# 第一章 计算机系统漫游

**1.文本文件和二进制文件的区别**

由ASCII组成的文件，叫做文本文件，而由二进制组成的文件，叫做二进制文件

**2.从源文件到目标文件经历的过程**

源文件以hello.c为例

**预处理：**预处理包括头文件的处理 #include 和注释的去除，解析include头文件，从系统或者是编译器中找到头文件的内容，插入源程序，得到一个新的C程序，此时以hello.i为后缀。

**编译**：将.i文件翻译成.s文件(生成的汇编代码)

**汇编**：将.s文件生成可重定位目标程序，生成.o文件，是二进制文件。机器语言

**链接**: 将.o文件与我们所需要调用的库函数所在的预编译好的目标文件关联在一起，使得可以调用库函数。

最终生成可执行文件，可加载至内存进行运行。

**3.shell**

shell是一种命令行解释器，将命令行输入的命令，进行执行。

**4.计算机系统的硬件组成**

**总线：**在计算机系统的各个部件之间传递信息。总线的字长与系统位数有关，或者说系统的位数取决于总线的字长(4字节还是8字节)

**IO设备:**这个好理解，每个IO设备通过IO控制器或者是适配器与IO总线相连。

**主存：**存储设备，我们可以认为主存包括内存和高速cache，是一组DRAM。主存会把程序或者是数据传入CPU，供CPU处理。

**处理器：**处理器的核心是一个大小为一个字的存储设备，称为程序计数器，程序计数器中指向着主存中的某条机器语言指令，即存放该指令的地址。处理器一直不断地执行程序计数器所指向的指令，然后更新程序计数器，使得其指向下一条要执行的指令。

为了解决从内存中读取数据慢的问题，引入了cache，cache中存储了近期可能需要的信息，cache是在CPU当中的。

**5.操作系统**

应用程序对硬件设备的访问，都需要经过操作系统。

应用程序的状态信息，我们称为上下文。在进行进程的切换时，操作系统会保存当前进程的上下文信息。(上下文信息是保存在哪里)(应用程序，或者说任务的状态信息，我们称为上下文)

当应用程序需要操作系统的某些操作时，触发系统调用，把CPU的控制权交给操作系统(操作系统说白了也是代码段而已)

**6.虚拟内存**

虚拟内存是一个抽象概念，它为每个进程提供了一个假象，也就是每个进程都在独占使用内存，每个进程看到的内存都是一致的，称为虚拟地址空间。

# 第二章信息的表示和处理

## 2.1信息存储

字节是大多数计算机的最小可寻址内存单位。一个int类型的数据，如果占4个字节，则对应连续四个地址。内存的每个字节都由唯一的地址来表示。这个地址，一定是虚拟地址，我们在编写程序时，不会接触到物理内存和物理地址。物理内存和物理地址的分配，需要由操作系统和内存、磁盘等硬件结合实现。

例如，一个类型为int的变量x的地址为0x100，也就是说，地址表达式 &x=0x100，那么在内存中，x的存储地址为0x100,0x101,0x102,0x103。

当x的值为0x00001234,一共是四个字节的数据，当最低有效字节占据最前面的地址(最前面的地址，指的是编号最小的地址)，我们称为小端模式。当最高有效字节占据最前面的地址，我们称为大端模式。

### 2.1.2 字数据大小

计算机的字长一般是32位或者64位。字长有以下几个含义：

首先代表着虚拟地址的最大大小，32位机子的虚拟地址大小为4GB(2^32-1)。(这其实和内存大小差不多了，感觉无法起到虚拟内存的扩容作用，但是目前一般是64位机子)。

其次，还代表着地址线的数量为32或者64。和虚拟地址大小相互对应。

另外，还代表着寄存器的大小为32位还是64位，代表着一次能处理的数据大小，这其实和数据线的数量相互对应。

**·不同字长机器下的变量长度**

在32位机和64位机中，long类型的长度是不一样的，在32位机子中，long类型依然是4个字节，而64位机中，long占8个字节。

但是对于double类型变量，在32位机中仍然占8个字节，可能是分两次处理。

当没有特殊说明，大部分数据类型都被编码成有符号数值。除非加上”unsigned”,才是无符号数。

### 2.1.3 寻址和字节顺序

对于一个多字节数据，它的地址是所使用的字节中，最小的地址。对于一个多字节数据，在内存当中绝大多数情况是连续存储的。

当最低有效字节数据在最前面时，称为“小端法”；当最高有效字节数据在最前面时，称为“大端法”。(字节顺序应该是由处理器决定的)

**·关于tyoedef关键字的用法**

typedef，其实是给关键字起一个别名。比如说typedef unsigned char \*byte\_pointer。机器或者说系统，或者说编译器，能将byte\_pointer识别为 unsigned char \*。至于为什么能这样识别，而不是将 \*byte\_pointer识别成unsigned char ,直觉上感觉是编译器识别的。(也有可能是因为\* byte\_pointer不符合C语言命名规范)

typedef和define的区别，typedef是取别名，更灵活，而define则是完全的字符代替。

### 2.1.7C语言中的逻辑运算

**·对于异或这一逻辑运算的特殊使用**

当变量A和自己进行异或时，会得到结果0

而0和任意一个变量A进行异或时，会得到变量A(保持不变)

利用“异或”运算这一特点，我们可以实现在不借助第三个变量的前提下，完成两个变量值的交换。代码如下：

void inplace\_swap(int \*x,int\*y)

{

\*y=\*x ^ \*y;

\*x=\*y ^ \*x; //第一行代码代入第二行，可以得到\*x = \*x ^\*x^\*y = \*y(位运算满足交换律)

\*y = \*x^ \*y;//此时的 \*y(值已经改变了) = \*x^\*y(原始值); 则最后\*y = \*y(x的值变成y了) ^ \*x ^ \*y = \*x

}

这样就可以实现在不借助第三个变量的情况下实现交换两个数。(在处理器层面，肯定还是用到了额外寄存器)

但是这种交换方法没有太大的实际用处吧，因为对于大多数人来说，不好理解。并且当x和y指向同一片内存空间时，在执行第一行代码时，会将该片内存空间置0，导致出错。

### 2.1.9 C语言中的移位计算

对于一个w位长度的数据类型变量(比如说int 长度为32bit)，当对其进行k个长度的位移时，当k＞w，一般编译器都会对k进行优化，最终移动的长度k1=mod w。但是一般要求位移长度小于数据长度(等于也不好，大多数编译器会优化成不位移)。

## 2.2 整数表示

### 2.2.1 整数数据类型

**·unsigned类型**

unsigned 等价于unsigned int ，定义为无符号的32位数据。

### 2.2.4 强制类型转换

当无符号数和有符号数发生强制类型转换时，编码不变，对二进制的解释发生变化

当发生有符号数和无符号数的比较时，会将有符号数默认强制类型转换为无符号数进行比较。

## 2.3 整数运算

### 2.3.2 补码加法

对于补码加法，存在正溢出和负溢出两种溢出可能。对于四位的数据，补码表示的范围为-8 到+7。当我们实现x+y操作时，当x+y的值小于-8，那么会发生负溢出，计算结果为正数(因为x+y的值可能会大于-16，再由于溢出会加上2^4,所以结果肯定大于0)。同理，当x+y的值大于8时，会发生正溢出，计算结果为负数(原因同上)

