第 5 章
虚拟与物理的内存

我们在第一部分花了很多时间介绍冯·诺依曼体系结构，讨论了一个计算机系统要怎样执行一条程序指令：我们需要内存存储**程序指令代码**、**运行栈**，需要**程序计数器**保存指令的地址，需要各种各样的**寄存器**来存储必要的数据，需要ALU进行算术逻辑运算。

在前 4 章，电子科学家向计算机科学家传授了很多知识。在这一章，计算机科学家要反过来向电子科学家教授知识了。我们从虚拟内存开始，讨论怎样同时运行多道程序。

5.1 从源代码到进程

电子科学家：“计算机科学家啊，我还有很多知识可以告诉你，不过那些知识是具体的电路，你应该不会感兴趣。”

计算机科学家：“没错，那些太底层了。现在，我们开始考虑怎样同时运行多道程序。这里面最突出的一个问题是怎样去管理内存，也就是虚拟内存系统。”

电子科学家：“看来你想要教导我了。不过到目前为止，我们依然不了解整个程序运行的全貌。为了更清楚地理解虚拟内存，我们有必要先大概了解 C 语言程序源代码要经过哪些步骤，最后才能在计算机系统中运行。”

计算机科学家：“行，那我们就从二进制的可执行文件开始说吧。”

5.1.1 编译源文件

计算机科学家：“我们从一个最简单的 C 语言程序开始，HelloWorld.c，见代码 5.1。这几乎已经成为一个事实上的标准——每一个初学 C 语言的程序员都会在键盘上敲下这样一个程序。”

代码 5.1 /vmem/HelloWorld.c

```
1 #include <stdio.h>
2 int main(int argc, const char *argv)
3 {
4     printf("Hello World!\n");
5     return 0;
6 }
```

电子科学家：“很好，那么你想向我说明什么?”

计算机科学家：“我想说的是，一个 C 语言程序怎样从文本变成可执行文件，再被加载进内存，然后执行。”

电子科学家：“明白了，那你开始吧。”

计算机科学家继续：“代码 5.1 文件的后缀名虽然是'.c'，但它其实只是一个简单的文本文件而已。我们用 hexedit 工具打开这个文件，看到的也只是一串 ASCII 字符而已（可以按 Ctrl + X 快捷键退出 hexedit）。”

```
>_                                                          Linux Terminal

> hexedit ./HelloWorld.c
00000000   23 69 6E 63  6C 75 64 65  20 3C 73 74  64 69 6F 2E  #include <stdio.
00000010   68 3E 0D 0A  69 6E 74 20  6D 61 69 6E  28 69 6E 74  h>..int main(int
00000020   20 61 72 67  63 2C 20 63  6F 6E 73 74  20 63 68 61   argc, const cha
00000030   72 20 2A 61  72 67 76 29  0D 0A 7B 0D  0A 20 20 20  r *argv)..{..
00000040   20 70 72 69  6E 74 66 28  22 48 65 6C  6C 6F 20 57   printf("Hello W
00000050   6F 72 6C 64  21 5C 6E 22  29 3B 0D 0A  20 20 20 20  orld!\n");..
00000060   72 65 74 75  72 6E 20 30  3B 0D 0A 7D               return 0;..}
```

他继续：“通过 gcc 编译器，我们可以把这个纯文本的源文件变成一个二进制文件。当然，不同的编译器编译的结果在细节上可能有所不同，但总体上是差不多的。我们用 gcc 进行编译，同时与 Linux 上的 C 语言标准库进行链接。”

```
>_                                                          Linux Terminal

> gcc ./HelloWorld.c -o Hello
```

计算机科学家：“这样生成的二进制文件被称为**可执行与可链接格式**（Executable and Linkable Format，ELF）文件，简称ELF文件。”

电子科学家：“所以 ELF 文件在本质上已经与 C 语言的源文件不同了，对吗？它包含程序指令，CPU 就是读取这些指令，然后在冯 • 诺依曼机器上执行的。”

