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：

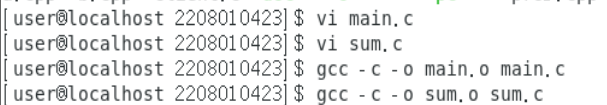
1. 分别用gcc -c sum.c和gcc -c main.c编译汇编两个源程序文件，生成sum.o和main.o可重定位目标文件；
2. 分别使用objdump -d sum.o和objdump -d main.o, readelf -r main.o查看两个文件的反汇编代码，**分析逻辑地址特点以及需要重定位的地址**；
3. 使用gcc -o main sum.o main.o生成可执行目标文件main，使用objdump反汇编，**分析可执行文件中的地址特点，并分析地址重定位如何完成的**；

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

## 问题一实验步骤

### 1）生成可重定位文件



### 2）查看反汇编代码

#### 逻辑地址特点以及需要重定位的地址

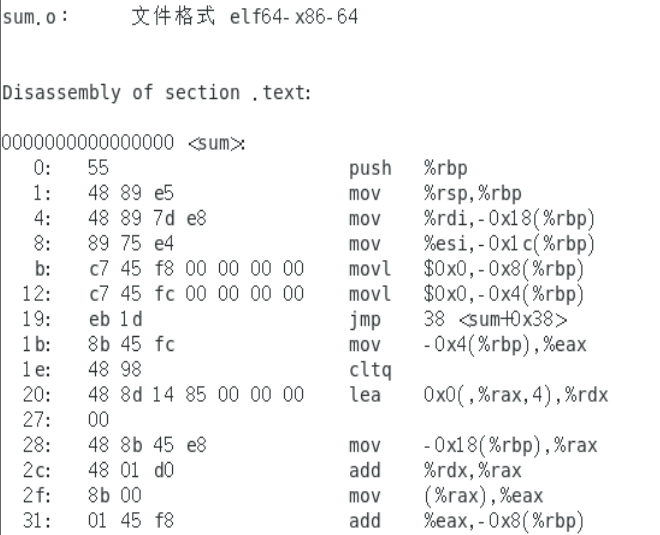
**一、特点 ：目标文件的逻辑地址是相对的，未分配实际内存位置，所有指令地址从 0x0 开始。**

1. **重定位的地址 ：**

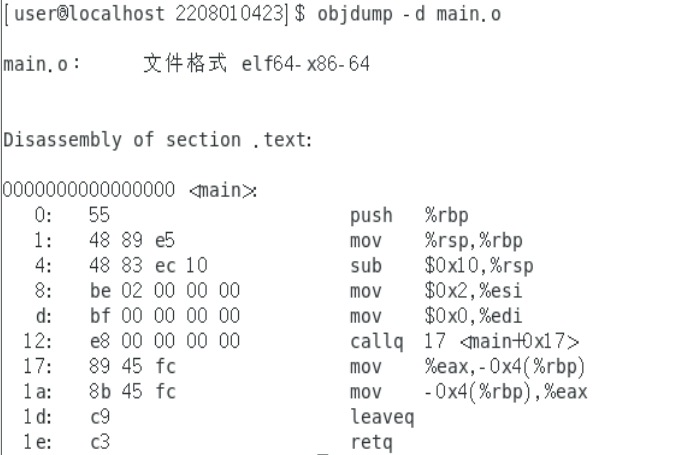
**外部函数调用（sum）：在 main.o 的偏移量 0x13 处，需确定 sum 的入口地址。**

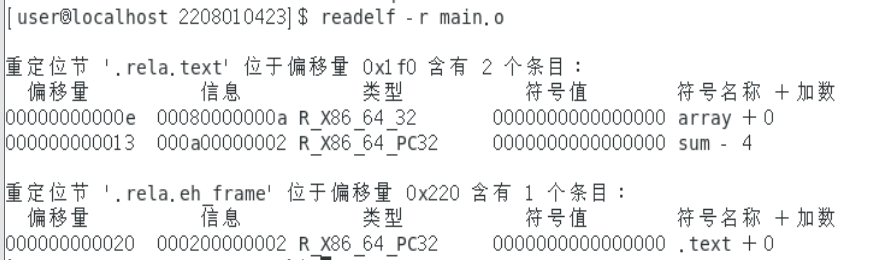
**全局变量引用（array）：在 main.o 的偏移量 0x0e 处，需解析 array 的实际地址。**