简而言之，当运算结果超过补码的表示范围，那么就会通过+-2 ^w，将运算结果保持在表示范围内，这就是溢出现象。

### 2.3.4 无符号乘法

整数乘法在计算机中如何实现？

可以一位一位算，也可以分解成2的幂 然后相加。后者大概需要三个时钟周期。而前者需要十几个时钟周期，因此在现代处理器当中，大多使用移位相加的方法。

### 2.3.5 补码乘法

对于K位的两个数相乘，当其乘法表达式的结果，在k位补码表示的范围内时，计算结果就会是正确的，当溢出，结果就会出错。举两个例子：

以4位二进制数为例，-2的补码表示为1110 ;-3的补码表示为1101。对这两个补码进行相乘运算(可以列竖式计算，也可以移位计算)，计算结果为10110110 ;截取4位后，得到的结果是0110;刚好是6的补码形式。

-2的补码表示为1110；3的补码表达形式是0011；对二者进行相乘运算，计算结果为101010，截取4位后，结果是1010，刚好是-6的补码形式。

**·对乘法结果进行扩展**

一般来说，乘法的运算结果的位数和两个数的位数相同。但是当会对乘法结果进行扩展时，那么我们就将运算的两个数，同样进行扩展，然后按照上面的运算方法进行计算。

### 2.3.6 乘以常数

在上面一小节已经说过了，对于乘以某常数，往往会拆分成移位+相加的形式。当然，这种乘法也存在溢出的可能性，和上面的情况是一样的。

### 2.3.7 除以2的幂

不管是无符号数的除以2的幂，还是有符号数，在右移时，不可避免地会将低位数据忽略，除以2的幂的最终结果是整数，是取整的结果。

**·有符号数的除以2的幂**

对于有符号数，在右移时，应该按照算术右移，最高位符号不断向右扩展。

对于有符号数，正数和负数都是向下舍入。但是我们希望最终结果是向0舍入，因此在移位之间会加上一个偏置量。这个偏置的大小，和我们的移位长度k有关。

## 2.4 浮点数的表示

### 2.4.2 IEEE浮点数表示

IEEE将浮点数的位划分成了三个字段，并采用如下的形式表示一个数。

其中s表示符号位，决定了V是正数还是负数，而E称为阶码，决定了权重值。并且E可正可负。M表示尾数，类似于科学计数法的小数部分。

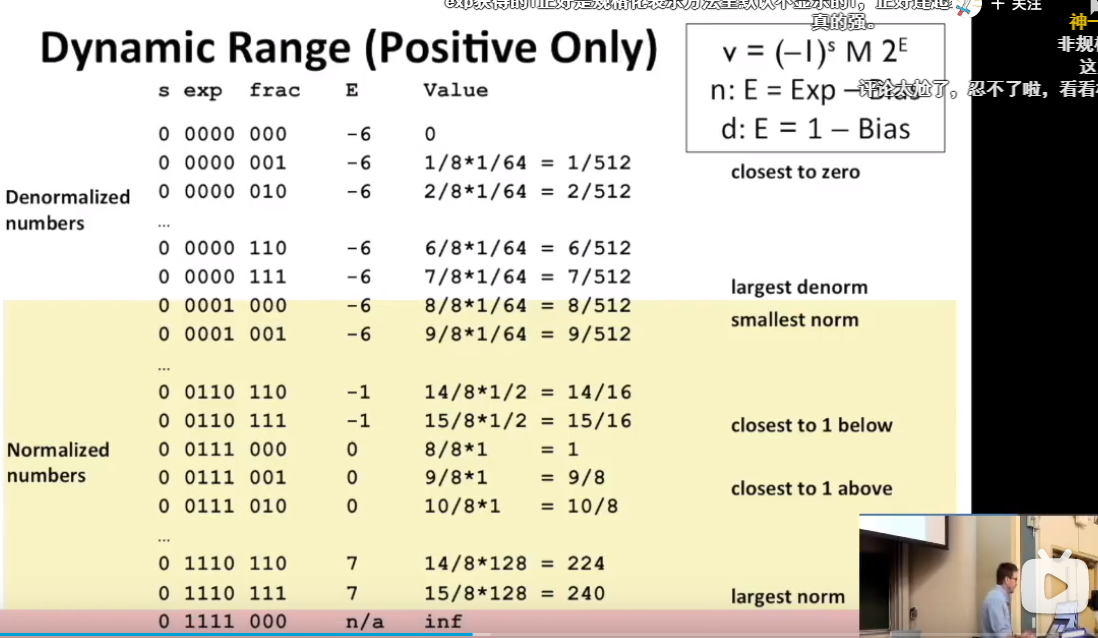
对于float和double类型，分别有固定长度的字段来编码符号位、阶码和尾数。

根据阶码字段的内容，可以将浮点格式分为规格化的值和非规格化的值。以float为例，阶码字段exp共有8位二进制表示。当阶码字段不全为0和全为1时，表示为规格化的值，此时E=2^exp -2^7;这样可以将E的范围扩展到-126到+127。

对于规格化的值，尾数M=1+n位小数字段的编码值。n位小数字段的编码值在0-1之间。比如:n位小数字段的值为1001，则M=1+9/16 = 25/16

当exp的值全为0时，用来表示非常接近0的浮点数，此时E=1-bias，其中bias=2^7

当exp的值全为1时，根据后面尾数的编码字段是否为0，可以定义为无穷大数或者是不存在数。

 下图是简化的浮点型编码的例子。白色部分是一些非常接近0的数。而黄色部分是规格化的编码值。红色部分是无穷大值。

### 2.4.4 舍入

浮点数的舍入遵循四舍六入五取偶，也就是当最后取整位为5的时候，取有效值为偶数的值。

### 2.4.5 浮点运算

第一步，对阶:把指数小的数的指数，转化为指数大的数的指数，并且吧指数小的数的尾数补0

第二步，将尾数相加减

第三步，规格化，规格化对尾数进行移位操作，尾数的移位操作会涉及到阶码的移动。

第四步，根据所给编码的长度，进行舍入

第五步，检查是否溢出

(浮点数的运算，总体上不太了解)

正是因为浮点数的运算存在上述五个过程，因此其舍入存在误差的可能性非常大，因此浮点数的运算不具有结合律。特别是对相差较大的数进行运算时。

float中的frac域没有足够的位来表示int(int需要32位，而frac域只有)

double的frac有52位，足够表示int

# 第三章程序的机器级表示

## 3.1 历史观点

对于32位和64位的理解。

这个位数，应该是一次性处理数据的长度大小。32位处理器一次性可以处理32位数据，而64位处理器一次性可以处理64位数据。这个位数并不是代表着寻址能力。譬如8086是16位处理器，但是地址总线是20根。所以这个位数和寻址能力(1mb)，没有必然的联系。

**·指令集架构**

指令集架构是来定义机器级程序的格式和行为，它定义了处理器状态，指令的格式，以及每条指令对状态的影响。大多数指令集架构都把程序的行为描述成每条指令按照顺序执行，但是实际上是并发地执行许多指令。

大体上分为RISC(精简指令集)和CISC(复杂指令集) x86是复杂指令集，而arm架构是精简指令集。

CISC处理的是不等长指令集，它必须对不等长指令集进行分割。

RISC处理的是等长精简指令集，速度较快并且性能稳定。

IA-32 因特尔32位体系架构

8086架构 IA32架构 x86 64架构

## 3.2 程序编码

#### 3.2.1 机器级代码

**·处理器内部寄存器定义**

**程序计数器(PC)，**里面存放的是下一条执行的指令在内存当中的存放地址(存放的应该是相对地址)，在x86中，用%rip来表示。

**整数寄存器，**x86共有16个整数寄存器，存放整数或者地址，具体如下所示。

(我们可以称为整数寄存器)