计算机科学家：“没错。如果我们用 32 位 RISC-V 的编译器进行编译，那么 Hello 就包含 RSIC-V 指令。如果在普通的 64 位家用计算机上编译，通常是 x86-64 的计算机，所得到的就是 x86-64 的指令。在这里，我仍然用 RSIC-V 架构来演示。”

他继续说：“我们可以用 readelf 或 objdump 工具来检查 ELF 文件的内容。如图 5.1 所示，整个程序文件其实由若干个**段**（Segment）或**节**（Section）构成。在考虑**链接**时，我们一般称‘节’，这时 ELF 文件尚未经过链接，尚不可以直接执行。在考虑**执行**时，我们一般称‘段’，此时 ELF 文件已经可以被加载到内存中执行了。我们这里主要讨论链接以后，因此称‘段’。”

电子科学家：“这只是名称上的不同?”

计算机科学家：“其实有一些细微的差别，但我觉得你可以忽略掉，就当作‘只是名称上的不同’就好。”

在文件开头，存放的是ELF **文件首部**（ELF Header），其中记录了一些元数据，包括**魔数**（Magic Number），用来表示文件类型；二进制的编码格式，大/小端机；操作系统；可执行的计算机体系结构等。

计算机科学家：“其中，对我们而言，最重要的是 3 个字段：(1) 整个程序第一条指令的位置，

即**入口地址**（Entry Point Address），为虚拟地址 `0x100dc`；（2）程序头表（Program Header），用来记录加载到内存中的段信息；（3）节头表（Section Headers），用来记录其他节/段的位置。”

电子科学家：“那是怎么找到这些表的呢？要知道，计算机拿到这个可执行文件，其实能知道的唯一信息就是它的大小，以及它在磁盘上的起始地址。”

计算机科学：“没错，这就是 ELF Header 的作用。计算机可以比较 ELF Header 中的特殊字段，得知这是一个‘可执行文件’。然后就根据一张张‘表’去找到另一张张‘表’。(2)、(3) 两个表是通过它们的第一个字节相对文件开头的**偏置**（Offset）找到的。例如程序头表从整个文件的第 `52` 个字节开始，节头表从第 `23 380` 个字节开始。”

