**实验8 Linux进程地址空间**

实验指导

1. **程序的编译、汇编和链接**

C语言源程序生成可执行程序需要经过预处理、编译、汇编、链接四个步骤，其中前三个步骤对每一个源程序（.c文件）生成目标文件（也叫可重定位目标文件，gcc生成.o文件），链接是将多个可重定位的.o文件（包括库函数目标模块）生成一个可执行目标文件。这里只讲汇编和链接过程。

1. **汇编**

使用命令“gcc -c”可以将源程序完成预处理、编译、汇编过程，生成.o目标文件，即可重定位目标文件。可以用反汇编的命令objdump来反汇编查看文件内容。

例：有两个C语言源程序，main.c和add.c

main.c：

#include<stdio.h>

int add(int i,int j);

int main()

{ return add(20,13);

}

add.c：

int add(int i,int j)

{ int x = i+j;

return x;

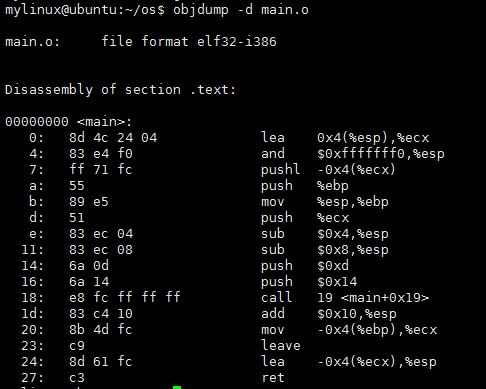
}

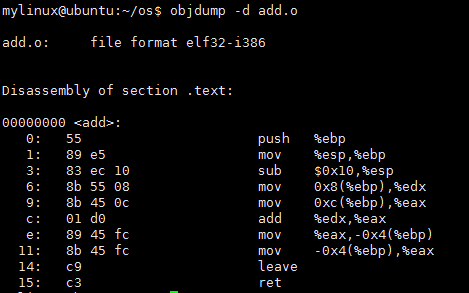
分别用“gcc -c”命令编译生成可重定位目标文件.o文件

$gcc -c -o main.o main.c

$gcc -c -o add.o add.c

然后使用反汇编命令objdump -d查看文件内容，包括：指令的地址、指令的二进制代码和反汇编的汇编代码。Linux中支持的目标程序文件格式为ELF格式。





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中的函数调用指令：call 19，对应的源程序中“return add(20,13);”函数调用语句，其中的0x19表示调用的函数起始位置在偏移量是0x19开始的四个字节记录，即“fc ff ff ff”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. **链接**

使用命令“gcc -o”可以将多个可重定位文件链接起来，生成一个可执行文件，可以用反汇编的命令objdump来反汇编查看文件内容。链接过程中链接器主要完成符号解析和地址重定位两个任务。符号解析是将可重定位目标文件中引用的符号（全局变量名和函数名）与确定的符号定义建立关联。地址重定位是将汇编后的.o文件中的逻辑地址，在完成多个.o文件合并后，按照Linux虚拟地址空间划分重新确定位置。