%rax :保存程序的返回值

%rbx:被调用者保存常做内存数据的指针

%rcx:计数寄存器，常用于字符串和循环操作中哦过的计数器

%rdx：数据寄存器，常用于乘除法和IO指针。

%rsi：来源索引寄存器，存储器指针、串指令中的源操作数指针。（rsi常用于保存函数的第二个参数）

%rdi: 目的索引寄存器(destination index)，存储器指针，串指令中的目的操作数指针（rdi常用于保存函数的第一个参数）

%rbp:基址指针寄存器，被调用者保存/栈基址寄存器，指向栈底

%rsp:堆栈指针寄存器 用来保存程序栈的结束位置，指向栈顶。(用来指明运行时栈的结束位置这是什么意思？不太明白)

%r8 to %r15

**条件寄存器**：存放最近一条执行指令的一些状态信息。包括进位、溢出、是否为零、符号位等。

**向量寄存器**：存放一个或多个整数或者浮点数值。

## 3.4访问信息

### 3.4.2 数据传送指令

·MOV指令

mov 源操作数 目的操作数

源操作数即可以是立即数，也可以是寄存器，也可以是内存引用，而目的操作数只能是寄存器或者是内存引用。但是mov指令的源操作数和目的操作数，不能都是内存地址

几个mov指令的例子如下所示：

movl $0x4050, %eax

movw %bp , %sp

movb (%rdi,%rcx),%al

movb $-17 , (%rsp)

movq %rax , -12(%rbp)

mov指令的后缀，和寄存器的大小需要匹配。

当mov指令的源操作数是立即数时，只能是32位的补码表示形式，并且对该数值进行符号位扩展后存入目的操作数。

任何对寄存器传送32位数据的指令，都会把该寄存器的高位置为0。

### 3.4.3 数据传送示例

在这一章中，提到了“局部变量通常是保存在寄存器当中”，但是在链接那节，提到了“局部变量保存在栈当中”。栈和寄存器？他们的关系如何理解？

### 3.4.4 压入和弹出栈数据

入栈和出栈，对应的栈，称作程序栈。程序栈是否是一个具象概念？程序栈最终在内存当中，开辟出一个空间，存放程序栈？是吗？

栈和程序代码，以及其他形式的程序数据，都是放在同一内存当中的。

当我们向栈内push一个数据时，默认是八字节数据，则先将栈指针-8(因为栈是向下伸展的)，然后往对应的地址写入数据。往栈顶写入输入。

pop一个数据也是同理。

(对栈读数据和写数据，一定需要八个字节八个字节地读写吗？不一定吧)

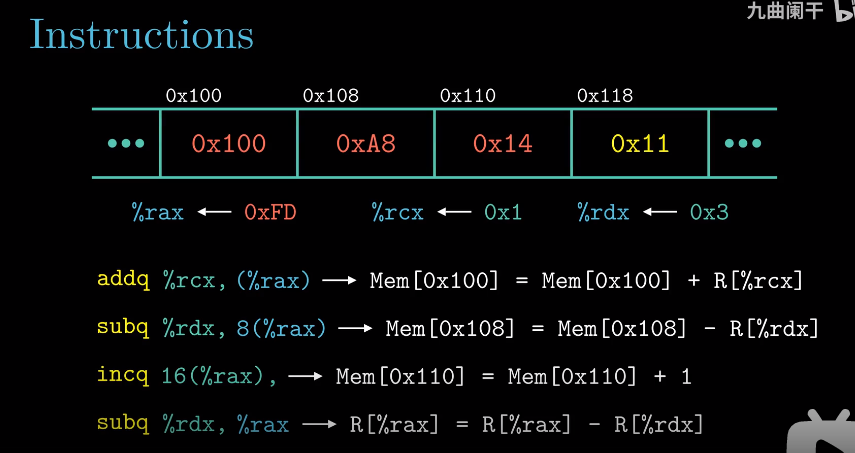
当在函数A中调用了函数B，那么我们称A为调用者，B为被调用者。

寄存器的保存策略：调用者保存，被调用者保存。当函数A调用函数B，并且函数B中会用到函数A中的使用的rbx寄存器，那么rbx寄存器在被再次使用之前，需要先将rbx原先的值保存住(保存策略分为调用者保存和被调用者保存)。则需要用到push和pop一对指令，进行入栈出栈。

## 3.5 算术和逻辑操作

### 3.5.1 加载有效地址指令

leaq 7(%rdx, %rdx,4) , %rax 这条指令是指把有效地址复制到寄存器rax中。有效地址的计算方式和内存地址的计算方式相同。

 leaq指令不仅可以加载有效地址，还可以进行加法和乘法运算。

### 3.5.5 特殊的算术操作

x86的寄存器只有64位。理论上最大支持的运算结果是8位。但是对于两个8字节的数据相乘，x86也提供了这类特殊的运算操作。将高八位和低八位的数据，分别存放在特定的寄存器当中。

16字节的除法也是这样的。

## 3.6 控制

机器代码中，jump和call指令，可以实现指令的跳转。

### 3.6.1 条件码

条件码寄存器是由CPU来维护的，长度是单个比特位，他描述了最近一次执行操作的属性。

·常用条件码

CF: 进位标志码 (CPU执行最近的一条指令，最高位是否发生进位)

ZF:零标志 (最近一条的执行结果是否为0，为0则置1)

SF:符号标志 当最近的操作结果小于0时，符号标志会被置1

OF:溢出标志位 对于有符号数来说，最近的操作导致正溢出和负溢出时，OF都会被置1。

有两类指令，只改变条件，码，不改变操作寄存器的值，分别是cmp指令和test指令。cmp指令对两个源操作数进行相减操作，并根据操作的结果设置条件码寄存器。test指令对两个源操作数进行与操作，并根据操作的结果设置条件码寄存器。

### 3.6.2 访问条件码

访问条件码有三种方式。1、根据条件码的某一组合，将某个字节设置为0或1(这里用到了SET指令，SET指令有各种组合)。2、可以条件跳转(jump) 3、可以条件转移数据(MOV)。

### 3.6.3 跳转指令

跳转指令中，可以带上对条件码寄存器的某几位的判断，从而形成条件跳转指令。

基于条件传送的代码会比基于跳转指令的代码效率高，这是因为现代处理器通过流水线获得高性能。跳转会降低流水线的性能。

### 3.6.4 跳转指令的编码

跳转指令有多种类型的编码，但是最常用的是PC相对编码。也就是跳转指令会将目标指令的地址与紧跟在跳转指令的下一条指令的地址之差，作为编码。

比如说

0x01 jump 0x05

0x02 xxx

跳转指令的跳转地址的编码是0x05，下一条指令的地址是0x02，所以实际跳转的指令的地址是0x05+0x02=0x07。(现在大概理解了)

这部分内容在链接的重定位部分中，有比较大的作用。.

### 3.6.5 用条件控制实现条件分支

利用jmp系列指令和cmp指令。但是利用条件跳转指令，指令之间的连贯性并不是很好，会有一个指令跳转的过程。

### 3.6.6 用条件传送来实现条件分支

但是条件传送实现条件分支，就不会存在这样的问题。主要是没有jmp过程。条件传送同样需要类似于cmp或者其他指令，改变条件码寄存器。利用形如cmovge的条件传送指令，再决定进不进行条件传送。整个执行过程是顺序的。

在引入流水线指令后，条件控制这类指令跳转过程，需要处理器进行分支预测，当分支预测错误，CPU的处理代价较大，大致需要15-30个周期，对运行程序的性能影响很大。而条件传送不存在跳转的问题。

### 3.6.7 循环

**·switch语句**

switch指令是编写了一张跳转表。来实现根据变量实现跳转到不同的程序块的。但是如果待判断的值，比较稀疏，则编译器会将switch语句转换成if-else语句；如果待判断的值的最小值不是0，比如说比较大或者是负数，那么编译器会加上一个偏置值，将待判断的值的最小值变为0(类似于数组的最小索引值为0)。