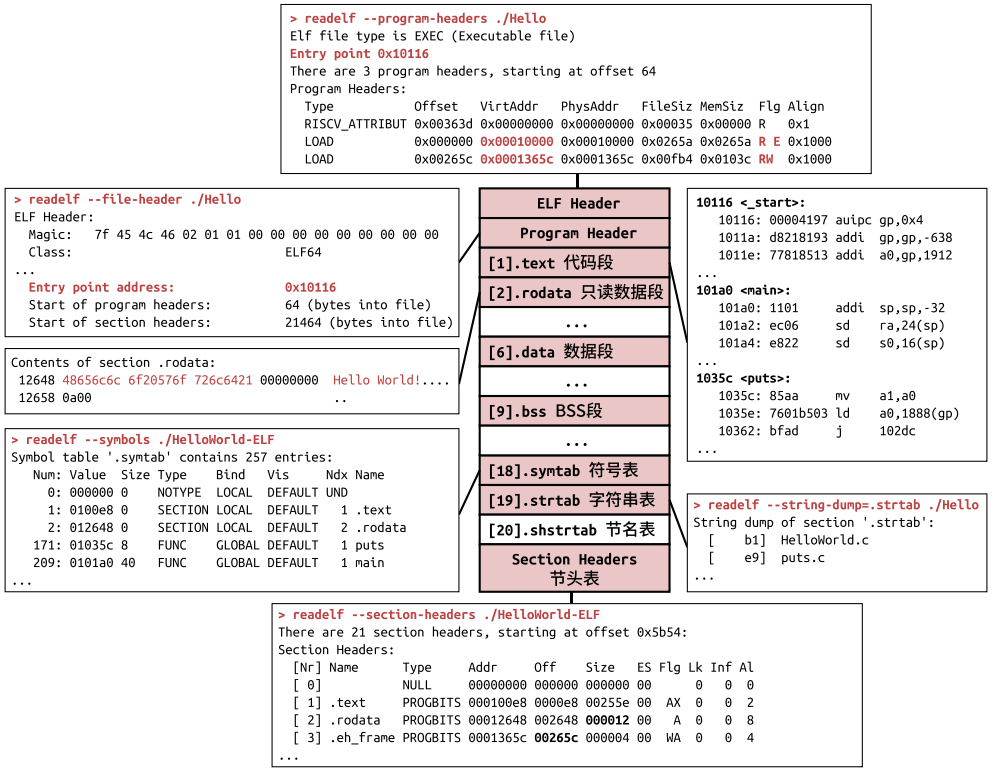

图 5.1　HelloWorld 程序 ELF 文件结构

5.1.2　链接 ELF 文件

计算机科学家：“**链接**（Link）分为两种，**静态链接**（Static Link）和**动态链接**（Dynamic Link），我主要介绍一下静态链接。”

他继续分析：“比如代码 5.1 中的 `printf()` 函数，其实是实现在其他文件中的。当编译代码 5.1 时，如果不进行链接（`gcc` 的`-c` 选项），我们将得到一个不含 `printf()` 函数指令的 ELF 文件。”

电子科学家问到：“哦？也就是说，`printf()` 函数的指令不在 ELF 文件中？”

计算机科学家打了一个响指：“没错！ELF 文件里其实没有 `printf()` 函数的指令。”

电子科学家：“那程序要怎么编译出来呢？之前大精灵介绍过的，过程调用需要把 `printf()` 函数的地址写成立即数，保存在 `jalr` 指令中。但如果没有 `printf()` 函数的指令，ELF 文件

中的 jalr 指令要怎么设置呢?”

计算机科学家：“你偷听我和大精灵的对话！算了，无所谓了。链接的策略是这样的，对于所有调用 printf() 函数的地方，用一个临时的指针替代，就像是一个**占位符**（Place Holder）。”

电子科学家：“那也就是说，有其他的 ELF 文件中包含了 printf() 的指令。等找到这个 ELF 文件时，才能知道 printf() 函数的真正指令地址?”

计算机科学家：“是的，到那时再将临时替代的指针换成函数指令的真正地址，完成静态链接。在这个过程中，最重要的就是**节头表**、**符号表**（Symbol Table）和**重定位表**（Relocation Table）。在这里，我不介绍重定位表，只介绍一下节头表和符号表。”

如图 5.1 所示，节头表主要用来描述每一个节/段的位置，以及它们的属性。如.text 段，它所在的文件位置是偏置 0x94，大小是 0x35dc 字节。有了节头表，就可以定位到可执行文件中每一个节/段。在这里，我们主要关心的节/段有。

（1）.text 节/段：用来存放指令代码，如 main() 函数、printf() 函数，它们的指令都存放在这个区域；

（2）.rodata 节/段：用来存放只读数据。如代码中的字符串"Hello World!"，这其实是一段不会被修改的字符串（字符串常量），因此被保存在这一区域；

（3）.data 节/段：用来存放可读可写，并且已经被初始化的全局数据；

（4）.bss 节/段：全称为Block Started by Symbol，用来存放未初始化的全局数据。一般未初始化的数据都填写为 0，所以就没必要专门储存 0 了，只要记住有哪些变量未初始化就好；

（5）.symtab 节：即**符号表**，用来记录文件、函数、全局变量等信息。例如，主函数 main() 就是一个 FUNC 类型的符号，作为全局可见的符号被记录；

（6）.strtab 节：即**字符串表**，符号表中对每一个符号会记录一个字符串表中的位置，用来表示该符号对应的名称。例如，main() 函数的符号会在字符串表中记录"main" 字符串的位置，可以用来调试等。

计算机科学家：“在静态链接中，链接器 ld 根据符号表处理各个 ELF 文件之间的符号冲突，按照优先级选择正确的符号，从而各个 ELF 文件的‘节’合并为可执行文件中的‘段’。选择正确的符号后，根据重定位表把临时指向符号的指针修改为正确的符号地址。”

5.1.3 加载可执行文件

计算机科学家：“当我们用 gcc 编译器从源代码文件 HelloWorld.c 编译、链接得到可执行文件 Hello 后，我们便可以将 Hello 加载到内存，并且执行它。”

电子科学家：“啊？就这么简单?”

