厦門大學

信息学院软件工程系

《实用操作系统》实验报告

题　　目　ELF文件格式与虚拟内存实验设计

任课老师　李贵林

组员姓名　周彦妮　任宇　向泽旭　左泽宇　侯妤欣　曾予岑

提交时间　2023 年 12 月 06 日

ELF文件格式与虚拟内存实验设计

一、 实验目的

- 根据ELF文件格式，修改ELF文件，实现预期效果。
- 观察ELF文件中各个段如何被映射到虚拟内存中。

二、 实验环境

- 操作系统：
  - 主机：Windows 10
  - 虚拟机：Ubuntu 18.04
- 开发板：IMAX6ULL MIN
- 文件传输工具：FileZilla
- 终端工具：MobaXterm

三、 实验思路

- 对于实验目的一，通过学习ELF文件相关知识，我们可以知道：
  - ELF（Executable and Linkable Format）文件是一种广泛使用的文件格式，用于定义可执行文件、目标代码、共享库和核心转储，它是标准的二进制文件格式。它主要有以下几个组成部分：
    - 头部：描述了整个文件的布局和属性，包括魔数、文件类型（如可执行文件、共享对象）、机器类型（如 x86、ARM）、入口点地址（程序开始执行的地方）等。
    - 程序头表：描述了程序运行时需要的各个段，这些段在加载时被映射到虚拟内存中，同时也包括段的类型、偏移、虚拟地址、物理地址、文件大小、内存大小、权限和对齐。
    - 节头表：描述了文件中的节（如代码、数据、符号表、重定位信息）。主要用于链接和调试。
    - 节：包含程序的实际数据，如代码（.text）、数据（.data、.bss）、符号表（.symtab）、字符串表（.strtab）、重定位信息等。
    - 段：由程序头表描述，用于程序的加载。一个段可以包含多个节，它描述了如何将节映射到内存中。

  基于这些基本了解，本次实验将对ELF文件中的text(代码)段和data(数据)段进行修改，我们首先需要找到其在ELF文件中的位置，然后进行我们期望的修改以实现预期的结果。

- 对于实验目的二，我们则是通过阅读LiteOS-A中实现ELF加载和执行的源代码，找出并修改其中执行内存映射的函数，增加打印功能，以便我们观察ELF文件中指出的地址与实际映射的虚拟内存地址间的关系。

四、 实验步骤

1. 修改ELF文件

1) 这部分实验研究如何针对ELF可执行文件实现代码注入，这是一个简单的示例，但却可以帮助理解ELF文件的格式。

首先编写一段代码，然后将其编译成可执行的ELF文件：

```c
#include <stdio.h>

int array_size = 15; // 全局变量，被错误赋值

// 冒泡排序函数
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] < arr[j+1]) { // 逻辑错误：应该是 arr[j] > arr[j+1]
                // 交换 arr[j] 和 arr[j+1]
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

// 主函数
int main() {
    int arr[10] = {64, 34, 25, 12, 22, 11, 90, 88, 76, 45};
    int n=10;
    bubbleSort(arr, array_size);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

```
book@ry-virtual-machine:~$ clang -target arm-liteos --sysroot=/home/book/openharmony/prebuilts/lite/sysroot/ \-o test_program test_program.c
book@ry-virtual-machine:~$
```

这个程序有一个明显的错误，那就是全局变量array_size被错误赋值，这会导致越界问题。

2) 全局变量被存储在data段中，为了解决这个问题，我们首先需要找出data段以及array_size变量在ELF文件中的位置，使用readelf文件检查文件(文件名为test_program)：

```
book@ry-virtual-machine:~$ readelf -a test_program
ELF 头：
  Magic:    7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF32
  数据:                              2 补码，小端序 (little endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              DYN (共享目标文件)
  系统架构:                          ARM
  版本:                              0x1
  入口点地址:               0x1000
```

查看节头部分，发现data段相对文件起始位置的偏移是0x3000：

