

西北工业大学 操作系统实验 实验报告

班号： 10012006 姓名： 夏卓 学号： 2020303245

实验日期： 2022/11/6 实验名称： 解析 ELF 文件

一、实验目的

熟悉可执行链接文件（ELF）的结构，了解 GeekOS 将 ELF 格式的可执行程序加载到内存，建立内核线程并运行的实现技术。

二、实验要求

1.修改 Project1 项目中的/GeekOS/elf.c 文件:在函数 Parse_ELF_Executable()中添加代码，分析 ELF 格式的可执行文件（包括分析得出 ELF 文件头、程序头），获取可执行文件长度、代码段、数据段等信息，并打印输出。并且，填充 Exe_Format 数据结构中的值域。

2.掌握 GeekOS 在核心态运行可执行程序的原理，绘制出可执行程序在内核中加载、运行的流程图（需反映关键函数的调用关系）。

3.回答课后思考题。第八章第 1 题。

三、实验过程及结果

1.修改 Project1 项目中的/GeekOS/elf.c 文件:在函数 Parse_ELF_Executable()中添加代码，分析 ELF 格式的可执行文件（包括分析得出 ELF 文件头、程序头），获取可执行文件长度、代码段、数据段等信息，并打印输出。并且，填充 Exe_Format 数据结构中的值域。

首先查看 elf.h 中的 ELF 头部结构与程序头部结构：

```
typedef struct {
    unsigned int    type;
    unsigned int    offset;
    unsigned int    vaddr;
    unsigned int    paddr;
    unsigned int    fileSize;
    unsigned int    memSize;
    unsigned int    flags;
    unsigned int    alignment;
} programHeader;
```

```
typedef struct {
    unsigned char    ident[16];
    unsigned short    type;
    unsigned short    machine;
    unsigned int    version;
    unsigned int    entry;
    unsigned int    phoff;
    unsigned int    sphoff;
    unsigned int    flags;
    unsigned short    ehsize;
    unsigned short    phentsize;
    unsigned short    phnum;
    unsigned short    shentsize;
    unsigned short    shnum;
    unsigned short    shstrndx;
} elfHeader;
```

接着查看 elf.h 中的 exeFormat 结构：

```
struct Exe_Format {
    struct Exe_Segment segmentList[EXE_MAX_SEGMENTS]; /* Definition of segments */
    int numSegments; /* Number of segments contained in the executable */
    ulong_t entryAddr; /* Code entry point address */
};

struct Exe_Segment {
    ulong_t offsetInFile; /* Offset of segment in executable file */
    ulong_t lengthInFile; /* Length of segment data in executable file */
    ulong_t startAddress; /* Start address of segment in user memory */
    ulong_t sizeInMemory; /* Size of segment in memory */
    int protFlags; /* VM protection flags; combination of VM_READ, VM_WRITE, VM_EXECUTE */
};
```

由此发现解析 elf，只需读取出 elf 文件中的程序头，并将对应信息填入到 exeFormat 结构体中即可。首先让一个 elfHeader 类型的指针 elf_head 指向该地址，然后就可以用 elf_head 找到程序入口地址与程序头部表项个数，分别赋值给 exeFormat 的相应字段，接着利用 elf_head->phoff + exeFileData 得到程序头部表项起始地址，并利用 for 循环逐一读出各个 segmentList 需要的内容，其中第一段是代码段，第二段是数据段，最后将结果打印处理即可，具体步骤如下：

```
int Parse_ELF_Executable(char *exeFileData, ulong_t exeFileLength,
    struct Exe_Format *exeFormat)
{
    // TODO("Parse an ELF executable image");
    elfHeader *elf_head = (elfHeader *)exeFileData;
    exeFormat->entryAddr = elf_head->entry;
    exeFormat->numSegments = elf_head->phnum;
    programHeader *phdr = (programHeader *) (exeFileData + elf_head->phoff);
    Print("elf Header = 0x%x\n", (unsigned int) elf_head);
    Print("program Header = 0x%x\n", (unsigned int) phdr);
    Print("exeFile Length = %d\n", exeFileLength);
    int i;
    for (i = 0; i < exeFormat->numSegments; i++, phdr++)
    {
        struct Exe_Segment *segment = &exeFormat->segmentList[i];
        segment->offsetInFile = phdr->offset;
        segment->lengthInFile = phdr->fileSize;
        segment->startAddress = phdr->vaddr;
        segment->sizeInMemory = phdr->memSize;
        segment->protFlags = phdr->flags;
        if (i==0) Print("Code segment message:\n");
        else Print("Data segment message:\n");
        Print("offset In File = 0x%x\n", (unsigned int) phdr->offset);
        Print("length In File = %d\n", phdr->fileSize);
        Print("start Address in Memory = 0x%x\n", (unsigned int) phdr->vaddr);
        Print("size In Memory = %d\n", phdr->memSize);
    }
    return 0;
}
```

