程序运行所需的全部带初值的全局变量和不带初值的全局变量。

5.3 段表

NT 程序头之后是段表，包含 NumberOfSections 个段头（SectionHeader）。每个段头登记一个逻辑段的属性，是名为 ImageSectionHeader 的数据结构（代码 8.2）。

NumberOf Sections 是程序包含的逻辑段数，定义在 NT 程序头 FileHeader 字段中。

```
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;
};
```

段头（ImageSectionHeader）中有 4 个变量对程序加载起重要作用：

① VirtualSize：线性地址空间中，本段的长度

② VirtualAddress：加载进内存后，本段的起始线性地址= ImageBase+ VirtualAddress

③ SizeOfRawData：可执行文件中，本段的长度

④ PointerToRawData：本段在 PE 文件中的偏移量

注意：VirtualSize 可能不等于 SizeOfRawData，比方说 BSS 段。原因在于 BSS 段中所有变量初值为 0、无需存储，因此 BSS 段的 SizeOfRawData=0，VirtualSize 是该段的内存需要量。此外，Characteristics 是段属性，其值为枚举量：0x00000002h 表示本段为代码段，0x00000004h 带初值的数据段，0x00000008h BSS 段；还有其它值，UNIX V6++不支持。

5.4 逻辑段

上述所有结构依次放置，形成 PE 程序头。程序头之后是程序所有代码和全局变量初值，不含 DLL 模块。后者包含的代码和数据由操作系统维护，运行时动态链接至需要的进程。UNIX V6++是个实验性的操作系统，暂不支持动态链接。

静态链接生成的可执行程序中有 4 个逻辑段与程序装入操作相关，按出现在进程中的次序，它们依次为：

0#段，TEXT_SECTION，代码段。程序运行需要调用的所有子程序；

1#段，DATA_SECTION，数据段。带初值的全局变量，存放在可执行文件中的数据段，

其内容是这些全局变量的初值；

2#段，RDATA_SECTION，只读数据段。常量，比如 printf 函数中出现的格式串。

3#段，BSS_SECTION，BSS 段。未赋初值的全局变量（初值为 0）。可执行文件中没有与 bss 段对应的空间。

5.5 读取，解析 PE 程序头 HeaderLoad( )

```
bool PEParser::HeaderLoad(Inode* p_inode)
{
    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();

    /* 读取 nt_Header */
    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();

    /* Unix V6++只能运行 PE 格式的可执行程序 */
    if ( ntHeader.Signature!=0x00004550 )
    {    // 签名不对，exec 系统调用失败
        return false;
    }

    /* 从可执行程序中读入段表，暂存进页表区。这里，为其临时分配 2 个连续的物理页框，是为了与相对虚实地址映射表对齐 */
    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();

   /* 计算、设置各个逻辑段在虚空间中的属性。
    * 静态链接的逻辑段在可执行文件中的出现顺序，依次为 .text->.data->.rdata->.bss。下面这段代
    * 码用这个硬编码的顺序，计算代码段，数据段的起始地址和长度，有点偷懒。*/
    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;

    this->StackSize = ntHeader.OptionalHeader.SizeOfStackCommit;
    this->HeapSize = ntHeader.OptionalHeader.SizeOfHeapCommit;

    this->EntryPointAddress = ntHeader.OptionalHeader.AddressOfEntryPoint +
                        ntHeader.OptionalHeader.ImageBase;

    return true;
}
```

5.6　从可执行文件中读出代码段和数据段 Relocate( )

```
unsigned int PEParser::Relocate(Inode* p_inode, int sharedText)
    // sharedText 为 1，表示可以与其它进程共享代码段。
{
    User& u = Kernel::Instance().GetUser();
    unsigned long srcAddress, desAddress;
    unsigned cnt = 0;
    unsigned int i = 0;
    unsigned int i0 = 0;

    /* 为读入正文段做准备。textBegin 是正文段在页表中的起始地址，textLength 是正文段的长度，以
页为单位。pointer 指针指向用户页表 */
    PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray();
    unsigned int textBegin = this->TextAddress >> 12 , textLength = this->TextSize >> 12;
    PageTableEntry* pointer = (PageTableEntry *)pUserPageTable;

    if(sharedText == 1)
        i = 1;          // 读入可执行文件，从数据段开始
    else
    {
        i = 0;          // 读入可执行文件，从代码段开始
        // 临时将正文段页面设为可写，为内核写代码段做准备
        for (i0 = textBegin; i0 < textBegin + textLength; i0++)
            pointer[i0].m_ReadWriter = 1;
        FlushPageDirectory();
```

```
}
/* 逐段清 0 所有用户页面。副作用：bss 变量赋初值 */
for (; i <= this->BSS_SECTION_IDX; i++ )
{
    ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
    int beginVM = sectionHeader->VirtualAddress +
                                        ntHeader.OptionalHeader.ImageBase;
    int size = ((sectionHeader->Misc.VirtualSize + PageManager::PAGE_SIZE –
                    1)>>12)<<12;

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

/* 读正文段（optional)；读文件，得全局变量的初值  */
for ( ; i < this->BSS_SECTION_IDX; i++ )
{
    ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
    srcAddress = sectionHeader->PointerToRawData;
    desAddress =
        this->ntHeader.OptionalHeader.ImageBase + sectionHeader->VirtualAddress;

    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;
}

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

/* 释放用来暂存段表的页面 */
KernelPageManager& kpm = Kernel::Instance().GetKernelPageManager();
```

```
    kpm.FreeMemory(PageManager::PAGE_SIZE * 2, (unsigned long)this->sectionHeaders
- 0xC0000000 );
    return    cnt;
}
```

6、简易的 shell 程序框架

Shell 程序是 Unix 系统为用户提供的命令行界面。成功登录系统后，shell 进程为用户提供命令行解析，应用程序加载、作业控制等一系列字符界面服务。下面给出简化的 shell 程序代码框架，支持程序后台运行，但每个命令行只能启动一个应用程序。它很像 tryExec.c。

```
main( )
{
   ……
   while( )
   {
     输出命令行提示符 $，睡眠等待用户输入命令行：command arg1 arg2 ……
     解析命令行，得到用户希望执行的应用程序，命令行参数，识别后台作业标识符'&'
     if( command==“logout” )
         exit(0) ;
     if( command==“cd” )
         执行系统调用，修改当前工作目录 u_dirp ;
       …… 识别、执行其它的内部命令 ……

     /* 装载应用程序：派发一个新进程，让这个进程承担执行应用程序 command 的任务 */
     while((i=fork( ))==-1);
     if( ! i)
          exec(“command\0”, “command\0”，arg1，arg2，……);
     else if(命令行中没有后台命令符号&)
          {  // 前台作业
             child = wait(&terminationStatus);     // shell 进程睡眠等待子进程终止，回收 PCB
             if（terminationStatus & 0xFF ！ = 0）     // terminationStatus 的 0~7 位是出错码
                根据出错码的值，输出段错误核心转储之类的报错信息
          }  //shell 进程不会睡眠等待负责执行后台作业的进程终止
   }
}
```

用户输入 logout，shell 进程终止。Shell 进程终止后。后台子进程会继续运行，它们的父进程是 1#进程。后台进程终止后，1#进程回收其 PCB。