#### 截图









### 分析可执行文件地址特点与地址重定位如何完成

1. **地址特点**

**ELF 文件的地址由多个阶段逐步确定：编译阶段生成相对地址，链接阶段完成绝对地址重定位，动态库在运行时解析符号。**

1. **关于ELF文件的分段与分区**

first : 可执行文件按功能分为多个段（Sections），如 .text（代码段）、.data（已初始化的全局变量）、.bss（未初始化的全局变量）。

second : 各段地址以段对齐为单位分配

1. **在内存中加载 ELF 文件时，程序会根据 ELF Header 中的虚拟地址（Virtual Address）将段映射到内存特定位置。**
2. **全局变量与堆栈分布特点**

first : .data 段和 .bss 段分别用于存储初始化和未初始化的全局变量。

second : 局部变量在栈中分配，示例中通过指令如 mov %rsp, %rbp 和 sub $0xN, %rsp 管理。

1. **地址重定位的过程**
2. **符号解析**

链接器将编译时未解析的符号（如函数名 sum 或变量地址）与目标文件或库中的定义匹配。

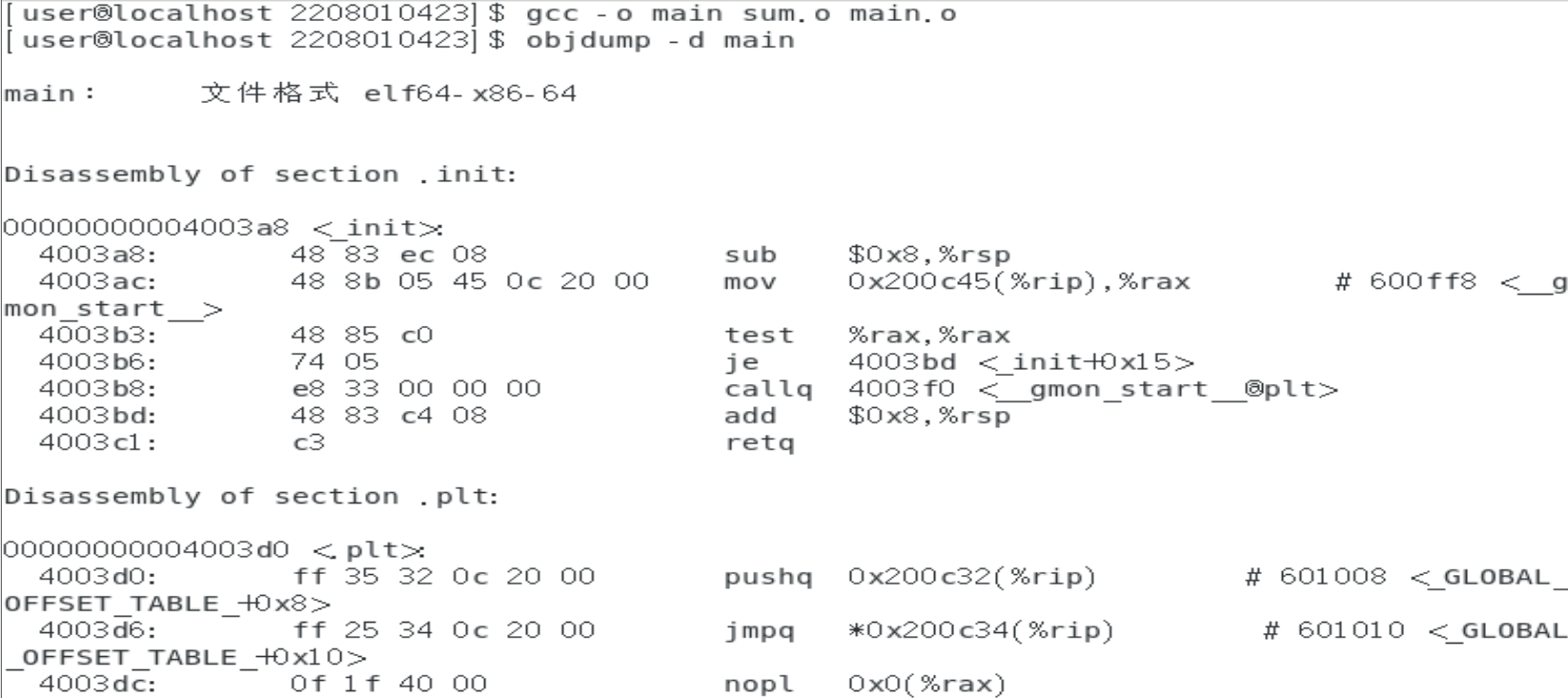
1. **重定位表**

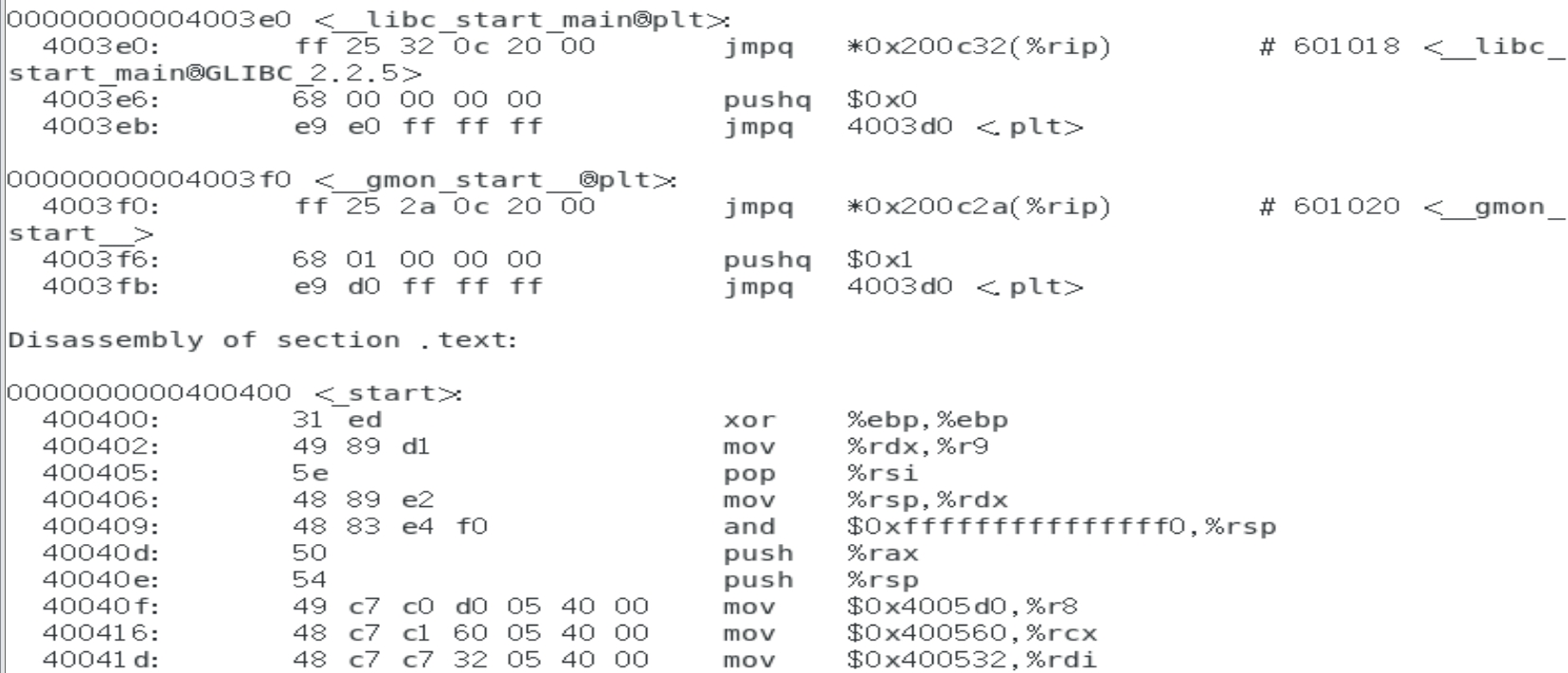
链接器读取重定位表，根据符号的最终地址更新对应位置的指令或数据。

1. **动态重定位**

动态库的符号在运行时完成绑定，动态链接器根据 .got 和 .plt 表逐步解析符号地址。

**（以下是部分截图）**





### 总结

1. **汇编后的可重定位目标文件和链接后的可执行目标文件的区别**
2. 地址空间的差异

可重定位目标文件中的地址是相对地址或段内偏移量，无法直接执行。

链接后的可执行目标文件中的地址是绝对地址或已经调整好的可执行地址。

1. 符号表的处理

可重定位目标文件中包含未解析的符号和需要重定位的符号表。

链接后的可执行目标文件中符号已经解析，符号表用于调试信息等非执行目的。

1. 重定位信息的变化

可重定位目标文件中包含重定位信息，用于标记需要修正的地址。

链接后的可执行目标文件中重定位信息已经消失，地址修正已完成。

1. 段的整合

