实验8 Linux进程地址空间

实验目的

1. 理解Linux中程序汇编和链接完成的任务
2. 理解Linux进程的虚拟地址空间构成
3. 理解程序从编辑到执行各阶段的地址空间变化

实验内容

1. 分别编写以下两个程序sum.c和main.c，然后分别编译生成.o目标文件，然后链接生成可执行程序，使用反汇编objdump命令查看，分析编译和链接的工作，.o目标文件和可执行程序中的逻辑地址有何不同。

sum.c:

int sum(int \*a ,int n)

{ int i,s=0;

for (i=0;i<n;i++)

{ s += a[i];

}

return s;

}

main.c：

int sum(int \*a ,int n);

int array[2] = {1,2};

int main()

{

int val = sum(array,2);

return val;

}

实验步骤：

1. 分别用gcc -c sum.c和gcc -c main.c编译汇编两个源程序文件，生成sum.o和main.o可重定位目标文件；
2. 分别使用objdump -d sum.o和objdump -d main.o, readelf -r main.o查看两个文件的反汇编代码，**分析逻辑地址特点以及需要重定位的地址**；

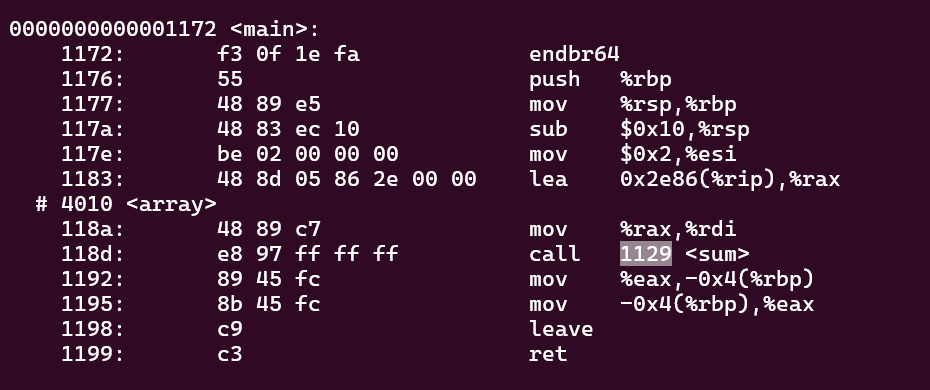


可以看到在两个可重定位目标文件中，最左侧的逻辑地址都是从0开始的，在main.o中的函数指令调用call 20对应源程序的return val；其中的0x20表示调用的函数起始位置在偏移量是0x20开始的四个字节记录，即“00 00 00 00”这里记录函数调用的起始位置，因为调用的是外部函数，需要在链接时确定调用位置。

1. 使用gcc -o main sum.o main.o生成可执行目标文件main，使用objdump反汇编，**分析可执行文件中的地址特点，并分析地址重定位如何完成的**；

结合本实验，总结汇编后的可重定位目标文件和链接后的可执行目标文件有何不同，分析链接器完成的任务。

要求：上传每一步截图并对结果分析



可以看到main中的call指令的地址不再是相对偏移量而是0x1129,这是sum函数在可执行文件中的实际地址。

可执行文件中的逻辑地址是实际的内存地址，不再是从0开始，main函数中的地址为0000000000001172。

链接器完成了重定位，将外部相对地址替换为了实际地址，确保所有符号和函数调用的地址一致。

2.（选做）编译运行以下程序，分析进程的虚拟地址空间构成。

#include<stdio.h>

#include<malloc.h>

#include<unistd.h>

int bss\_var0;

int bss\_var1=0;

int data\_var0=1;

const int a=10;

int main(int argc,char \*\*argv)