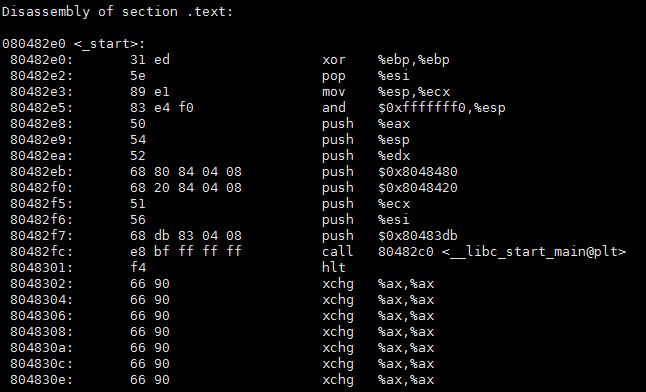
例：将以上两个main.o和add.o链接成可执行目标文件add

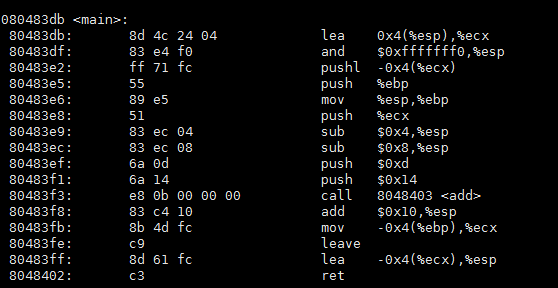
$gcc -o add main.o add.o

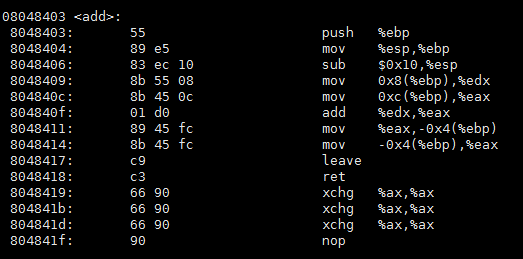
反汇编后查看文件内容：

$objdump -d add

链接后生成很多section，称为节，看一下代码对应的.text节中的main和add部分，可以看到左侧的逻辑地址都变成了统一的相对地址编址，.text节的起始地址为080482e0开始连续编址的地址空间。







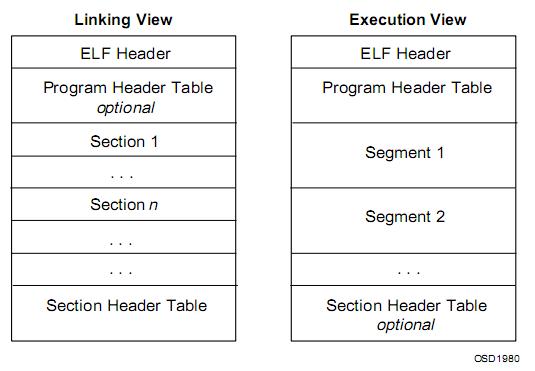
main中的函数调用指令：

80483f3: e8 0b 00 00 00 call 8048403 <add>

call的调用位置经过链接重定位后变成了：8048403，对应的就是add的起始地址。这就是链接过程经过对add符号进行符号解析，然后进行地址重定位得到的：将main.o和add.o的节合并后计算出add的起始地址和main中call指令的偏移量，写到e8 后的四个字节：0b 00 00 00，对应的是从后一条指令：80483f8: 83 c4 10地址80483f8偏移0b个字节，即8048403位置（add的起始位置）。

1. **ELF文件**

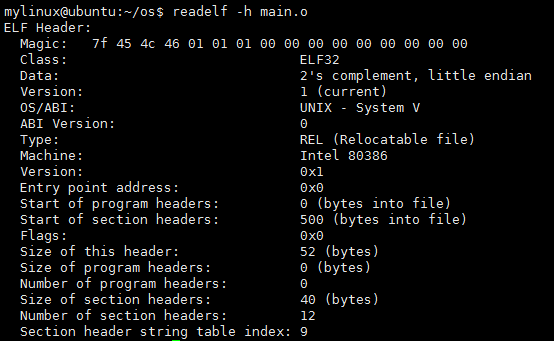
经过预处理、编译、汇编之后的得到的可重定位目标文件，对应生成的ELF文件是链接视图，经过链接之后生成的可执行目标文件，对应生成ELF文件的执行视图，如下图所示。



1. **链接视图**

链接视图由ELF头、多个不同的节和节头表组成，主要的节有：.text节（代码节）、.rodata节（只读数据节，如printf格式串）、.data节（初始化的数据）、.bss节（未初始化的数据）、.symtab节（函数名和全局变量等符号相关信息）等。