## 3.7 过程

假设我们的函数P调用函数 Q，在调用函数Q之前，需要将程序计数器PC设置为Q代码的起始地址，然后在返回时，要将程序计数器设置为调用Q后的下一条指令的地址。这就要求，下一条指令的地址需要事先被存入栈中。

### 3.7.1运行时栈

理解栈和栈帧的概念。栈是运行程序时，不断扩大的内存空间，在里面会存放一些局部变量等内容。栈的大小是有上限的，上限和寻址能力有关，我们可以把所能寻址的地址，都分配成栈，理论上是可以的。但是一般栈不会设置得太大。

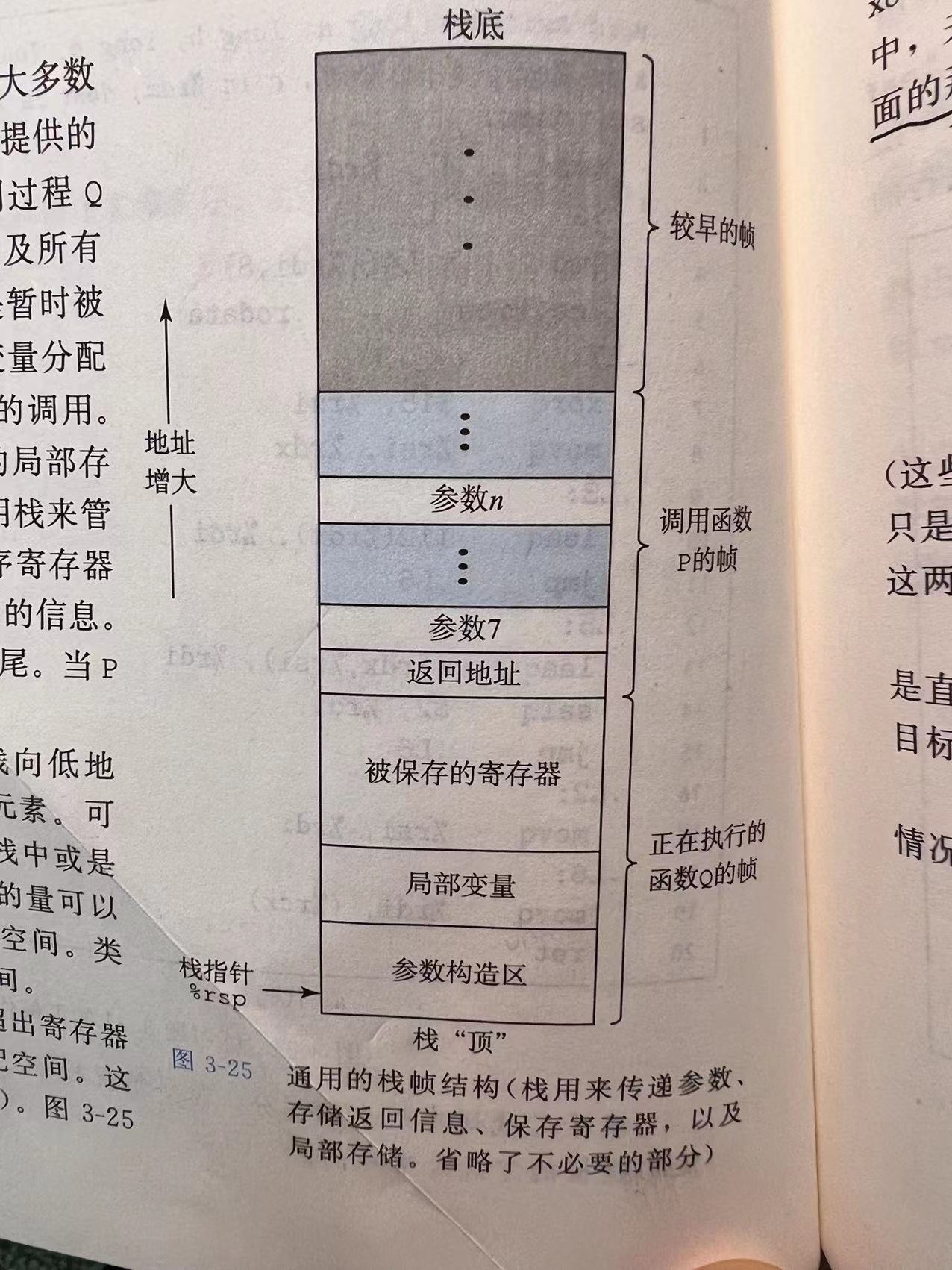
**·栈帧的概念**

在函数P调用函数Q时，P的局部变量、参数等数据，都会被暂存在栈当中，这一部分空间就叫做栈帧。(有点类似于上下文切换时保存数据)，并且会将返回地址存放在栈当中。可以认为是一种挂起状态。

在调用函数Q时，会在栈上分配局部变量等等内容，当Q返回时，这些局部变量所占据的内存都会被释放。然后将P的栈帧内容恢复到寄存器当中(当参数过多时，也会将一部分形参存放在栈帧当中)