{

printf("Now is the process's virtual memory\n");

printf("Text location:\n");

printf("\tAddress of main(Code Segment):%p\n",main);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

int stack\_var0=2;

printf("Stack Location:\n");

printf("\tInitial end of stack(int stack\_var0=2):%p\n",&stack\_var0);

int stack\_var1=3;

printf("\tnew end of stack(int stack\_var1=3):%p\n",&stack\_var1);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Data Location:\n");

printf("\tAddress of data\_var0(int data\_var0=1)(Data Segment):%p\n",&data\_var0);

static int data\_var1=4;

printf("\tNew end of data\_var1(static int data\_var1=4)(Data Segment):%p\n",&data\_var1);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("BSS Location:\n");

printf("\tAddress of bss\_var(int bss\_var0):%p\n",&bss\_var0);

printf("\tAddress of bss\_var(static int bss\_var1):%p\n",&bss\_var1);

static int bss\_var2;

printf("\tAddress of bss\_var(static int bss\_var1):%p\n",&bss\_var2);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

int \*heap\_var0 = malloc(sizeof(int)\*2);

printf("Heap Location:\n");

printf("\tAddress of heap\_vari(int \*heap\_var0):%p\n",heap\_var0);

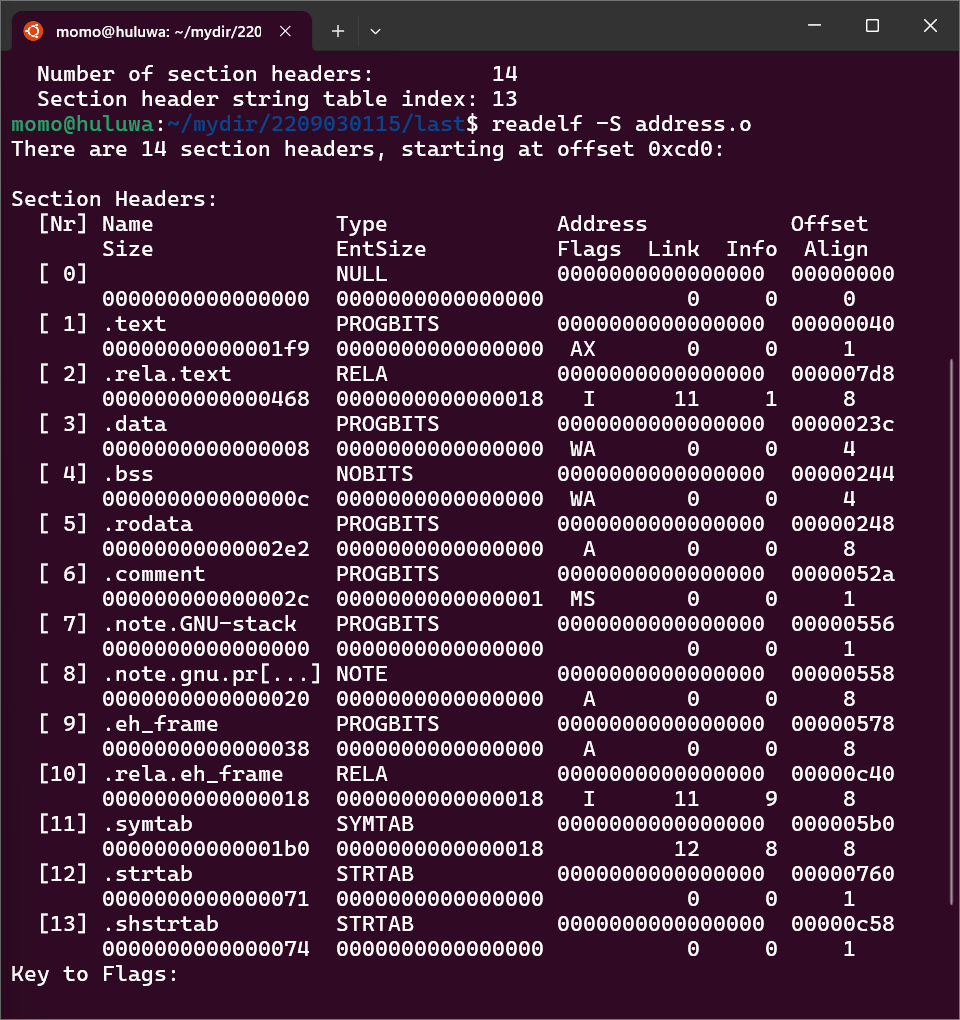
getchar();

return 0;