```
节头：
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00000194 000194 000016 00   A  0   0  1
  [ 2] .dynsym           DYNSYM          000001ac 0001ac 000070 10   A  5   1  4
  [ 3] .gnu.hash         GNU_HASH        0000021c 00021c 000020 00   A  2   0  4
  [ 4] .hash             HASH            0000023c 00023c 000040 04   A  2   0  4
  [ 5] .dynstr           STRTAB          0000027c 00027c 000053 00   A  0   0  1
  [ 6] .rel.dyn          REL             000002d0 0002d0 000040 08   A  2   0  4
  [ 7] .ARM.exidx        ARM_EXIDX       00000310 000310 000010 00  AL 12   0  4
  [ 8] .rel.plt          REL             00000320 000320 000020 08   A  2  20  4
  [ 9] .rodata           PROGBITS        00000340 000340 00003e 00 AMS  0   0  4
  [10] .eh_frame_hdr     PROGBITS        00000380 000380 00000c 00   A  0   0  4
  [11] .eh_frame         PROGBITS        0000038c 00038c 000004 00   A  0   0  4
  [12] .text             PROGBITS        00001000 001000 000318 00  AX  0   0  4
  [13] .init             PROGBITS        00001318 001318 00000c 00  AX  0   0  1
  [14] .fini             PROGBITS        00001324 001324 00000c 00  AX  0   0  1
  [15] .plt              PROGBITS        00001330 001330 000060 00  AX  0   0 16
  [16] .init_array       INIT_ARRAY      00002000 002000 000004 00  WA  0   0  4
  [17] .fini_array       FINI_ARRAY      00002004 002004 000004 00  WA  0   0  4
  [18] .dynamic          DYNAMIC         00002008 002008 0000c8 08  WA  5   0  4
  [19] .got              PROGBITS        000020d0 0020d0 000014 00  WA  0   0  4
  [20] .got.plt          PROGBITS        000020e4 0020e4 00001c 00  WA  0   0  4
  [21] .bss.rel.ro       NOBITS          00002100 002100 000000 00  WA  0   0  1
  [22] .data             PROGBITS        00003000 003000 000008 00  WA  0   0  4
  [23] .bss              NOBITS          00003008 003008 000025 00  WA  0   0  4
  [24] .comment          PROGBITS        00000000 003008 0000db 01  MS  0   0  1
  [25] .ARM.attributes   ARM_ATTRIBUTES  00000000 0030e3 000038 00      0   0  1
```

接着查看符号表部分，找到array_size符号，可以发现其位置为0x3004：

```
    47: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __register_frame_in
    48: 00003004     4 OBJECT  GLOBAL DEFAULT   22 array_size
    49: 00001154   228 FUNC    GLOBAL DEFAULT   12 bubbleSort
    50: 00000000     0 FUNC    GLOBAL DEFAULT  UND printf
```

3) 使用HexEdit修改ELF文件(hexedit test_program)，并将程序加载到liteos.bin文件中，接着在开发板上运行观察结果：

可以发现，test_program文件中0x3004的位置确实存储了array_size的值（15），我们将其修改为10。

```
00003000   00 30 00 00   0F 00 00 00   4C 69 6E 6B   65 72 3A 20
00003020   2F 33 34 30   32 34 2D 6D   65 72 67 65   64 2F 74 6F
00003040   6A 65 63 74   2F 6C 6C 64   20 63 32 30   63 64 35 66
00003060   39 61 30 64   66 37 36 34   39 39 62 65   66 61 61 34
---  test_program        --0x3004/0x3B2C----------------------
```

可以发现，test_program文件中0x3004的位置确实存储了array_size的值（15），我们将其修改为10：

```
00003000   00 30 00 00   0A 00 00 00   4C 69 6E 6B
00003020   2F 33 34 30   32 34 2D 6D   65 72 67 65
00003040   6A 65 63 74   2F 6C 6C 64   20 63 32 30
00003060   39 61 30 64   66 37 36 34   39 39 62 65
-**  test_program        --0x3005/0x3B2C----------
```

可见程序正确运行：

```
OHOS # ./bin/test_program
OHOS # Sorted array:
90 88 76 64 45 34 25 22 12 11

OHOS #
```

4) 现在test_program实现的是降序排序，如果我们想实现升序排序且不更改并编译源文件，我们该怎么做呢？一种简单的方法是修改ELF文件中代码段的机器指令，首先使用objdump命令反汇编test_program，找出实现BubbleSort函数的部分，接着确定待修改指令的位置：

```
book@ry-virtual-machine:~$ arm-linux-gnueabi-objdump -d test_program

test_program:     文件格式 elf32-littlearm


Disassembly of section .text:

00001000 <_start>:
    1000:       e3a0b000        mov     fp, #0
    1004:       e3a0e000        mov     lr, #0
    1008:       e59f1010        ldr     r1, [pc, #16]   ; 1020 <_start+0x20>
    100c:       e08f1001        add     r1, pc, r1
    1010:       e1a0200d        mov     r2, sp
```