计算机科学家：“当然不是。实际上，在建立内存与程序文件之间的数据映射前，Linux 还有很多其他准备工作，但这里就不提了，我们忽略这些细节。”

电子科学家：“那你不忽略的是什么?”

计算机科学家：“这些过程主要依赖**程序头表**，程序头表主要记录的是程序要运行时，加载到内存中的信息。如图 5.1 所示，我们主要关心后两个表项。”

一个虚拟地址为 0x00010000，大小是 0x265a，标志位是 RE。这表示从文件起始位置开始，偏置范围为 0x000000（Offset）到 0x000000 + 0x265a = 0x265a（Offset + FileSiz）的

数据，当程序开始运行时，全部被加载到内存中。这部分数据对应的虚拟地址是 0x00010000 到 0x00010000 + 0x265a = 0x0001265a（VirtAddr + MemSiz），并且这部分内存的权限是**可读可执行**（Flg = R E）。因此，这部分数据其实对应的是.text 与.rodata，也就是程序的**指令代码**和**只读数据**。

另一个虚拟地址为 0x0001365c，大小是 0x103c，标志位是 RW。这表示从文件起始位置开始，偏置范围为 0x00265c（Offset）到 0x00265c + 0xfb4 = 3610（Offset + FileSiz）的数据，当程序开始运行时，全部被加载到内存中。这部分数据对应的虚拟地址是 0x0001365c 到 0x0001365c + 0x103c = 0x00014698（VirtAddr + MemSiz），并且这部分内存的权限是**可读可写**（Flg = RW）。因此，这部分数据其实对应的是.data 与.bss 等，也就是程序的**已初始化的数据**和**未初始化的数据**。

计算机科学家提醒：“注意，由于.bss 段加载到内存以后是要分配值为 0 的内存的，所以 FileSiz 与 MemSiz 不一致。上述过程如图 5.2 所示。”

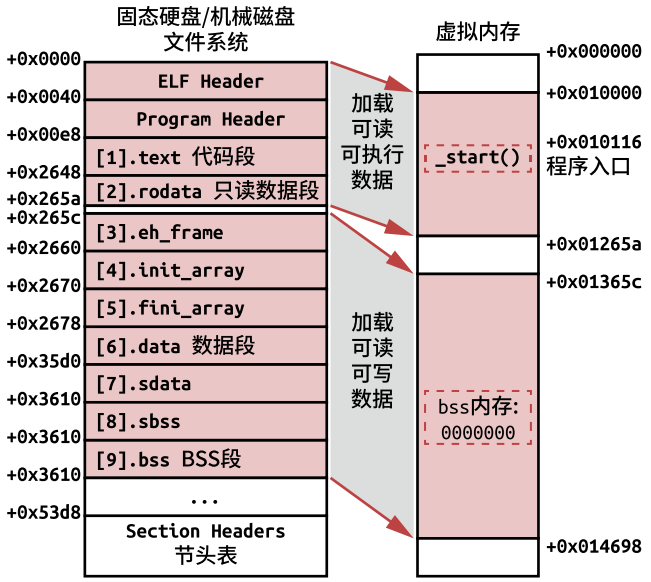

图 5.2　Program Header 所描述的加载过程

计算机科学：“在这其中，符号表、字符串表、节头表都不会被加载到内存中，而.text、.rodata、.data、.bss 段则会被加载进内存，用于程序的执行。”

电子科学家：“真的要把数据从文件中拷贝到内存吗？这实在是太浪费时间了。我感觉我平时执行程序没有这么慢的。”

计算机科学家：“是的，这是一个可以优化的地方。举一个例子，假如.rodata 段中有一个字符串，要在程序运行很久以后才会被访问。这样一来，就没必要在程序刚加载的时候便拷贝数据。”

电子科学家：“没错，这样合理很多。”

计算机科学家：“Linux 采用的就是这样的‘懒惰’策略——只使用 mmap() 系统调用先建立映射，将虚拟内存 [0x010000 : 0x01265a)（左闭右开区间）映射为程序文件的 [0x0000 :