**ELF头**：位于目标文件开头，可以使用命令：“readelf -h main.o”对main.o文件的ELF头解析。



主要含义：

Data：说明本文件是采用二进制补码，小端方式

Type：可重定位文件

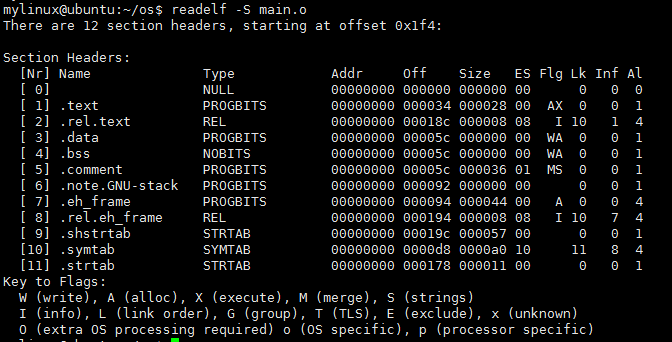
Entry point address:起始地址为0x0

Start of section headers:节头表离文件开始偏移量为500字节

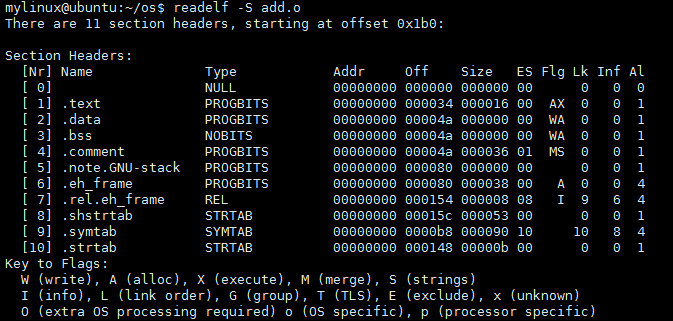
Size of section headers: 节头表中一个表项大小为40字节

Number of section headers:节头表中有12个表项，即12个节

**节头表**：表中每一项对应一个节，可以使用命令：readelf -S main.o对ELF节头表解析。如man.o可重定位文件的节头表：



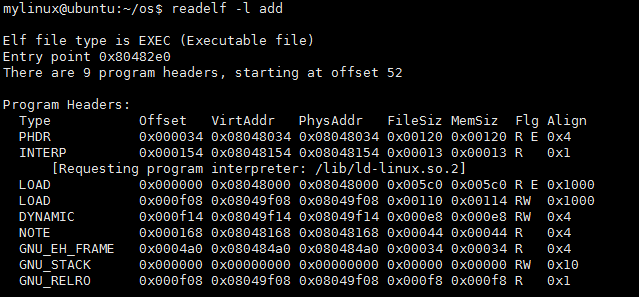
main.o文件中有12个节，节头表从0x1f4字节开始，flg标志中，A表示需要分配存储空间，W表示写，X表示执行，所以.text节、.data节、.bss节需要存储器分配空间，根据每个节在文件的偏移地址和长度，可以确定可重定位文件的结构。

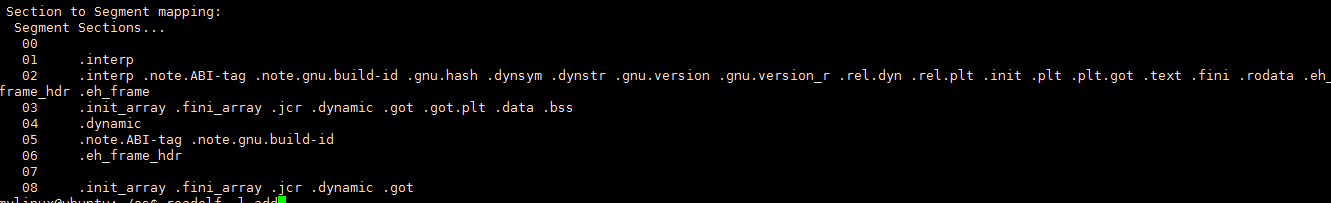


1. **执行视图**

链接器将多个.o可重定位目标文件中相同的代码和数据节合并，形成可执行目标文件中的节，得到ELF执行视图。执行视图可以看做是由不同的段segment组成，描述不同的节如何映射到虚拟存储空间的段中，可以多个节映射一个段。执行视图中多了一个程序头表，描述段或者节到虚拟空间的映射关系。