栈帧的大小由两个指针相间隔，分别是rsp和rbp。rbp是基地址指针。在rsp和rbp之间的内存大小，存放的就是当前运行函数的栈帧。

当发生函数调用时，调用函数的rbp的值，也会存入栈帧中，并将当前rbp的值更新为rsp的值，表示重新开辟一个栈帧。

当前正在执行的过程的帧，总会在栈顶。

大多数过程的栈帧都是定长的，在过程开始时就分配好了，但是有些过程需要变长的帧。并且对于函数来说，栈帧是可选的，有些函数根本用不到栈帧。

栈帧的理解可以参考右侧

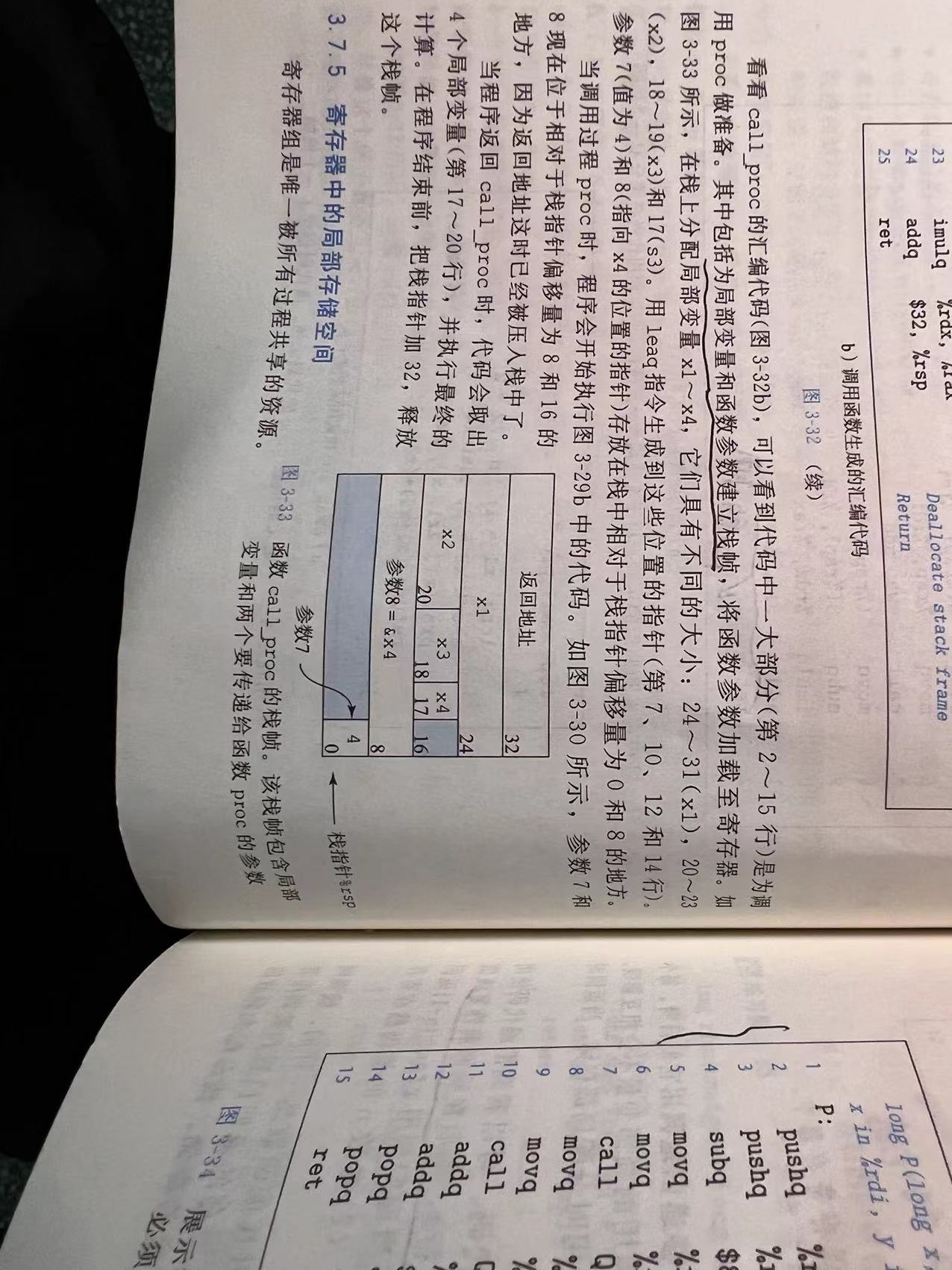
### 3.7.2 转移控制

通过栈传递参数时，所有的数据大小都向8的倍数对齐。

### 3.7.4 栈上的局部存储

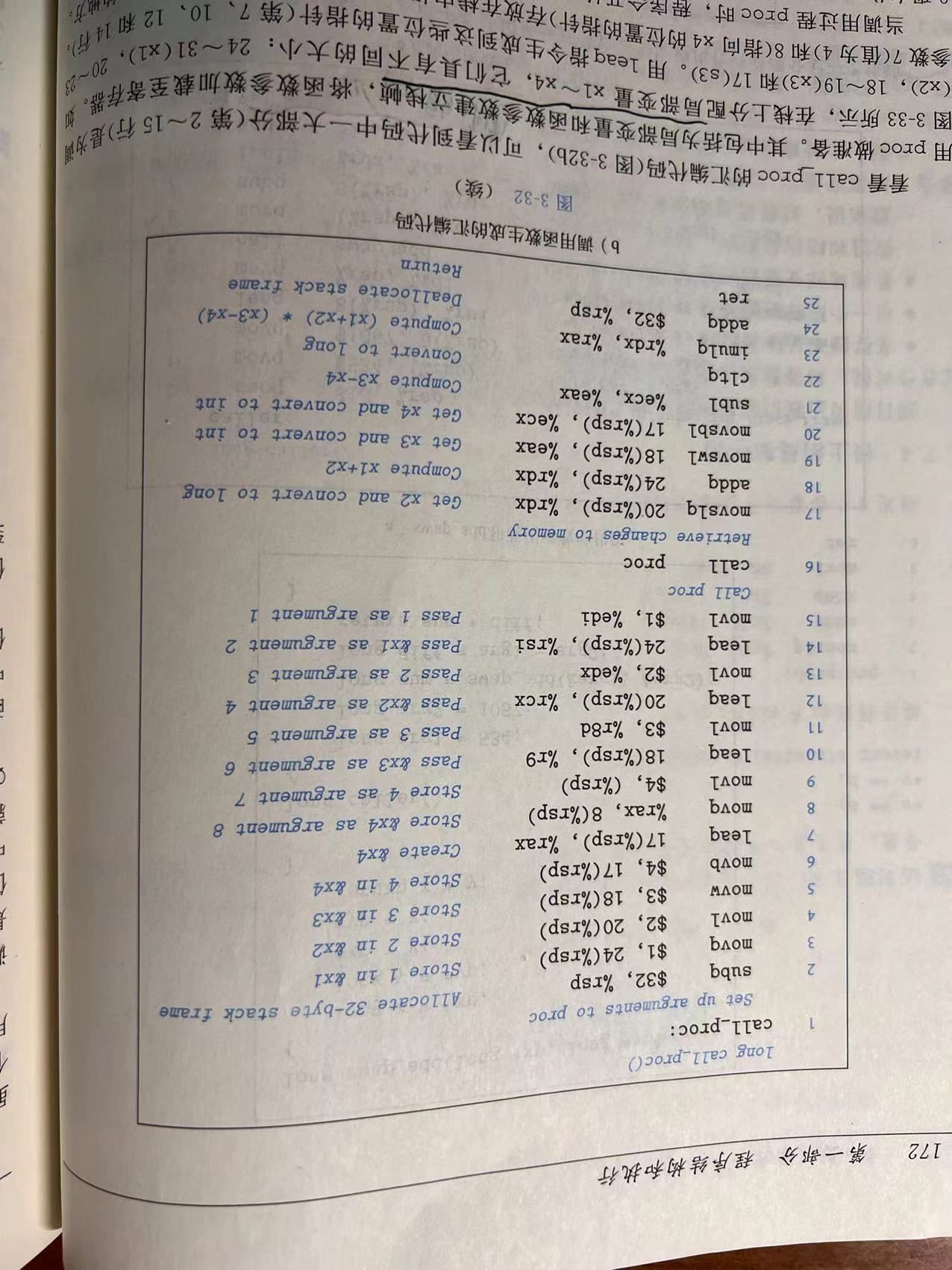
·如何理解在栈内数据的对齐

如下图函数所示：



分析栈的分配内容。当发生call\_proc()函数调用时，我们可以看到，在栈中存了一个返回地址，这是为了在函数完成后，能继续执行调用函数的下一条指令而入栈的。然后在函数中开辟了32个字节的空间(rsp-32,这个可以根据汇编来观察到)。前8个字节存long 1 ；接下来的4个字节存放int 2；接下来的2个字节存放short 3 ；接下来的1个字节存放char 4；然后接下来由于做调用proc函数前的准备，由于proc函数的变量超过6个，所以x4和&x4都会存放在栈中。都需要重新存放一遍在栈中。就是上图中的 倒数第二行和倒数第一行。

在这里格外注意在内存当中的对齐情况。如上图的灰色部分。某一类型的数据，或者说是字节数不同的数据，存放在内存当中的起始地址，一定要能整除其字节长度。



## 3.8 数组分配和访问

### 3.8.1 基本原则

**·指针数组和数组指针**

数组指针，是指一个指向一个数组的指针。比如说 int (\*p)[10]。括号优先级高，因此先是说明p是一个指针。在进行[10]。说明这是一个指向元素有10的数组。对p取内容，是第一个元素的值，但是p++并不会指向第一个元素。(以10个为一个改变的单位)。

指针数组，说明数组元素是指针。形容 int \*p[10]。每一个元素都是指针

**3.8.3 嵌套的数组**

对于下面这种定义，不太理解。

typedef int row3\_t[3];

为什么row3\_t[3]被定义成3个整数的数组？

## 3.9 异质的数据结构

在定义了一个新的数据类型后(struct去定义)，并没有在内存中为其开辟空间。在声明一个对象后，才会在内存中开辟空间。

段错误：访问一段还没有被分配的内存(地址有效但是未被分配)