`0x265a)`，虚拟内存 `[0x01365c : 0x014698)` 映射为程序文件的 `[0x265c : 0x3610)`，但是不拷贝数据；只有当程序真正访问这一段虚拟内存时，才从硬盘上拷贝数据进内存。这种策略也被称为 **‘按需加载’**（Demand Loading），之后讲**页表映射**时我们还会讨论。”

电子科学家：“那建立虚拟内存与程序文件的地址映射后，再将程序计数器 `pc` 的值设置为虚拟地址 `0x10116`，那么 CPU 就会从程序的第一条指令开始执行整个程序了。”

计算机科学家：“是的。一个程序被 CPU 执行指令，访问内存，我们称这样处于**运行时**（run time）的程序为一个**进程**（process）。一个进程是程序在‘运行时’的抽象，包含运行时所需要的一切资源——程序计数器、寄存器、ALU、物理内存上的`.text` 段、`.rodata` 段、`.data` 段、`.bss` 段、堆、栈等。”

计算机科学家兴奋道：“进程是整个计算机科学中最深刻的抽象之一，有了‘进程’的概念，程序员就能够管理计算资源。进程拥有实实在在的数据结构，具体参见第 6 章。”

电子科学家：“行，我等着第 6 章再和你讨论这个话题。”

5.2 进程的虚拟地址空间

计算机科学家：“在第 3.3 节的图 3.4 中，我和数学家（主要是我指导数学家）一致认为，程序在运行时需要一个栈，用来存放局部变量、函数调用关系等。”

电子科学家：“嗯嗯，我相信你。那现在我们也在图 5.2 中看到了`.text` 段、`.data` 段这些数据怎样被映射在内存里。那么，一个程序在运行时，或者说一个**进程**，它的虚拟地址上有哪些数据呢？这些数据又来自哪里呢?”

计算机科学：“如图 5.3 所示，运行中的进程主要有`.text` 段、`.rodata` 段、`.data` 段、`.bss` 段、堆、内存映射、栈等。”

他继续：“`.text`、`.rodata`、`.data`、`.bss` 这几个段我们已经介绍过了，它们都是从程序文件中加载到内存的数据。其中，`.text` 段、`.rodata` 段是只读数据，可以执行；`.data` 段、`.bss` 段可读可写，不可执行。”

电子科学家：“有点乱，你一一解释一下吧。”

5.2.1 按页（4096）对齐

在程序文件中，可读可执行段及可读可写段是连续的。但加载到内存以后，这两个区域却不再连续了，而它们之间的内存也没有任何作用。这是因为运行中的内存按照 `4096` 字节进行**分页**（Paging），从虚拟地址 `0x0` 开始，每 4096 字节为一个虚拟内存页。而内存的可读、可写、可执行权限是按照内存页的粒度决定的。也就是说，如果某一个内存页不可写，那么这个内存页中的每一个字节都不可写。

电子科学家：“那这么说，‘只读’和‘可读可写’两种数据必须在不同的内存页上?”

计算机科学：“没错。举个例子，如图 5.2 所示，‘只读’在文件中的范围是 `[0x0000 : 0x265a)`，对应的虚拟内存范围是 `[0x010000 : 0x01265a)`。这一片虚拟内存位于第 $\left\lfloor \frac{\texttt{0x010000}}{4096} \right\rfloor = 16$ 页（向下取整）到第 $\left\lfloor \frac{\texttt{0x01265a}}{4096} \right\rfloor = 18$ 页之间——`[16 : 18]`”。