使用命令：readelf -l add显示可执行目标文件add的程序头表信息，从左到右依次是：存储段类型，本段首字节在文件的偏移地址，本段首字节虚拟地址，本段首字节物理地址（由操作系统决定，内容无效），本段在文件中所占字节数，本段在存储器中所占字节数，存取权限，对齐方式。





其中有两个可装入段（Type=LOAD）对应的表项信息，一个是02只读代码段，包括.text 、.fini 、.rodata、 . init等节，一个是03可读可写数据段，包括 .data 、.bss等节，其中.bss节在磁盘文件中不占空间，在存储器中需要分配空间。

**二、Linux进程地址空间**

进程地址空间（process address space），就是从进程的视角看到的地址空间，是进程运行时所用到的虚拟地址的集合。

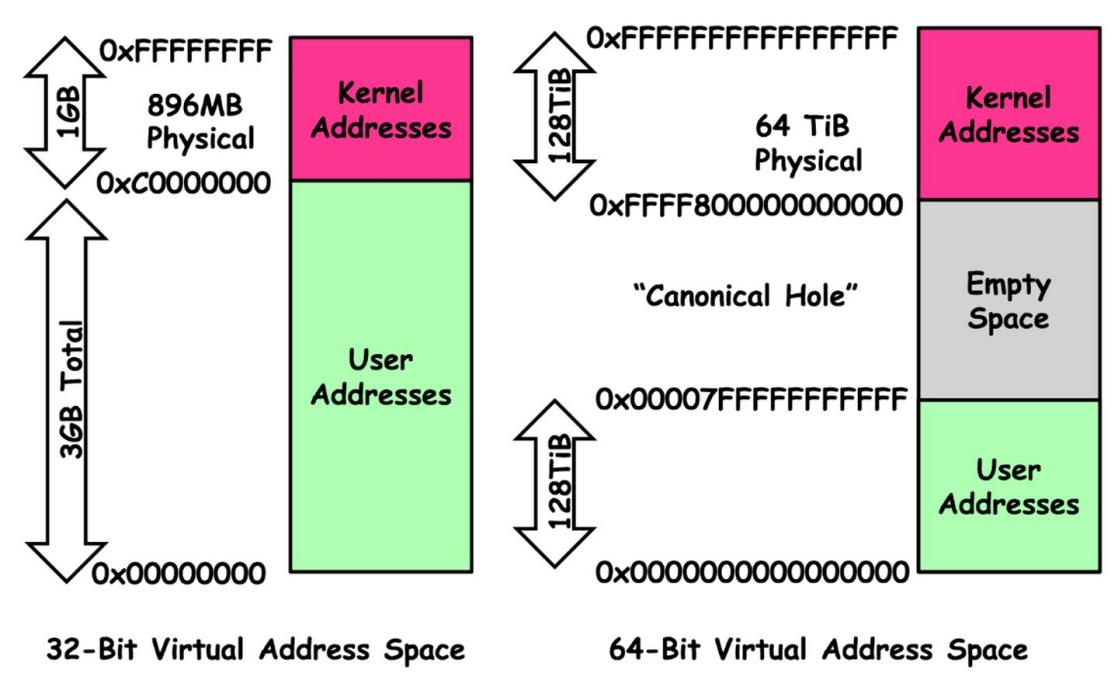
**1.32位系统的进程地址空间**

以IA-32处理器为例，其虚拟地址为32位，因此其虚拟地址空间的范围为232=4GB，Linux系统将地址空间按3:1比例划分，其中用户空间（user space）占3GB，内核空间（kernel space）占1GB。