由此便完成了 elf 的解析任务，但是此时输出结果中无法显示第二句字符串：

```
Welcome to GeekOS!
Starting the Spawner thread...
elf Header = 0x1eb2d4
program Header = 0x1eb308
exeFile Length = 4990
Code segment message:
offset In File = 0x1000
length In File = 162
start Address in Memory = 0x1000
size In Memory = 162
Data segment message:
offset In File = 0x10c0
length In File = 40
start Address in Memory = 0x20c0
size In Memory = 40
Hi ! This is the first string
Hi ! This is the third (and last) string
If you see this you're happy

IPS: 87.781M  A: NUM CAPS SCRL HD:0-H
```

于是查看内核加载的源程序：

```
char s1[40] = "Hi ! This is the first string\n";

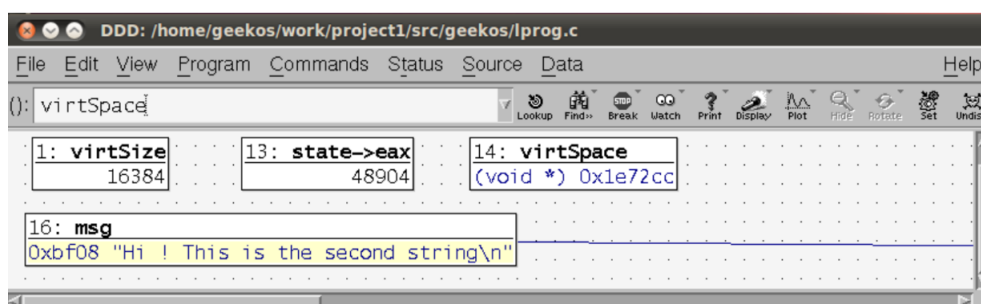
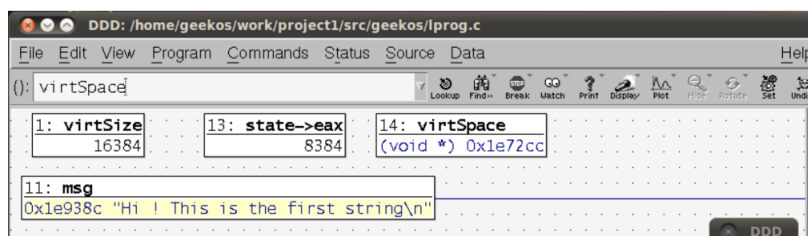
int main(int argc, char** argv)
{
    char s2[40] = "Hi ! This is the second string\n";

    ELF_Print(s1);
    ELF_Print(s2);

    return 0;
}
```

分析可知，这是由于 s2 属于带有初始值的局部变量，在编译过程中会直接将其存储在栈上，并利用可执行代码中的立即数进行初始化。而内核在加载该程序的过程中，虽然为其分配了栈页，但 esp 却没有改变，仍然指向内核栈。所以这段字符串并不在线程的内存空间中，因此无法在在线程栈中找到并输出它。