可重定位目标文件中不同模块的段（如代码段、数据段）是独立的。

链接后的可执行目标文件将各模块的段整合为一个整体布局。

1. **链接器完成的任务**
2. 解析并链接符号，确定函数和变量的最终地址。
3. 结合重定位信息，修正目标文件中的地址。
4. 合并代码段、数据段和其他段，形成可执行文件的内存布局。
5. 更新符号表和调试信息以供后续使用。

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;

}

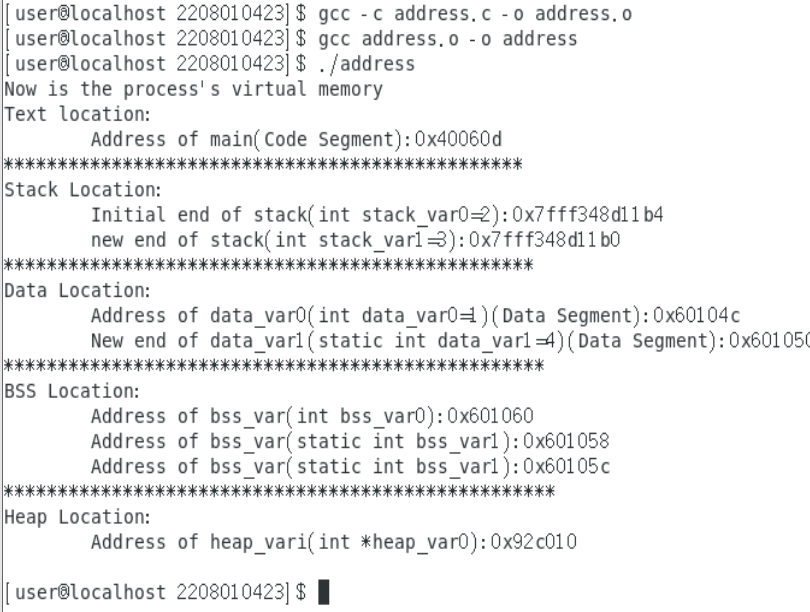
实验步骤：

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；
3. 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；
4. 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；
5. 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。