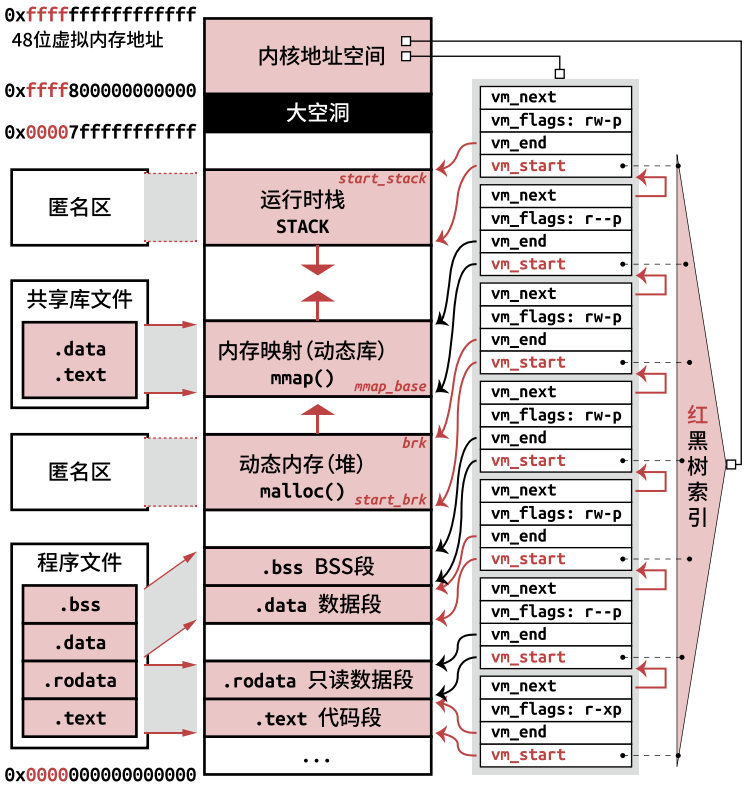

图 5.3　进程的用户态地址空间

电子科学家指出:“而‘可读可写’在程序文件中紧接着‘只读’,但在虚拟地址上却分开了,范围是 `[0x01365c : 0x014698)`,对应第 $\left\lfloor \frac{\texttt{0x01365c}}{4096} \right\rfloor = 19$ 页到第 $\left\lfloor \frac{\texttt{0x014698}}{4096} \right\rfloor = 20$ 页——`[19 : 20]`。”

计算机科学家总结道:“它们的虚拟地址可能不连续,但虚拟页是连续的。”

5.2.2　文件备份与匿名

除了`.text`、`.rodata`、`.data`、`.bss` 这几个区域外,还有一些区域是从文件中加载的,包括**共享目标文件**(Shared Object File)。如 `libc.so`,这种共享的目标文件是通过 `mmap()` 系统调用加载到进程的内存中的,以**动态链接**的方式被进程使用,并且通过页表的机制与其他进程同时使用。

电子科学家:“我懂了,这里是可以优化的。因为 `libc.so` 包含自己的`.text` 只读代码段,所以没必要让每一个进程都加载一份只读数据。所有进程都用同一份只读数据就可以了,这样能够节省内存。”

不论是从程序本身加载到内存的区域,或是通过 `mmap()` 加载到内存并被共享的内存区域,这些内存区域背后都有一个文件支撑。因此它们被称为**文件背景区域**(File-Backed Virtual Memory Area)。

计算机科学家:“一旦这一类内存出现任何问题,直接从文件中再加载一次数据即可。”

与之相对地，堆、栈这两种数据只存在于内存之中，没有任何文件在其背后支撑。因此，这一类内存区域被称为**匿名区域**（Anonymous Virutal Memory Area）。

5.2.3 堆、内存映射、栈

堆、内存映射、栈这些区域可以通过结构体 `vmem_t` 标记，见代码 5.2。结构体的内存被分配在内核空间中。在 Linux 中，起到同样作用的结构体是头文件 `source/include/linux/mm_types.h` 中的 `mm_struct`，意思是“Main Memory Struct”。

代码 5.2 `/vmem/Vmem.h`

```
#ifndef _VMEM_H
#define _VMEM_H
#include <stdint.h>
#include "code/rbt/VMA.h"

typedef struct VMEM_STRUCT
{
    uint64_t pgd;                   // 第一级页表的物理页号

    uint64_t start_code, end_code;  // 代码段的起始、结束虚拟地址
    uint64_t start_data, end_data;  // 数据段的起始、结束虚拟地址
    uint64_t start_brk, brk;        // 动态内存的起始、结束虚拟地址
    uint64_t mmap_base;             // 内存映射的基地址
    uint64_t start_stack;           // 栈的起始地址（高地址）

    uint64_t arg_start, arg_end;    // 程序运行参数的起始、结束地址
    uint64_t env_start, env_end;    // 程序运行环境的起始、结束地址

    vma_t *vm_areas;                // Virtual Memory Area链表
} vmem_t;
#endif
```