下面使用 ddd 调试工具进行进一步分析验证：

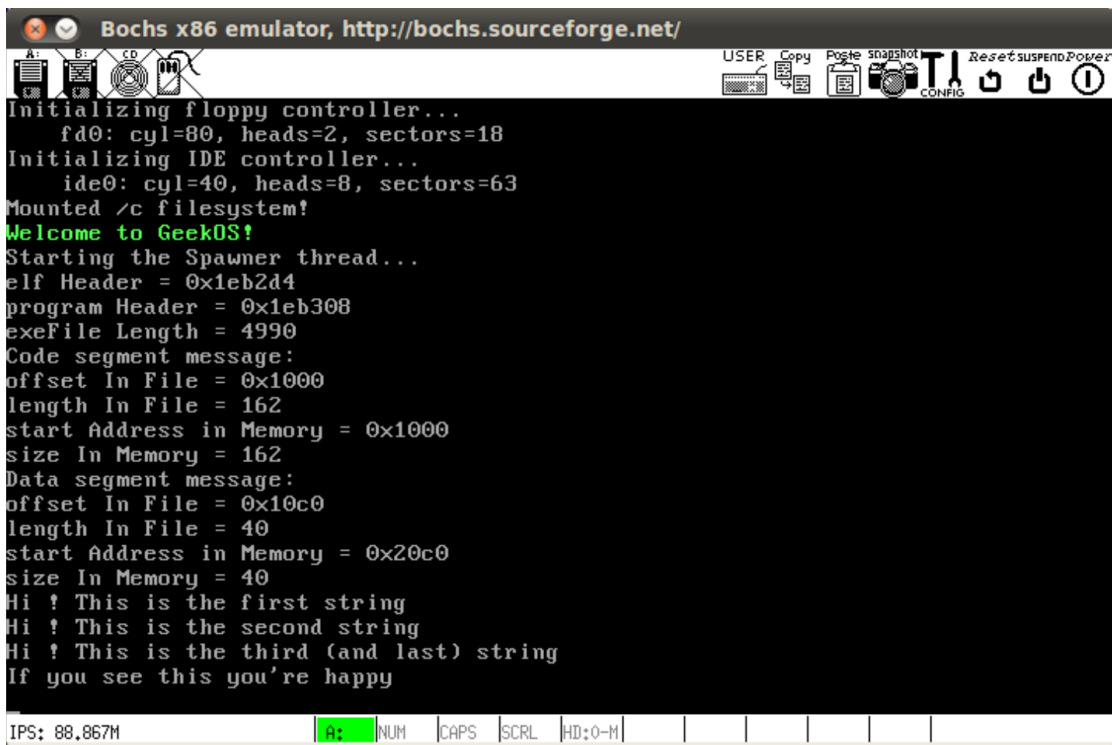


由图可以看出，第一段字符串传入 eax 寄存器的地址是字符串在线程内存空间中的偏移地址，只需加上 virtSpace 即可找到字符串；而第二段字符串传入 eax 寄存器的地址已经超过了分配给线程的内存空间大小，因此不可能在线程的内存空间中，而是在内核栈中，其地址即是指向第二段字符串的指针，因此可以直接访问到该字符串。

综上所述，为了能够输出第二段字符串，我们需要在 Printrap_Handler 函数中先判断传入 eax 寄存器中的地址所在的位置，如果是在线程栈中，则需要加上虚拟空间的基地址，否则若是在内核栈中则可直接输出，具体步骤如下：

```
static void Printrap_Handler( struct Interrupt_State* state )
{
    char *msg;
    if (state->eax <= virtSize)
        msg = (char *)virtSpace + state->eax;
    else
        msg = (char *)state->eax;
    Print(msg);
    g_needReschedule = true;
    return;
}
```