```
    11bc:       e5900004        ldr     r0, [r0, #4]
    11c0:       e1510000        cmp     r1, r0
    11c4:       aa00000f        bge     1208 <bubbleSort+0xb4>
    11c8:       eaffffff        b       11cc <bubbleSort+0x78>
    11cc:       e59d0010        ldr     r0, [sp, #16]
```

在这里，cmp r1, r0指令比较r1和r0寄存器的值，这两个寄存器分别加载了 arr[j] 和 arr[j+1] 的值。紧接着的 bge 指令是一个条件跳转，它基于比较的结果来决定是否跳转到指定的地址。在当前的降序排序实现中，如果 arr[j] 大于等于 arr[j+1]，则跳转继续下一个循环迭代。要改为升序排序，我们需要在 arr[j] 小于 arr[j+1] 时进行交换。因 此，需要更改 bge 指令为 blt 指令，它会在 arr[j] 小于 arr[j+1] 时跳转。机器码中，bge 指令的操作码是 AA，而 jl 指令的操作码是 DA。 同时可以得到的信息是，这个命令位于test_program的0x11C4位置。

5) 使用HexEdit修改ELF文件，并将程序加载到liteos.bin文件中，接着在开发板上运行观察结果：

```
00001180   FF FF FF EA  00 00 A0 E3  04 00 8D E5
000011A0   01 10 82 E0  01 00 50 E1  1B 00 00 AA
000011C0   00 00 51 E1  0F 00 00 AA  FF FF FF EA
-**  test_program       --0x11C8/0x3B2C----------
```

```
000011C0   00 00 51 E1  0F 00 00 DA  FF FF FF EA  10 00 9D E
000011E0   04 10 9D E5  01 21 80 E0  04 20 92 E5  01 21 80 E
00001200   04 00 81 E5  FF FF FF EA  FF FF FF EA  04 00 9D E
00001220   08 00 9D E5  01 00 80 E2  08 00 8D E5  CE FF FF E
00001240   40 D0 4D E2  00 00 A0 E3  14 00 0B E5  B0 10 9F E
00001260   70 50 A3 E8  70 50 91 E8  70 50 83 E8  0A 10 A0 E
00001280   02 00 A0 E1  B2 FF FF EB  7C 00 9F E5  00 00 8F E
---  test_program       --0x11C7/0x3B2C----------
```

```
OHOS # ./bin/test_program
OHOS # Sorted array:
11 12 22 25 34 45 64 76 88 90

OHOS #
```

可以发现程序成功改为升序排序。

2. 观察ELF如何映射到虚拟内存

1) 修改los_load_elf.c文件，为其增加一个测试用的函数，当加载特定名称的ELF文件时打印段地址（此处为test_program），修改OsLoadELFSegment函数，增加下图所示代码：

```
    if(!strcmp(loadInfo->fileName,"bin/test_program"))
    ret = OsMmapELFFileTest(loadInfo->execFD, loadInfo->elfPhdr, &loadInfo->elfEhdr, &loadInfo->loadAddr, mapSize,
                        &loadBase);
    else
    ret = OsMmapELFFile(loadInfo->execFD, loadInfo->elfPhdr, &loadInfo->elfEhdr, &loadInfo->loadAddr, mapSize,
                        &loadBase);
```

2) 编写测试函数，这里的测试函数参照OsMmapELFFile函数设计，只不过在每次循环中打印段地址：

```
STATIC INT32 OsMmapELFFileTest(INT32 fd, const LD_ELF_PHDR *elfPhdr, const LD_ELF_EHDR *elfEhdr, UINTPTR *elfLoadAddr,
                        UINT32 mapSize, UINTPTR *loadBase)
{
    const LD_ELF_PHDR *elfPhdrTemp = elfPhdr;
    UINTPTR vAddr, mapAddr, bssStart;
    UINT32 bssEnd, elfProt, elfFlags;
    INT32 ret, i;

    for (i = 0; i < elfEhdr->elfPhNum; ++i, ++elfPhdrTemp) {
        if (elfPhdrTemp->type != LD_PT_LOAD) {
            continue;
        }
        if (elfEhdr->elfType == LD_ET_EXEC) {
            if (OsVerifyELFPhdr(elfPhdrTemp) != LOS_OK) {
                return -ENOEXEC;
            }
        }