所以无论何时，合理的地址都是低地址的这一部分和高地址的这一部分，中间是无人区，访问会引发段错误

## 3.10 在机器级程序中将控制与数据结合起来

### 3.10.1 理解指针

**·特殊类型的指针**

void \* :这代表着通用指针，可以对其进行强制类型转换，转换成一个有类型的指针。

比如说，malloc函数会产生一片内存，返回的指针类型就是一个void \*。我们通过显式的强制类型转换，转换成我们想要的类型的指针。不同类型的指针存在什么区别？当对指针++时，存在区别。地址向后移动对应类型的字节长度。

P192页的例子不太理解。

**·函数指针**

int fun (int ,int \*);

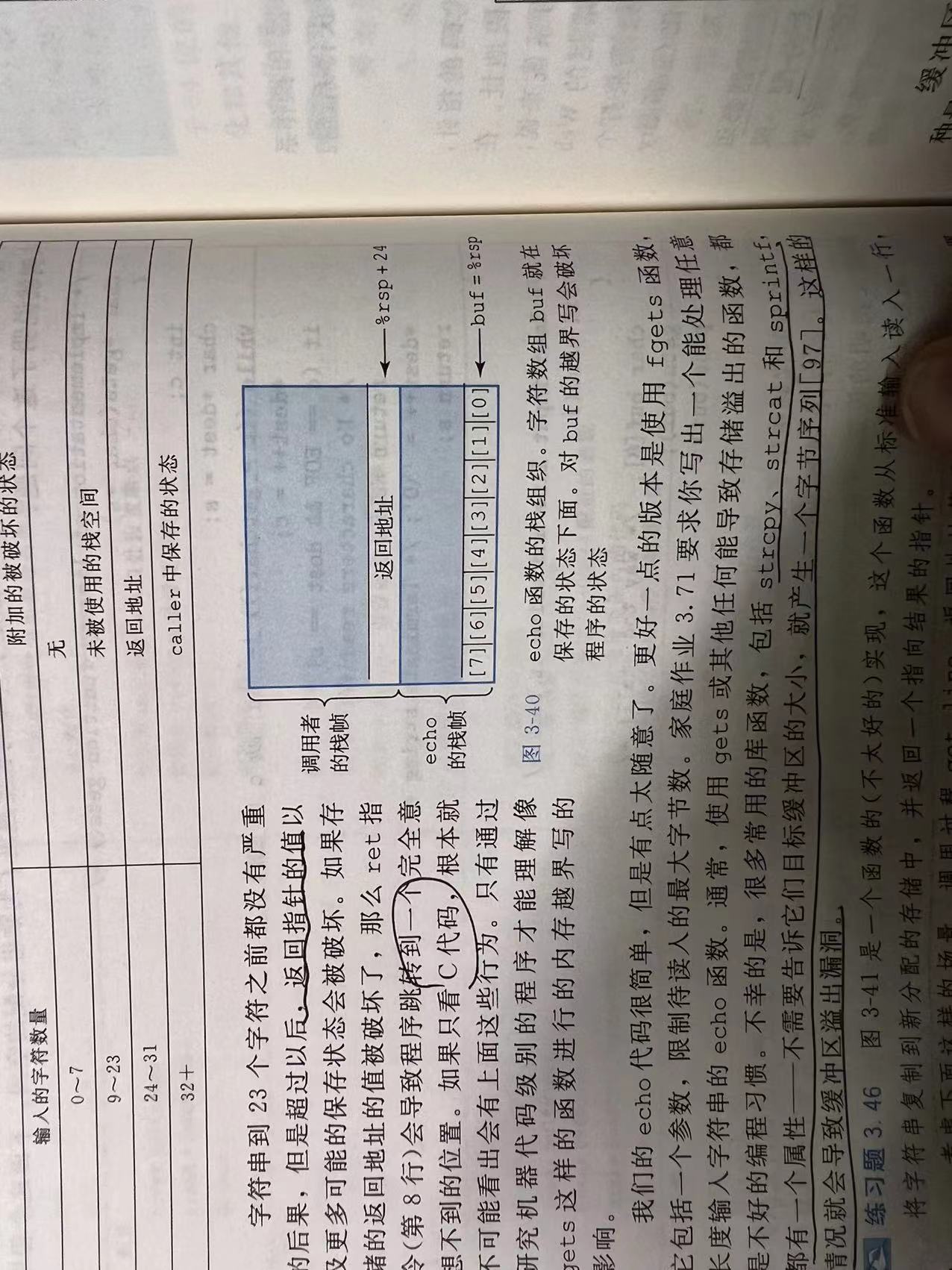
int (\*fp)(int ,int \*); //这说明 fp是一个函数指针，参数为int 和int\* ,返回值为int

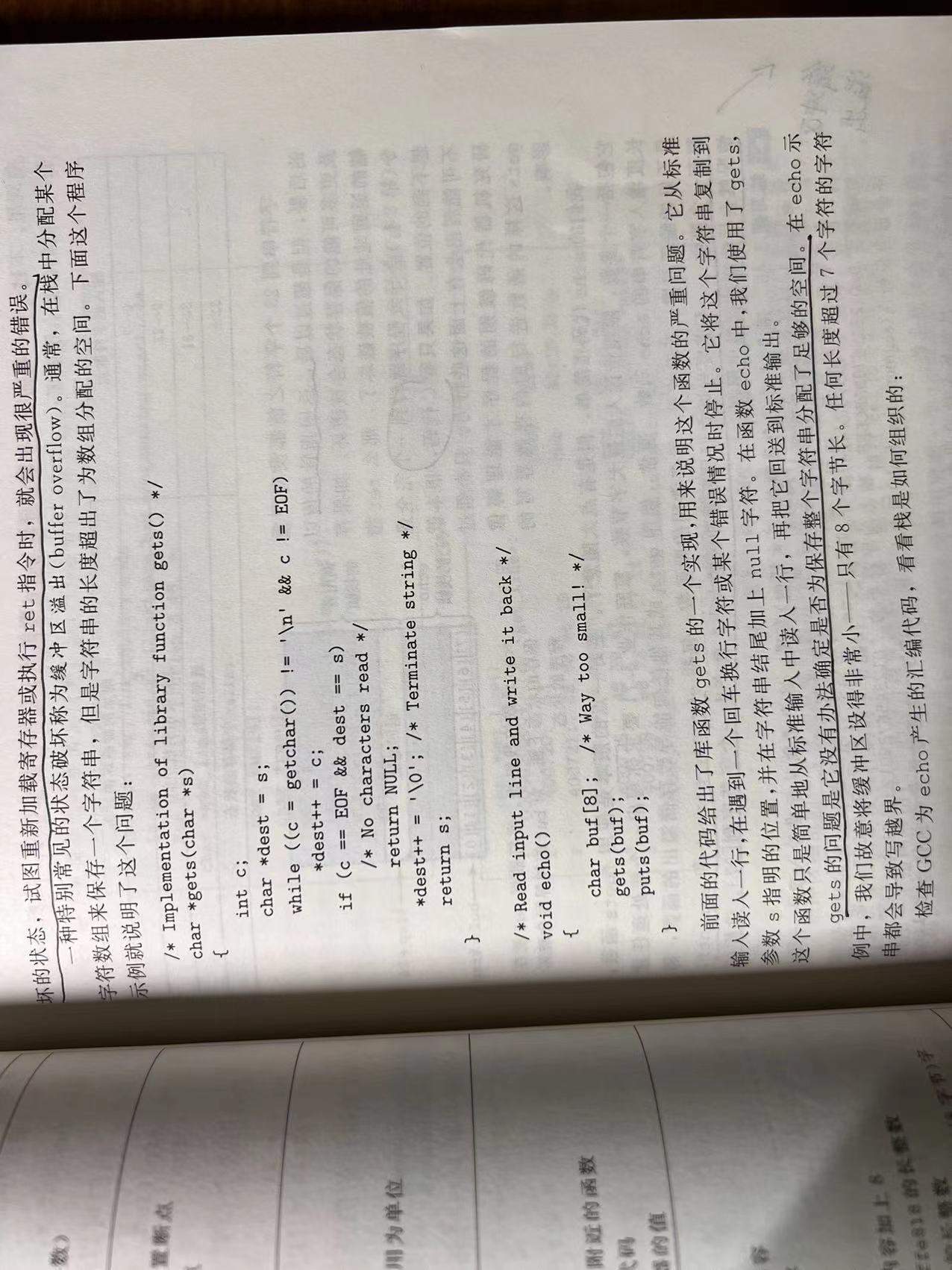
fp=fun ; //将该函数指针fp指向定义好的函数fun。参数和返回类型需要相同。