**2.64位系统的进程地址空间**

在64位系统中，进程地址空间的大小就不固定了，以ARMv8-A为例，它的page大小可以是4KB, 16KB或者64KB（默认为4KB，选一种来用，不要混用），可采用3级页表或4级页表，因此可以有多种组合的形式。

以采用4KB的页，4级页表，虚拟地址为48位的系统为例（从ARMv8.2架构开始，支持虚拟地址和物理地址的大小最多为52位），其虚拟地址空间的范围为248=256TB  ，按照1:1的比例划分，内核空间和用户空间各占128TB。

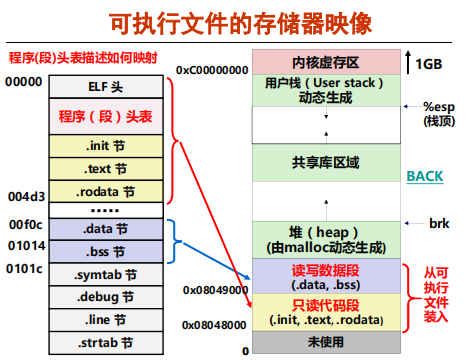


**3.地址空间构成**

可执行目标文件到虚拟地址空间的映像遵循ABI规范，只读代码段总是映射到虚拟地址为0x8048000开始的一段区域，可读可写数据段映射到只读代码段后按4KB对齐的高地址部分，.bss节所在存储区在运行时初始化为0，堆在可读写数据段后面按4KB对齐的高地址部分，调用malloc动态向高地址分配空间；栈空间从用户空间的最大地址往低地址方向增长；堆区和栈区中间有一块保留给共享库目标代码。

运行可执行目标文件时，操作系统的加载器负责代码段、数据段和BSS段的加载，并在内存中为这些段分配空间。栈也由操作系统分配和管理；堆由程序员自己管理，即显式地申请和释放空间。

可执行目标文件的ELF各部分和虚拟地址空间的对应关系如图：



进程虚拟地址空间各部分内容如下表所示(按地址递减顺序)：

|  |  |
| --- | --- |
| **名称** | **存储内容** |
| 用户栈 | 局部变量、函数参数、返回地址等 |
| 共享库 | 动态链接库、内存映射 |
| 堆 | 动态分配的内存 |
| BSS段 | 未初始化或初值为0的全局变量和静态局部变量 |
| 数据段 | 已初始化且初值非0的全局变量和静态局部变量 |
| 代码段 | 可执行代码、字符串字面值、只读变量 |

**（1）用户栈**

由编译器自动分配释放，主要有三个用途：

为函数内部声明的非静态局部变量提供存储空间。

记录函数调用过程相关的维护性信息，称为栈帧(Stack Frame)，包括函数返回地址，不适合装入寄存器的函数参数及一些寄存器值的保存。

临时存储区，用于暂存长算术表达式部分计算结果或alloca()函数分配的栈内内存。

**（2）共享库**

该区域用于映射可执行文件用到的动态链接库。若可执行文件依赖共享库，则系统会为这些动态库在从0x40000000开始的地址分配相应空间，并在程序装载时将其载入到该空间。在Linux 2.6内核中，共享库的起始地址被往上移动至更靠近栈区的位置。

**（3）堆**

堆用于存放进程运行时动态分配的内存段，可动态扩张或缩减。堆中内容是匿名的，不能按名字直接访问，只能通过指针间接访问。当进程调用malloc(C)/new(C++)等函数分配内存时，新分配的内存动态添加到堆上(扩张)；当调用free(C)/delete(C++)等函数释放内存时，被释放的内存从堆中剔除(缩减) 。