要求：上传每一步截图并对结果分析

## 问题二实验步骤

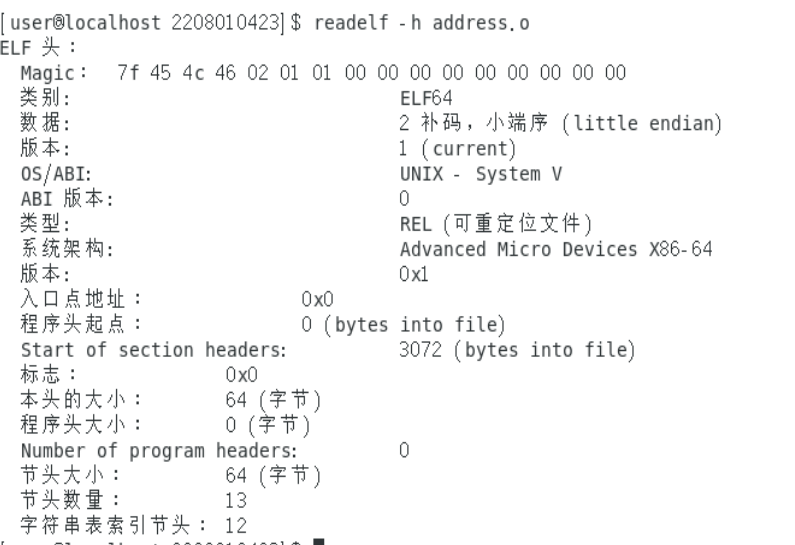
### 编译链接



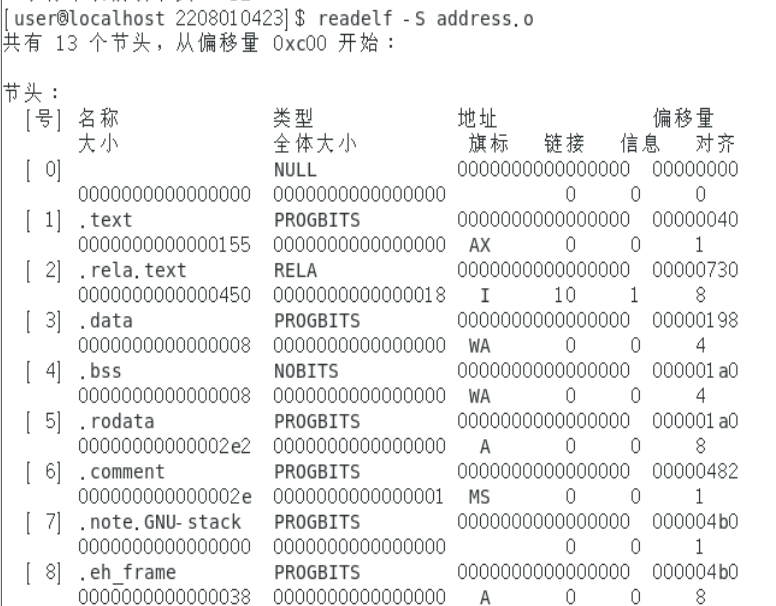
### readelf -h & -s

**【答】 ： 共有 13 个节，其中 5 个节需要需要分配存储空间**

**readelf -h address.o**

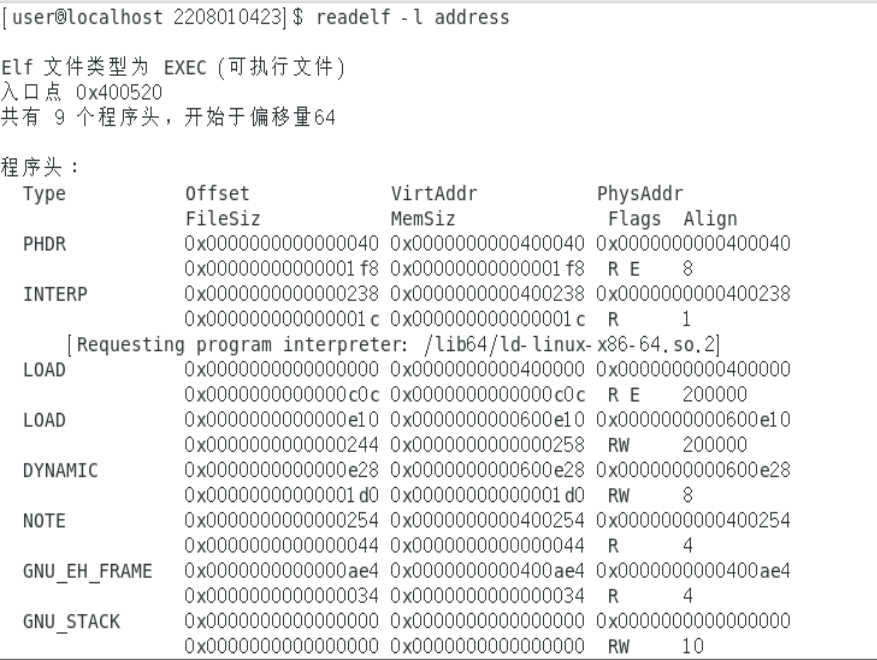