### 3.10.3 内存越界引用和缓冲区溢出

对于如下函数：

gets()函数从标准输入读入数据，存储在以char \*s为起始地址的连续字节当中。在echo()函数调用gets函数时，定义的缓冲区为char buf[8],也就是预留了八个字节的缓冲区。

 我们可以观察到其在内存中的分配(这个内存都是虚拟内存)



在存储了调用之前的返回地址后，开辟了echo函数的栈帧。我们可以看到，在分配缓冲区的之前还有一部分内存空间，可能是用来存储echo函数的局部变量等参数。当我们读取过多的数据时，会侵占这一部分内存空间，更有甚者会侵占返回地址。甚至造成代码攻击。

### 3.10.4 对抗缓冲区溢出攻击

这个缓冲区，就是在栈内局部分配的。可以采用栈随机化、栈破坏检测(金丝雀)以及限制可执行代码区域实现对抗缓冲区溢出攻击。

# 第四章 处理器体系架构

一个处理器支持的指令和指令的字节级编码称为他的指令集结构。

指令系统结构：是计算机软件和硬件交互的接口。程序员通过指令系统结构设计软件，CPU设计人员根据指令系统设计硬件。

一条汇编指令，可以根据一定的规定，转成机器代码。任何一个字节序列都是一个唯一的指令序列的编码。

状态码：表示程序运行的总体状态，这个值有可能是1 代表正常，2代表处理器执行了一条halt指令，3表示处理器遇到非法地址，4表示遇到非法指令。

在处理器内部，寄存器文件和算数逻辑单元(ALU)是串联的

## 4.3 Y86-64的顺序实现

·对于取指的理解

当取出当前指令的地址并找到当前指令时，我们需要及时更新PC，更新为下一条指令的地址。PC的更新值有几种可能性 1、当前PC+已经取出的指令长度(指令存储在连续地址空间中) 2、指令的数据段部分(跳转指令，可能根据条件判断结果，跳转地址设定为给出的立即数、以及call指令，也会发生跳转，但是call相比于j，会将下一条指令的地址存入内存当中) 3. 从内存中读出的数据段(比如说ret指令，就是从内存，或者说是栈中出一个地址，然后跳转至该地址)

当发生跳转时，下一条指针的地址，有可能是直接寻址，也有可能是间接寻址。

·译码阶段

从寄存器文件中读取数据

·执行阶段

算数逻辑单元主要执行三类操作：1.执行算数逻辑运算 2.计算内存引用的有效地址 3.针对push指令和pop指令

·访存阶段

访存阶段主要是针对内存的读写操作

·写回阶段

针对寄存器文件进行操作，写寄存器文件

·更新PC

将PC的内容设置成下一条指令的地址。

处理器在无限循环当前过程，当发生任何异常时，处理器就会停止，执行halt指令或者是非法指令，进行异常处理模式。

以上所有的过程，都要在一个时钟周期内执行完毕。(这句话有些片面了，其实有些指令的执行，会需要多个时钟周期。比如说乘除法)

ALU的功能：计算算数逻辑指令、还要计算访问内存的有效地址，以及针对栈指针的运算。因此，ALU的输出端口，会和数据内存的地址逻辑单元相连。

## 4.5 Y86-64的流水线实现

由于流水线实现过程中会涉及到控制相关。因此对于条件控制转移和条件数据传送来说，条件控制转移的预测更难，因此在有条件的可能性下，尽量选择条件数据传送。优秀的编译器也会帮我们对代码进行优化，最好是条件数据传送。

在非流水化的实现之前，开始下一条指令之前必须完成上一条指令的执行。

流水化操作是指当前一条指令进入下一阶段的过程后(已经进入译码阶段后，下一条指令可以进入取址阶段)

这要求将一个整体的设计，划分成多个延迟都相等的子阶段。在不同的阶段之间，插入寄存器，暂存数据。实际的运行速率，会受到最慢阶段的限制。

流水线操作的硬件实现：在不同的阶段之间，加入寄存器，保存状态。

之前一直是假设指令之间相互独立，会比较符合流水线的过程，但是实际上，指令之间可能存在存在依赖关系。可能必须在上一条指令执行完毕后，才会执行下一条指令。

这称为数据依赖。

·冒险的处理

对于加载使用冒险，一般采用暂停和转发，实现对冒险的处理，但是不可避免地会浪费时钟周期。

**数据依赖：**

当执行下一条指令时，需要用到上一条操作的寄存器的值时，如果上一条指令处于访存或者是执行阶段，那么读取的寄存器的值可能会出现错误，这种情况叫做数据依赖。

解决方法1：当CPU通过查看寄存器ID等手段，发现两条指令之间存在数据依赖，则下一条指令在译码阶段，会等待一段时间，在等待的时间中， 插入气泡，代替流水线执行。(那么等待多久呢？)

但是这种暂停方法，会大大影响流水线的效率。并且在执行过程中，数据相关的概率是非常大的。

解决方法2：数据转发，也称旁路。将上一条指令的执行结果，直接发送给下一条指令的译码阶段，不会先存入寄存器，然后再由下一条指令去取，这样可以加快指令执行

**控制依赖：**

当下一条指令的执行，需要上一条指令的结果做预测，我们就称为控制依赖。

解决方法1： 分支预测。假设分支总是跳转或者是不跳转。当出现预测错误之后，需要采用响应的方法进行处理。

在流水线中，引入反馈是非常危险的。我们必须想办法处理指令之间的数据依赖和控制依赖。

数据依赖可能会造成得到错误的值，可能下一条指令希望取某一寄存器的值时，上一条指令还在访问阶段，寄存器内的值还是0，也就是读取错误。

我们称为冒险或者冲突。

# 第五章 优化程序性能

## 5.1 优化编译器的能力和局限性

不同的指针，实际上可能指向同一片内存区域，这种情况叫做“内存别名引用”。编译器必须假设不同的指针可能指向同一片内存区域而进行优化，因此优化程度会受到限制。

或者说，不同的指针所指向的内存区域有重合的情况。

## 5.7 理解现代处理器

**·指令级并行**

在处理器中，可能会对多条指令同时进行求值处理，称为指令级并行。在运行上采用多条指令并行执行。但是在逻辑层面，呈现出顺序执行指令的现象

**·寄存器重命名**

寄存器重命名机制，将某一寄存器r与对其译码产生的标记t，记录在一张表中。当其他指令对r进行操作修改后，可以将修改结果直接记录在表中，加快数据转发的过程，避免必须将结果存入寄存器造成的时钟浪费，加快数据转发。

# 第六章 存储器的层次结构

## 6.1 存储技术

**·DRAM**

当使用内存控制器对DRAM进行管理时，内存控制器先通过地址总线发送行地址，DRAM会从存储空间中读出数据到行缓冲器。然后内存控制器再发送列地址，这样就选中某某一单元的数据，并通过数据总线读出。