（4）**BSS和data段**

进程的可读写数据段包括两部分：.bss节和.data节。

.bss节中通常存放程序中以下符号：

* 未初始化的全局变量和静态局部变量
* 初始值为0的全局变量和静态局部变量

BSS段仅为未初始化的静态分配变量预留位置，在目标文件中并不占据空间，这样可减少目标文件体积，但程序运行时需为变量分配内存空间。

data段通常用于存放程序中已初始化且初值不为0的全局变量和静态局部变量。属于静态内存分配(静态存储区)，可读可写。数据段保存在目标文件中，其内容由程序初始化。例如，对于全局变量int gVar = 10，必须在目标文件数据段中保存10这个数据，然后在程序加载时复制到相应的内存。

 BSS段不占用物理文件尺寸，但占用内存空间；data段占用物理文件，也占用内存空间。

运行时data段和BSS段的整个区段通常称为数据区

**（5）代码段**

代码段也称正文段、text段，通常用于存放程序执行代码(即CPU执行的机器指令)。一般C语言执行语句都编译成机器代码保存在代码段。通常代码段是可共享的，因此频繁执行的程序只需要在内存中拥有一份拷贝即可。

**4.地址空间查看**

可以通过用/proc文件系统和 pmap工具查看进程的虚拟内存空间及所包含的内存区域。

1. **查看proc文件**

例如：运行add程序，使用cat /proc/进程id/maps查看进程虚拟地址空间

输出的信息依次为：

第一列：本段在虚拟内存中的地址范围。

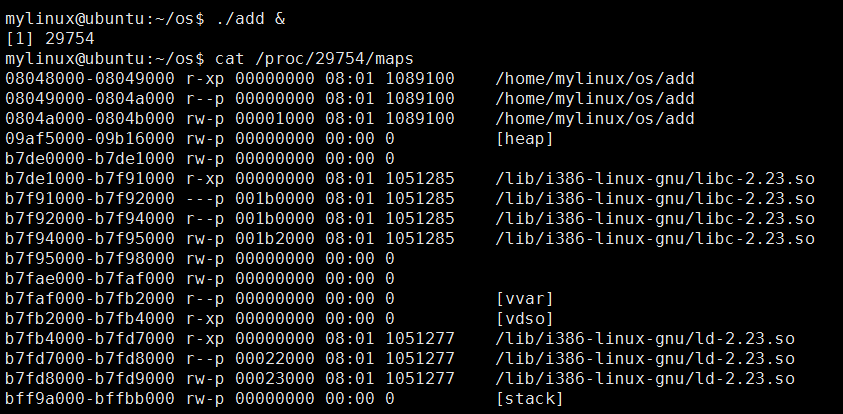
第二列：本段的权限。

第三列：偏移地址，即指本段映射地址在文件中的偏移。

第四列：主设备号与次设备号。

第五列：文件索引节点号。

第六列：映射的文件名。



第一行对应的是可读可执行的代码段，包括.init、.fini、.rodata、.text节

第二行对应的是只读数据段，包括.init\_array .fini\_array节

第三行对应的是可读可写的数据段，包括 .data .bss节

第四行对应的是堆段

中间多行对应共享库存储空间

最后一行对应栈空间

1. **pmap工具**

使用pmap工具查看，每一行输出表示：

Address: 内存开始地址

Kbytes: 占用内存的字节数(KB)

RSS: 驻留内存的字节数(KB)

Offset: 文件偏移

Device: 设备名 (major:minor)

Mapping: 占用内存的文件、或[anon](分配的内存)、或[stack](堆栈)