        elfProt = OsGetProt(elfPhdrTemp->flags);
        if ((elfProt & PROT_READ) == 0) {
            return -ENOEXEC;
        }
        elfFlags = MAP_PRIVATE;
        vAddr = elfPhdrTemp->vAddr;

        mapAddr = OsDoMmapFile(fd, (vAddr + *loadBase), elfPhdrTemp, elfProt, elfFlags, mapSize);
        PRINT_ERR("Mapping segment %d at address: 0x%x\n", i, mapAddr);
        if (!LOS_IsUserAddress((VADDR_T)mapAddr)) {
            return -ENOMEM;
        }
#ifdef LOSCFG_DRIVERS_TZDRIVER
        if ((elfPhdrTemp->flags & PF_R) && (elfPhdrTemp->flags & PF_X) && !(elfPhdrTemp->flags & PF_W)) {
            SetVmmRegionCodeStart(vAddr + *loadBase, elfPhdrTemp->memSize);
        }
#endif
        mapSize = 0;

        if (*elfLoadAddr == 0) {
            *elfLoadAddr = mapAddr + ROUNDOFFSET(vAddr, PAGE_SIZE);
        }

        if ((*loadBase == 0) && (elfEhdr->elfType == LD_ET_DYN)) {
            *loadBase = mapAddr;
            elfFlags |= MAP_FIXED;
        }

        if ((elfPhdrTemp->memSize > elfPhdrTemp->fileSize) && (elfPhdrTemp->flags & PF_W)) {
            bssStart = mapAddr + ROUNDOFFSET(vAddr, PAGE_SIZE) + elfPhdrTemp->fileSize;
            bssEnd = mapAddr + ROUNDOFFSET(vAddr, PAGE_SIZE) + elfPhdrTemp->memSize;
            ret = OsSetBss(elfPhdrTemp, fd, bssStart, bssEnd, elfProt);
            if (ret != LOS_OK) {
                return ret;
            }
        }
    }

    return LOS_OK;
```

为了方便在用户态下观察打印信息，这里使用PRINT_ERR输出。函数会遍历 ELF 文件的程序头数组并过滤非加载段。函数仅处理类型为 LD_PT_LOAD 的段，这些是需要被加载到内存的段。接着会使用 OsDoMmapFile 函数将段映射到虚拟内存。映射地址存储在 mapAddr 变量中。如果返回正确，对于每个映射的段，打印映射的段号和地址，以便于调试和验证。

3) 编译内核，并加载到开发板中运行：

```
OHOS # ./bin/test_program
OHOS # [ERR]Mapping segment 2 at address: 0x2000000
[ERR]Mapping segment 3 at address: 0x2001000
[ERR]Mapping segment 4 at address: 0x2002000
[ERR]Mapping segment 5 at address: 0x2003000
Sorted array:
11 12 22 25 34 45 64 76 88 90
```

4） 观察输出结果，并于ELF文件中指出的虚拟地址作比较：

```
程序头：
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x00000194 0x00000194 0x00016 0x00016 R   0x1
      [Requesting program interpreter: /lib/ld-musl-arm.so.1]
  LOAD           0x000000 0x00000000 0x00000000 0x00390 0x00390 R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x00390 0x00390 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x00100 0x00100 RW  0x1000
  LOAD           0x003000 0x00003000 0x00003000 0x00008 0x0002d RW  0x1000
  DYNAMIC        0x002008 0x00002008 0x00002008 0x000c8 0x000c8 RW  0x4
  GNU_RELRO      0x002000 0x00002000 0x00002000 0x00100 0x01000 R   0x1
  GNU_EH_FRAME   0x000380 0x00000380 0x00000380 0x0000c 0x0000c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0
  EXIDX          0x000310 0x00000310 0x00000310 0x00010 0x00010 R   0x4
```

观察ELF的程序头，我们可以发现0x00000000、0x00001000、0x00002000、0x00003000 这些地址被映射到了0x2000000、0x2001000、0x2002000、0x2003000。也就是说在加载时，加载器将ELF文件中指出的这些地址映射到了实际的虚拟内存地址上。这是因为操作系统为每个进程提供了一个独立的虚拟地址空间。ELF 文件中指定的虚拟地址是相对于这个地址空间的起始地址。同时还要注意一种情况，那就是操作系统加载器在将程序加载到内存时有可能进行地址重定位。这是一种常见的情况，尤其是在支持地址空间布局随机化（ASLR）的系统中。

5） 补充：在Ubuntu18.04中的ELF文件与LiteOs-A中的ELF文件有何不同？

首先最明显的区别就是两者格式不同，但这是由于编译导致的：

LiteOS-A：