**readelf -S address.o**



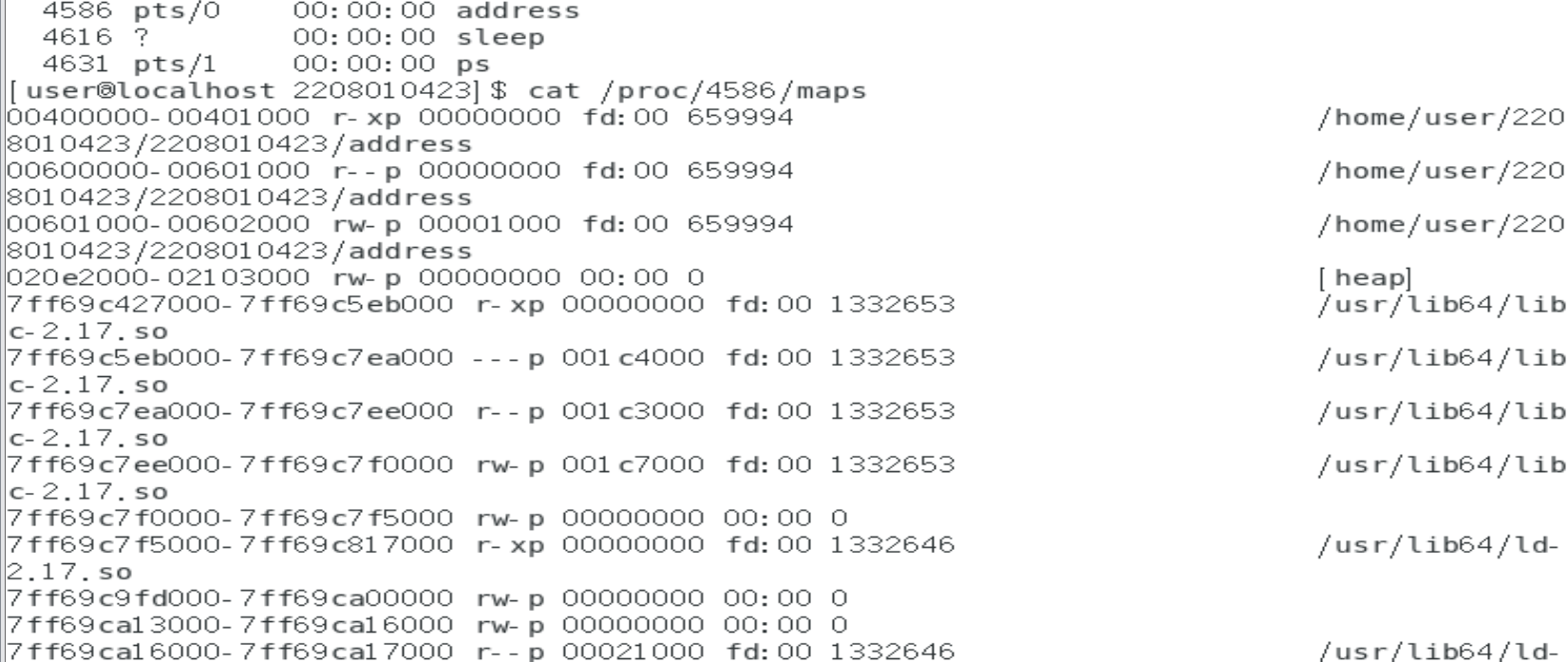
### readelf -l

**段 3 和 段 4 （即 .text 和 .data） 需要加载到内存。**



### 虚拟地址空间构成

1. **代码段（.text）**，只读并可执行
2. **数据段（.data）**，可读可写
3. **堆**，可读可写
4. **栈**，可读可写
5. **共享库（如 libc 和 ld）**
6. **VDSO 和 Vsyscall，用于系统调用和操作系统内部机制**



### 各变量在进程虚拟地址空间中的位置