堆的起始地址是 `vmem_t.start_brk`，结束地址是 `vmem_t.brk`，这段内存可读、可写。是用户程序通过调用 `malloc()` 函数分配的内存。当空闲内存不够时，`brk` 指针会向高地址增长，为堆分配更多的空间。

内存映射的起始地址是 `vmem_t.mmap_base`。当调用 `mmap()` 函数，要为新的文件建立内存映射时，Linux 会查找可用的虚拟内存，然后向高地址增长，分配空间。

栈的起始地址是 `vmem_t.start_stack`，结束地址是栈指针寄存器 `sp` 所保存的虚拟地址。为了防范恶意攻击，起始地址 `start_stack` 一般是随机选取的。

5.2.4 VM Area Struct

计算机科学家：“Linux 用 `vm_area_struct`（VMA）这个结构体来描述地址空间，我们则用第 7 章中的代码 7.2 来近似。”

电子科学家：“你这扯得有点远了，第 7 章是你和数学家的对话，我怎么知道你们说了什么。”

计算机科学家：“你往后翻翻不就看到了，大概看一下，看完了再回到这一页来。”

计算机科学家等电子科学家看完代码 7.2，说：“刚才我们已经提过，整个地址空间被划分为不同的区域，每一个区域都由若干 4096 字节的‘内存页’组成。Linux 用 `vm_area_struct`

表示每一个区域，并且把它们当作链表一样连接起来，如图 5.3 所示。”

每一个区域都有一个对应的 `vm_area_struct` 结构体，用来描述这个区域的起始虚拟地址（`vm_start`）、结束虚拟地址（`vm_end`），以及指向下一个区域的指针（`vm_next`）。这些结构体被储存在内存上，它们的虚拟地址储存在内核空间里。

在同一个区域中，每一处字节都具有相同的性质：可读（`r`）、可写（`w`）、可执行（`x`）、私有（`p`），这些性质由 `vm_flags` 记录。如标记为 `r-xp`，表示这个区域的数据是进程私有的，并且数据可读、可执行；`rw-p` 表示可读、可写、私有。

不仅如此，`vm_area_struct` 还会记录这块区域是“匿名”还是“文件映射”。如果是匿名区，会记录匿名区的相关信息。如果是文件映射，则有指针指向对应的文件描述符。

电子科学家：“这就是你让我先看第 7 章中的代码 7.2 的原因吗，这是用链表组织的结构体，使我们只能按顺序访问。”

计算机科学家：“是的，例如要查找和栈有关的结构体，我们不得不从代码段开始，通过 `vma_t.next` 或 `vm_area_struct.vm_next` 指针向后查找。如果程序加载了多个共享库，那么 `mmap` 部分的结构体可能会非常多，这样一来就要花费很多时间。”

他继续：“因此，Linux 给 `vm_area_struct` 增加了一个**红黑树**（Red Black Tree）。红黑树是一种平衡的二叉搜索树，在这里，红黑树以 `vm_start` 作为索引值，就可以根据虚拟地址在 $O(\log(N))$ 时间内找到对应的结构体。”

但计算机科学家特别提醒：“不过，Linux 正在使用另一种数据结构。新的 Linux 内核代码正在用**枫树**（Maple Tree）去替代红黑树，作为 VMA 的索引。”

电子科学家：“所以说，每一个进程只能访问这些区域内的内存地址，访问其他内存都是非法的，是吗?”