```
ELF 头：
  Magic：   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF32
  数据：                             2 补码，小端序 (little endian)
  版本：                             1 (current)
  OS/ABI：                           UNIX - System V
  ABI 版本：                         0
  类型：                             DYN (共享目标文件)
  系统架构：                         ARM
  版本：                             0x1
  入口点地址：               0x1000
  程序头起点：          52 (bytes into file)
  Start of section headers:          13988 (bytes into file)
```

Ubuntu18.04:

```
ELF 头：
  Magic：   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF64
  数据：                             2 补码，小端序 (little endian)
  版本：                             1 (current)
  OS/ABI：                           UNIX - System V
  ABI 版本：                         0
  类型：                             DYN (共享目标文件)
  系统架构：                         Advanced Micro Devices X86-64
  版本：                             0x1
  入口点地址：               0x630
```

我们可以观察Ubuntu18.04平台上能运行的ELF文件头，会发现其也为各个段指定了映射到哪个虚拟地址：

```
程序头:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000b08 0x0000000000000b08  R E    0x200000
  LOAD           0x0000000000000da0 0x0000000000200da0 0x0000000000200da0
                 0x0000000000000274 0x0000000000000278  RW     0x200000
  DYNAMIC        0x0000000000000db0 0x0000000000200db0 0x0000000000200db0
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000998 0x0000000000000998 0x0000000000000998
```

其中offset是指相对于ELF文件的偏移量，而VirtAddr则是指定的虚拟内存地址，我们运行这个程序并观察其内存分布：

```
book@ry-virtual-machine:~$ ./test_program
Sorted array:
11 12 22 25 34 45 64 76 88 90 ^Z
[1]+  已停止(SIGTSTP)      ./test_program
book@ry-virtual-machine:~$ ./test_program
Sorted array:
11 12 22 25 34 45 64 76 88 90 ^Z
[2]+  已停止(SIGTSTP)      ./test_program
book@ry-virtual-machine:~$ ps
    PID TTY          TIME CMD
  60545 pts/1    00:00:00 test_program
  60588 pts/1    00:00:00 test_program
  60620 pts/1    00:00:00 ps
 113112 pts/1    00:00:00 sh
```

执行两个test_program进程，并查看其/proc/pid/maps：