}

实验步骤：

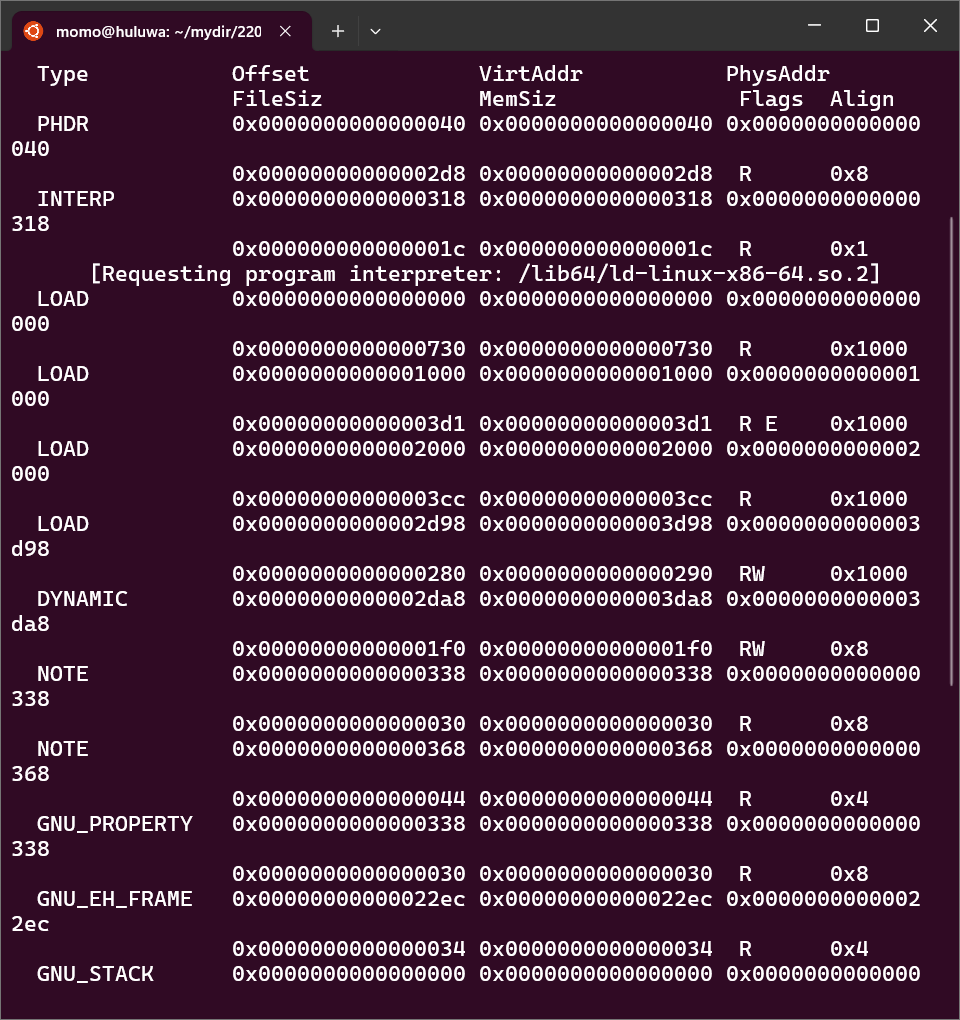
1. 编译链接该源程序文件，生成可重定位目标文件address.o和可执行目标文件address；
2. 使用readelf -h address.o和readelf -S address.o命令，查看可重定位目标文件address.o的ELF文件的链接视图，分析该文件有多少个节，哪些节需要分配存储空间；