最终，程序运行结果如下图所示：



```
Bochs x86 emulator, http://bochs.sourceforge.net/
Initializing floppy controller...
  fd0: cyl=80, heads=2, sectors=18
Initializing IDE controller...
  ide0: cyl=40, heads=8, sectors=63
Mounted /c filesystem!
Welcome to GeekOS!
Starting the Spawner thread...
elf Header = 0x1eb2d4
program Header = 0x1eb308
exeFile Length = 4990
Code segment message:
offset In File = 0x1000
length In File = 162
start Address in Memory = 0x1000
size In Memory = 162
Data segment message:
offset In File = 0x10c0
length In File = 40
start Address in Memory = 0x20c0
size In Memory = 40
Hi ! This is the first string
Hi ! This is the second string
Hi ! This is the third (and last) string
If you see this you're happy

IPS: 88,867M  A: NUM CAPS SCRL HD:0-H
```

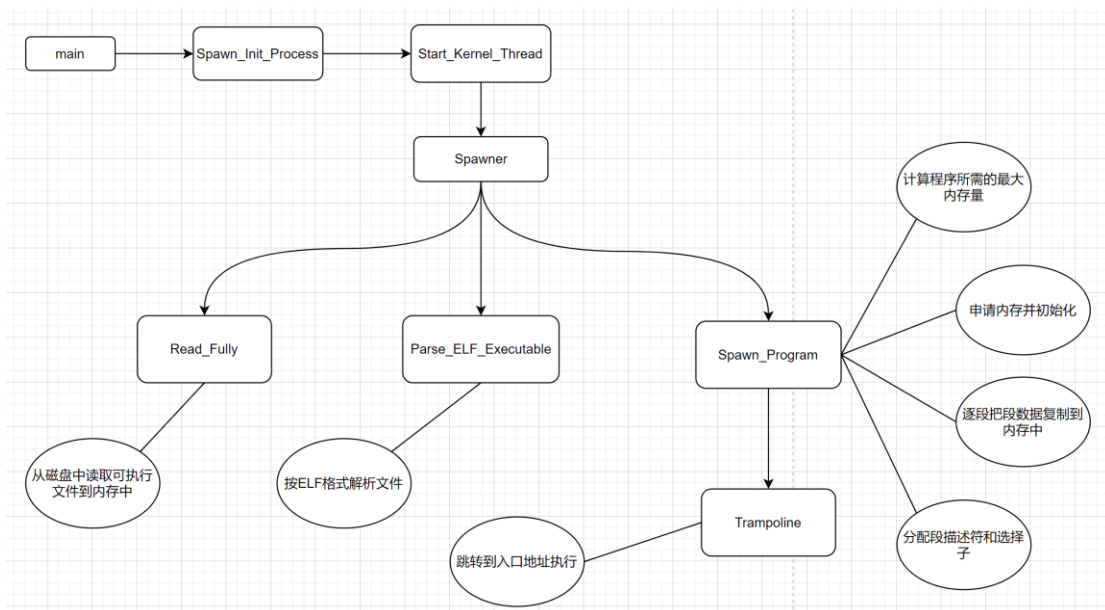
2.掌握 GeekOS 在核心态运行可执行程序的原理，绘制出可执行程序在内核中加载、运行的流程图（需反映关键函数的调用关系）。

ELF 文件保存在 GeekOS 的 PFAT 文件系统中，当文件系统挂载到系统中后我们就可以使用”/c/a.exe”来访问磁盘上的文件了。首先 main 函数会调用 `Spawn_Init_Process` 来创建一个新的线程，该函数直接调用了 `Start_Kernel_Thread` 创建一个名为 Spawner 的内核线程。

Spawner 函数的主要任务是从磁盘中读取可执行文件到内存中，然后按 ELF 格式解析文件，并将可执行文件的代码段和数据段复制到内存对应区域，之后建立线程执行装入内存相应的代码，最后需要释放分配的内存空间。为了完成这一系列工作，Spawner 函数调用了 `Read_Fully` 来将文件从磁盘读入内存，调用 `Parse_ELF_Executable` 来解析 ELF 文件，并调用 `Spawn_Program` 来启动对应线程。