```
book@ry-virtual-machine:~$ cat /proc/60545/maps
55c5855f5000-55c5855f6000 r-xp 00000000 08:01 944394                     /home/book/test_program
55c5857f5000-55c5857f6000 r--p 00000000 08:01 944394                     /home/book/test_program
55c5857f6000-55c5857f7000 rw-p 00001000 08:01 944394                     /home/book/test_program
55c587086000-55c5870a7000 rw-p 00000000 00:00 0                          [heap]
7fe98d2bc000-7fe98d4a3000 r-xp 00000000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7fe98d4a3000-7fe98d6a3000 ---p 001e7000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7fe98d6a3000-7fe98d6a7000 r--p 001e7000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7fe98d6a7000-7fe98d6a9000 rw-p 001eb000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7fe98d6a9000-7fe98d6ad000 rw-p 00000000 00:00 0
7fe98d6ad000-7fe98d6d6000 r-xp 00000000 08:01 403249                     /lib/x86_64-linux-gnu/ld-2.27.so
7fe98d8bb000-7fe98d8bd000 rw-p 00000000 00:00 0
7fe98d8d6000-7fe98d8d7000 r--p 00029000 08:01 403249                     /lib/x86_64-linux-gnu/ld-2.27.so
7fe98d8d7000-7fe98d8d8000 rw-p 0002a000 08:01 403249                     /lib/x86_64-linux-gnu/ld-2.27.so
7fe98d8d8000-7fe98d8d9000 rw-p 00000000 00:00 0
7fff49607000-7fff49628000 rw-p 00000000 00:00 0                          [stack]
7fff49709000-7fff4970c000 r--p 00000000 00:00 0                          [vvar]
7fff4970c000-7fff4970e000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

```
book@ry-virtual-machine:~$ cat /proc/60588/maps
55e35b6a3000-55e35b6a4000 r-xp 00000000 08:01 944394                     /home/book/test_program
55e35b8a3000-55e35b8a4000 r--p 00000000 08:01 944394                     /home/book/test_program
55e35b8a4000-55e35b8a5000 rw-p 00001000 08:01 944394                     /home/book/test_program
55e35cc37000-55e35cc58000 rw-p 00000000 00:00 0                          [heap]
7f2da3bd5000-7f2da3dbc000 r-xp 00000000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7f2da3dbc000-7f2da3fbc000 ---p 001e7000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7f2da3fbc000-7f2da3fc0000 r--p 001e7000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7f2da3fc0000-7f2da3fc2000 rw-p 001eb000 08:01 403253                     /lib/x86_64-linux-gnu/libc-2.27.so
7f2da3fc2000-7f2da3fc6000 rw-p 00000000 00:00 0
7f2da3fc6000-7f2da3fef000 r-xp 00000000 08:01 403249                     /lib/x86_64-linux-gnu/ld-2.27.so
7f2da41d4000-7f2da41d6000 rw-p 00000000 00:00 0
7f2da41ef000-7f2da41f0000 r--p 00029000 08:01 403249                     /lib/x86_64-linux-gnu/ld-2.27.so
7f2da41f0000-7f2da41f1000 rw-p 0002a000 08:01 403249                     /lib/x86_64-linux-gnu/ld-2.27.so
7f2da41f1000-7f2da41f2000 rw-p 00000000 00:00 0
7ffd3e363000-7ffd3e384000 rw-p 00000000 00:00 0                          [stack]
7ffd3e3df000-7ffd3e3e2000 r--p 00000000 00:00 0                          [vvar]
7ffd3e3e2000-7ffd3e3e4000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

可以发现尽管执行的是同一个ELF文件，但是装入到虚拟内存时却是装入了不同的地址。选择一个进行观察：
55c5855f5000-55c5855f6000 r-xp /home/book/test_program
//这个段标记为 r-xp，即可读、可执行，不可写。它对应于包含程序代码的 LOAD 段。在程序头表中，第一个 LOAD 段（位于文件偏移 0x00000000）是可执行的（标记为 RE），与这个段匹配。
55c5857f6000-55c5857f7000 rw-p /home/book/test_program
//对应ELF文件头中的数据段，0x201000+55c5855f5000=55c5857f6000

五、 实验总结

本次实验的目标是深入理解ELF（Executable and Linkable Format）文件格式，并观察ELF文件如何被映射到虚拟内存中。我们首先学习了ELF文件的基本组成部分，包括文件头部、程序头表、节头表和各个节。特别地，我们关注了ELF文件中的text（代码）段和data（数据）段，以及这些段如何映射到内存中。

实验的第一部分涉及对ELF文件进行修改。我们通过修改data段中的变量值和text段中的机器指令，实现了对程序行为的控制。例如，我们成功修改了全局变量的值，以及改变了程序的排序算法。这不仅展示了ELF文件的灵活性，也加深了我们对程序编译和链接过程的理解。

第二部分的实验重点在于观察ELF文件映射到虚拟内存的过程。通过修改LiteOS-A内核的源代码和增加打印功能，我们能够观察到不同段的映射地址。这一过程帮助我们理解了操作系统如何将程序文件中的地址映射到进程的地址空间，以及地址重定位的机制。

总体而言，这次实验不仅加深了我们对ELF文件格式的理解，还让我们对程序如何被操作系统加载到内存中有了更加直观的认识。通过实际操作和观察，我们更加深刻地理解了程序运行时的内存管理和映射机制。此外，实验过程中的挑战也提升了我们的问题解决能力和团队合作技巧。

六、 遇到的问题及如何解决

1. 在反汇编test_program文件时，发现使用objdump并不能够处理arm架构的文件，解决方案是：
   安装 ARM 架构的交叉编译工具链。对于 ARM，通常的包是 binutils-arm-linux-gnueabi。使用 ARM 版本的 objdump 来反汇编ELF 文件：
   arm-linux-gnueabi-objdump -d test_program

2. 在修改los_load_elf.c文件时发现，如果只是在内核代码中增加PRINTK函数是无法在用户态下输出信息的，解决方案是：
   使用PRINT_ERR宏进行打印，这样就可以在用户态下输出信息且不会影响程序的正常运行。

七、 参考文献

1. 深入浅出ELF - 知乎 (zhihu.com)
2. 一个elf程序实现代码注入的实例-腾讯云开发者社区-腾讯云 (tencent.com)
3. kernel_liteos_a: LiteOS kernel for embedded devices with rich resources | 适用于资源较丰富嵌入式设备的LiteOS内核 (gitee.com)