计算机科学家：“是的。这告诉我们：在虚拟内存上，进程的内存访问一定集中在几个区域。而整个虚拟内存空间上，有很多内存是用不到的，呈现一种稀疏性。这种特性才使得我们能够使用页表来做虚拟地址的映射。”

5.2.5　用户空间与内核空间

计算机科学家：“最后，我们来看一下虚拟地址怎样划分**用户空间**（User Space）与**内核空间**（Kernel Space）。”

RISC-V 中虚拟地址可以选择不同的长度，目前有 32、39、48、57 位。我们考虑 48 位虚拟地址的情况（Sv48），因为它用了经典的四级页表结构。Sv32 是二级页表，Sv39 是三级页表，Sv57 是五级页表。

Sv48 规定：最高的 17 位必须是 $\texttt{0x00000}_{17}$ 或者 $\texttt{0x1ffff}_{17}$。因此，整个 2^{64} 字节的地址空间被划分为两片区域，一片是 `0x0000000000000000`～`0x00007fffffffffff`，另一片是 `0xffff800000000000`～`0xffffffffffffffff`。这两片空间的大小都是 128 TB。其中，低地址的部分称为**用户空间**，高地址的部分称为**内核空间**。

电子科学家：“那居于中间的地址呢？它们代表什么?”

计算机科学家一脸神秘莫测：“在用户空间与内核空间之间的，则是一个巨大的空洞，大小大约是 16TB。”

电子科学家困惑道："空洞?"

计算机科学肯定："没错，空洞。这一段地址没有任何作用，在 Sv48 的规定下，是非法的内存地址。"

用户空间主要用来存放用户数据，也就是图 5.3 中所列举的数据。而内核空间的访问是受限的，只有当 CPU 处于一定的**特权模式**（Privileged Mode）时，才可以访问内核空间的内存。

计算机科学家："打一个比方，用户空间就像是一个工厂的生产环境，应用程序就像是雇员，在生产环境中工作。而内核空间就像是雇主的办公室，只有雇主才可以进入办公室，从而管理雇员的行为。"

sstatus 寄存器保存特权级别

CPU 当前是否处于特权模式，由控制寄存器 `sstatus` 保存。在 CPU 中，除了有 `x0`～`x31` 这样的通用寄存器外，还有一种特殊的**控制与状态寄存器**（Control and Status Registers，CSR）。在 CSR 中，有一个**监管者状态寄存器**（Supervisor Status Register），即 `sstatus`，由它保存当前 CPU 是否处于**"监管者"**（Supervisor）的特权级别。当 CPU 处于**"监管者模式"**（Supervisor Mode，S-Mode）时，也就是我们常说的"内核态"；处于**"用户模式"**（User Mode，U-Mode）时，也就是我们常说的"用户态"。

计算机科学家："内核空间的内存主要用来管理进程的运行，例如，分配我们刚才介绍的 `vm_area_struct/vma_t` 链表、红黑树、`mm_struct/vmem_t` 内存。当进程要与一个共享库进行动态链接时，要把 CPU 切换到特权模式，执行操作系统内核的指令，新增 `vm_area_struct/vma_t`。这部分工作完成以后，再退出特权模式，执行用户程序的指令。"

电子科学家："那内核的权限是非常大的。"

计算机科学家："没错，因此内核空间的内存是绝对不能暴露给用户程序的，一定要通过特权模式保护起来。否则我们甚至可以自己写一个程序，去修改自己的地址空间了。这是极其危险的，这样一来，黑客可以轻而易举地通过一个普通的程序获得所有其他进程的信息，窃取银行卡密码更是不在话下。"

5.3 多级页表：虚拟地址映射

电子科学家："内存除了 DRAM 芯片外，还真是复杂啊。"

计算机科学家："你说得对。有了这些思想，我们甚至不需要 DRAM，而可以把任何可读写、可寻址的介质当作内存来用，只不过运行效率可能会降低很多。嘿嘿，到那时候，我可就用不着你的帮助了。"

电子科学家充耳不闻："别跑题了，还是讲讲页表吧。"

计算机科学家："行。到此为止，我们终于可以开始讨论虚拟内存的映射——页表。这是通过操作系统（软件）与处理器（硬件）协调工作，才得以实现的。"

他说："我们来回顾一下为什么需要一个映射，将虚拟地址转换为物理地址。在 5.1 节中，我们了解了 ELF 文件的格式。在 5.2 节中，我们了解了进程在执行时，它的虚拟地址空间是怎