具体来说，`Parse_ELF_Executable` 根据 ELF 文件的结构，将已经读入内存的可执行文件映像中相应的地址填充到结构 `exeFormat` 中；然后 `Spawn_Program` 需要根据 `exeFormat` 把可执行文件运行起来，首先找到最大的虚地址，由此计算出程序所需的最大内存量，申请相应大小的空间并进行初始化，接着逐段把段数据复制到内存中，每个段的位置由段的 `startAddress` 决定，接着分别给代码段和数据段分配段描述符和选择子，根据需要安装中断处理函数以扩展系统调用，最后调用 `Trampoline` 汇编程序以跳转到可执行程序所在的入口地址，就可以开始运行了。

具体流程图如下所示，其中圆角矩形表示关键函数，椭圆形表示函数实现的具体功能，箭头表示函数调用关系，线段表示功能的所属关系：



四、实验分析

本实验涉及 ELF 文件的解析与内核线程的加载问题。ELF 文件是一种用来描述可执行文件总体结构信息的二进制文件，其目标文件格式有链接视图和执行视图之分，具体结构如下图所示：

| 链接视图 | 执行视图 |
|-----------|-----------|
| ELF 头部 | ELF 头部 |
| 程序头部表（可选） | 程序头部表 |
| 节区 1 | 段 1 |
| ... | |
| 节区 n | 段 2 |
| ... | |
| ... | ... |
| 节区头部表 | 节区头部表（可选） |

本实验主要涉及执行视图的解读，与链接视图相比，其多了一个段的概念。链接器对编译生成的目标文件进行链接时，首先进行符号解析，找出外部符号在哪定义，接着链接器进行符号重定位。因为编译器在生成目标文件时，通常使用从零开始的相对地址，所以在链接过程中，链接器需要从一个指定的地址开始，根据输入目标文件的顺序，以段为单位将它们拼装起来。其中每个段可以包括很多个节。

另外，GeekOS 从磁盘加载一个内核线程的过程中，**Spawner** 函数发挥了很大的作用，它的主要功能就是从磁盘中读取可执行文件到内存中，然后按 ELF 格式解析文件，并将可执行文件的代码段和数据段复制到内存对应区域，之后建立线程，执行装入内存相应的代码，执行完毕后需要释放分配的内存空间。

五、所遇问题及解决方法

实验中编写 `Parse_ELF_Executable` 函数还是比较简单的，只需要对比 ELF 头部结构和程序头部结构与 `exeFormat` 结构的关系，然后进行简单的赋值即可。但是仅仅改写此函数并不能输出第二句字符串，观察加载的源代码可以发现它与第一句的区别就在于其是局部变量。结合课本分析可知该局部变量被存储在了栈中，但不是分配给线程的栈里，由此推知它很有可能存储在了内核栈中。在使用 `ddd` 进行调试的时候，由于很少使用此类调试工具因而

花了些时间在熟悉上，但老师讲解过后感觉清晰了很多。查看传入到 `eax` 寄存器的值可以发现第二句的地址已经超过了分配给线程的内存大小，有理由相信其确实是位于内核栈中，输出结果也验证了这一点。于是我通过比较 `eax` 与内存空间大小来判断字符串所在位置，从而实现了第二句字符串的输出。

六、思考与练习

1. 在程序中输出字符串为什么要通过中断实现？直接调用 `Print` 函数是否可行？

直接调用 `Print` 函数会报错：

```
ld -o user/a.exe -Ttext 0x1000 -e _Entry \
    libc/entry.o user/a.o libc/libc.a
user/a.o: In function `main':
a.c:(.text+0x61): undefined reference to `Print'
a.c:(.text+0x6d): undefined reference to `Print'
make: *** [user/a.exe] Error 1
rm libc/entry.o
```

这是因为 `Print` 函数是内核中实现的函数，并不在动态链接库中，外部可执行程序若想要使用则必须进行系统调用，而进行系统功能调用则需要通过中断实现，因此在程序中输出字符串要使用 `ELF_Print` 而不能直接使用 `Print`。