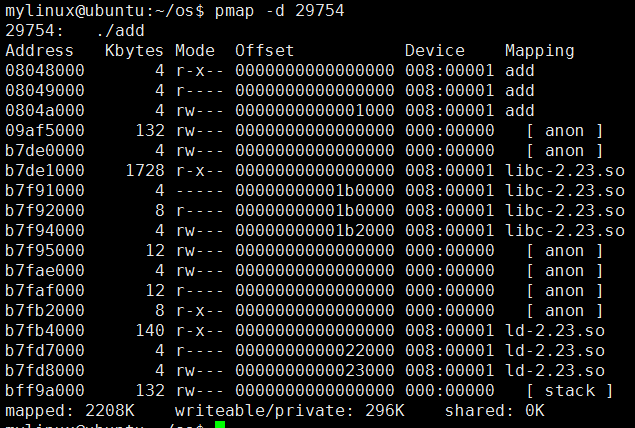
最后一行的值：

mapped ：表示该进程映射的虚拟地址空间大小，也就是该进程预先分配的虚拟内存大小。

writeable/private ： 表示进程所占用的私有地址空间大小，也就是该进程实际使用的内存大小。

shared： 表示进程和其他进程共享的内存大小。

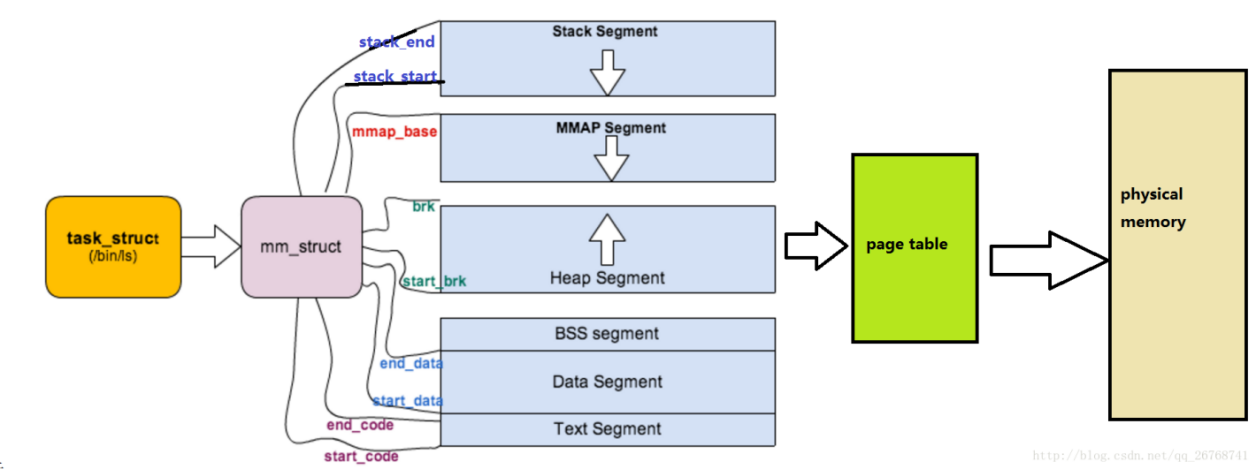
注意：共享库只在内存中有一份，是所有进程共享的，这里该进程实际占用空间大小为writeable/private占用的空间大小。



**三、Linux内存管理数据结构**

Linux内核的进程控制块是task\_struct结构体，进程的虚拟地址空间记录在task\_struct结构的指针成员mm中(struct mm\_struct)，里面包含pgd、代码段、堆、栈地址等信息。

mm->pgd指向进程的页表内存位置，进程调度时，页表被切换，一般会有一个CPU寄存器来保存页表的地址，比如X86下的CR3，页表切换就是改变该寄存器的值。



mm有一个名为mmap的指针，它指向一个链表，链表每一个元素都是vm\_area\_struct结构，称为VMA。这个链表的每一段（每个元素）都是这个进程的一个虚拟地址空间。

一个进程的地址空间都是一段一段的，通过pmap pid命令、/proc/pid/maps、/proc/pid/smaps都能看出一个进程的虚拟地址空间分布情况。