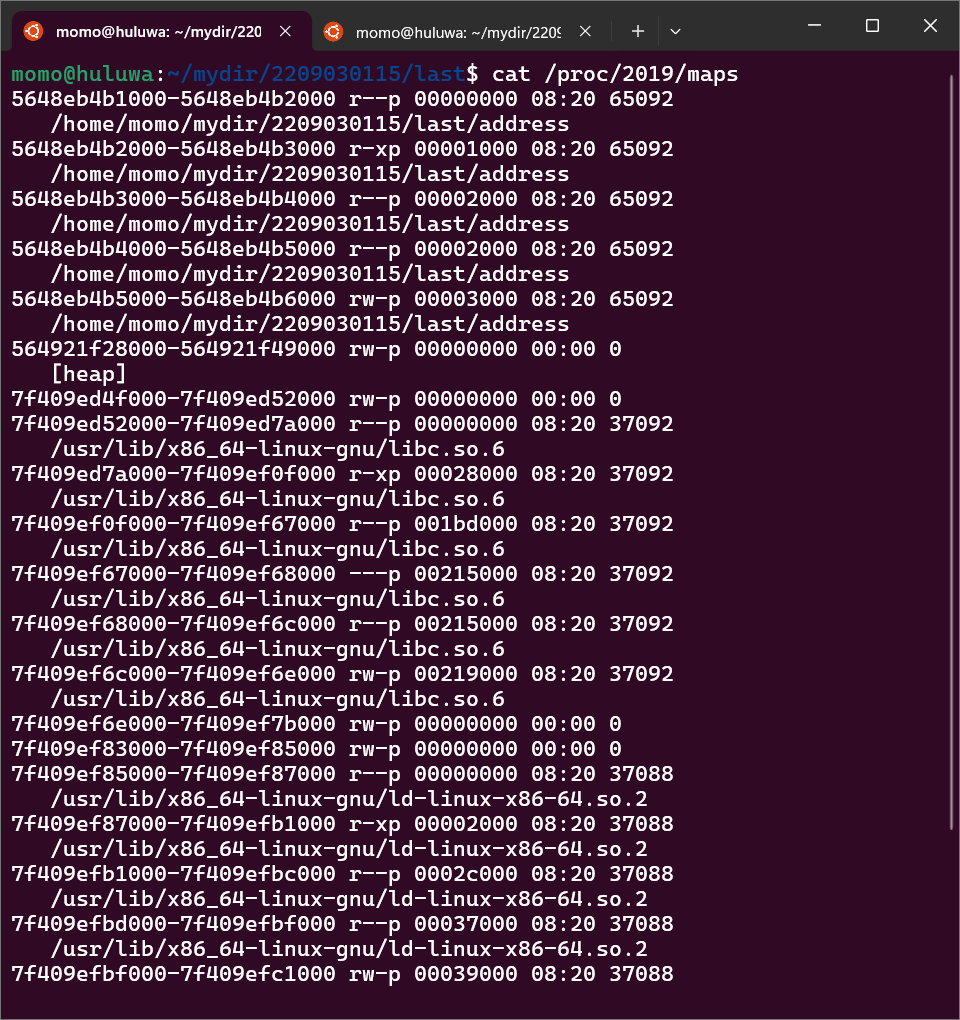
一共有十三个节，.text. data .bss .rodata需要分配存储空间。

1. 使用readelf -l address 命令查看可执行文件address的ELF文件执行视图，分析文件的节和段的映射关系，哪些段需要加载到内存；

3，4，5，6段需要加载到内存中。



1. 后台或者另一个终端执行该程序，使用命令：cat /proc/该进程id/maps ，查看分析该进程的虚拟地址空间构成；



显示了进程虚拟地址空间的详细信息，包括代码段（映射为只读r-xp），存储程序的代码，数据段（通常映射为（rw-p）），存储已初始化的遍量，堆（通常映射为读写（rw-p）），栈与BSS段均映射为rw-p段。

1. 分析程序中各变量在进程虚拟地址空间中的位置，可以结合命令readelf -S address确定各个节的位置，可画图表示。

数据段.data包含已经初始化的全局变量如：data\_var0和data\_var1。

BSS段。Bss包含未初始化的静态变量，如：bss\_var0，bss\_var1，这些变量在程序启动时会被分配内存。

局部变量stack\_var0，和stack\_var1存储在栈上。

使用malloc动态分配的变量heap\_var0存储在堆上。

要求：上传每一步截图并对结果分析