**·传统DRAM的结构**

DRAM的芯片，一般是按照超单元展开，每个超单元都由若干位组成。(一般是八位，组成一个字节)

比如说一个16\*8的DRAM 芯片。有16个超单元，每个超单元的内容都是8bit数据。对超单元的寻址，一般按照行和列的方法寻址。

对于16个超单元的芯片，一般采用4\*4的结构，行地址需要两位，列地址需要两位，并且行列地址是依次发送的，所以地址线只需要两根。

内存控制器向DRAM先发送行地址，DRAM会将所有该行的数据保存到内部行缓冲区内，然后发送列地址，从缓冲区内读取一个超单元的数据。

**·机械磁盘：**

机械磁盘由磁道记录数据，分为扇区和盘片，并通过磁头读写数据

寻道时间：传动臂将磁头移动到目标扇区所在的磁道的时间

旋转时间等等，这个不重要。

**·机械磁盘的读写方法**

机械磁盘通给改变读写区域的极性，标定为1or0。同样的，通给读取的区域的极性，判断读写的数据为1or0。

**·固态硬盘：**

固态硬盘由我们的闪存芯片构成，使用的是Nand Flash.对于Flash来说，未写入数据时，所有位都是1，只能写0。

**·DMA**

当CPU发送读取硬盘中扇区数据的指令。IO控制器去执行该过程，IO控制器占据总线的使用权，将数据从硬盘读取到内存当中。此时这个读取数据过程，CPU是不可知的，这就叫DAM。当读取完毕后，IO控制器向CPU发送一个中断信号，表示DMA完成。

总线的使用权有多种仲裁方式。

当IO控制器在进行数据传输时，CPU可以去处理其他指令，提高执行效率。

**·ROM**

只读器件，bios就是存放在ROM当中，开机自启。自运行。

## 6.2 局部性原理

局部性分为空间局部性和时间局部性

·空间局部性

一个内存位置被引用了，那么在不久的将来，该内存位置附近的内存块被引用的概率高

·时间局部性

一个内存位置被引用了，该内存位置在不久的将来被重复引用的概率高。

## 6.3 存储器的层次结构

存储器的层次结构，从高到低，依次寄存器、L1缓存、L2缓存、L3缓存、内存，硬盘和网络存储等，第k层可以看成是第k+1层的缓存。

当我们需要查找第K+1层的数据时，我们首先去第k层去找到，是否有该数据的缓存。当在第k层能找到该数据时，称为缓存命中。当第k层无法找到该数据时，称为缓存不命中情况。

缓存不命中的原因有很多，其中有一种称为冲突不命中。冲突不命中的原因是缓存的放置策略造成的。

通常，硬件缓存通常使用严格的放置策略，即将第k+1层的某些块，限制放在第k层的某个位置当中。比如说第k+1层的块0、4、8放在第k层的块1中，第k+1层的块1、5、9放在第k层的块2中。

因此当我们需要访问第k+1层的4 和8时(交替访问)，那么就会反复出现冲突不命中的情况。

## 6.4 高速缓存存储器

### 6.4.1 高速缓存存储器

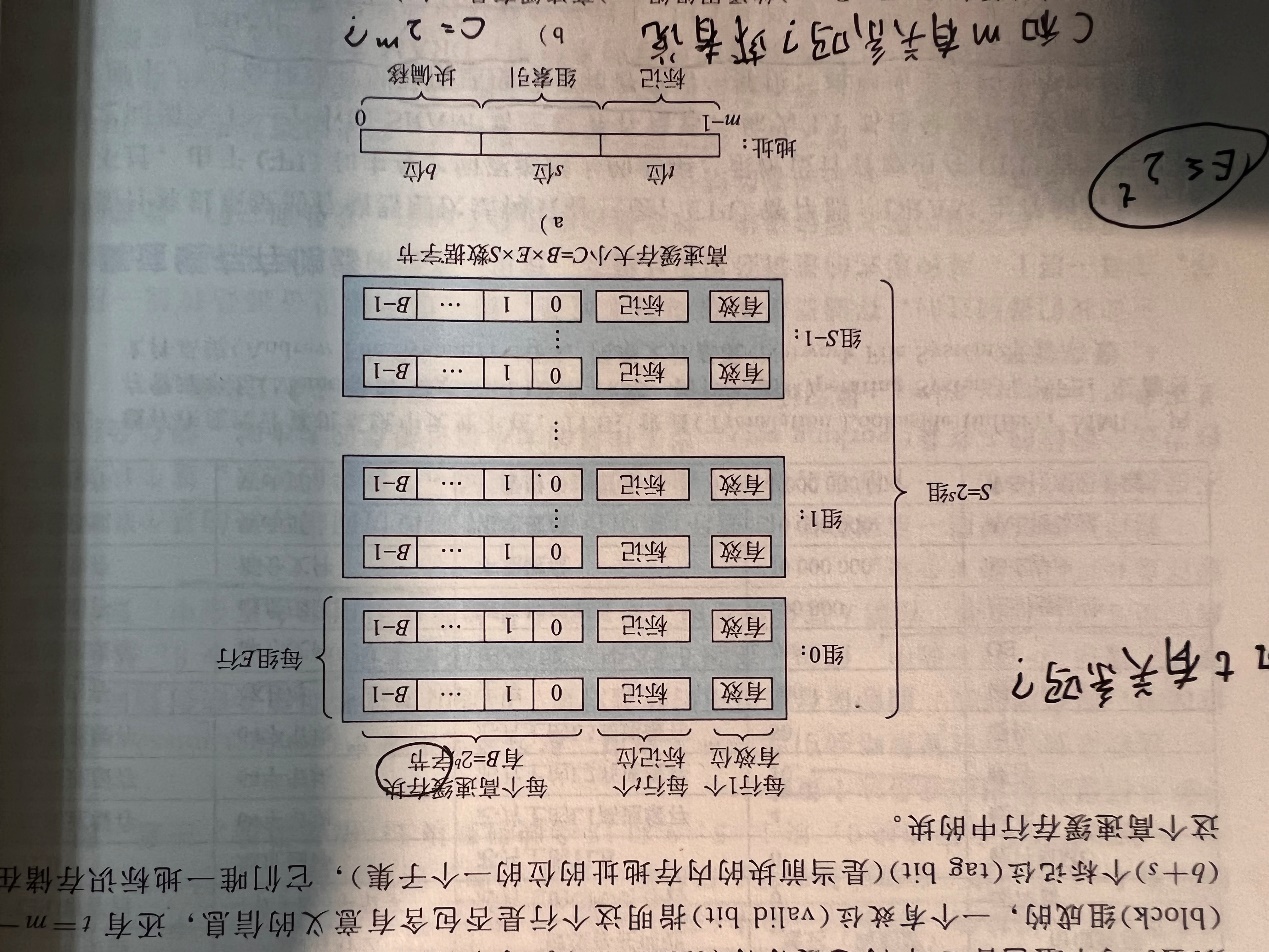
对于一个计算机系统，内存当作是m位，则内存中的每一个地址，都会有一个地址，则内存地址为2^m个。

但是高速缓存的大小，一定是远远小于2^m的。

**·高速缓存的组织形式**

高速缓存会根据内存地址(实际上就是物理地址)的s位，划分成S=2^s组别。在每个组别中，会包含E个高速缓存行。每个行都是由连续2^b个字节的数据块组成的。数据块由物理地址最后b位进行分类。在每一个高速缓存行中，还包含一些控制信息，包括有效位和标记位。在CPU进行地址查询时，只有组别和有效位以及标记位都匹配后，才认为找到数据，并且从数据块(block)中根据偏移量找到对应的数据。

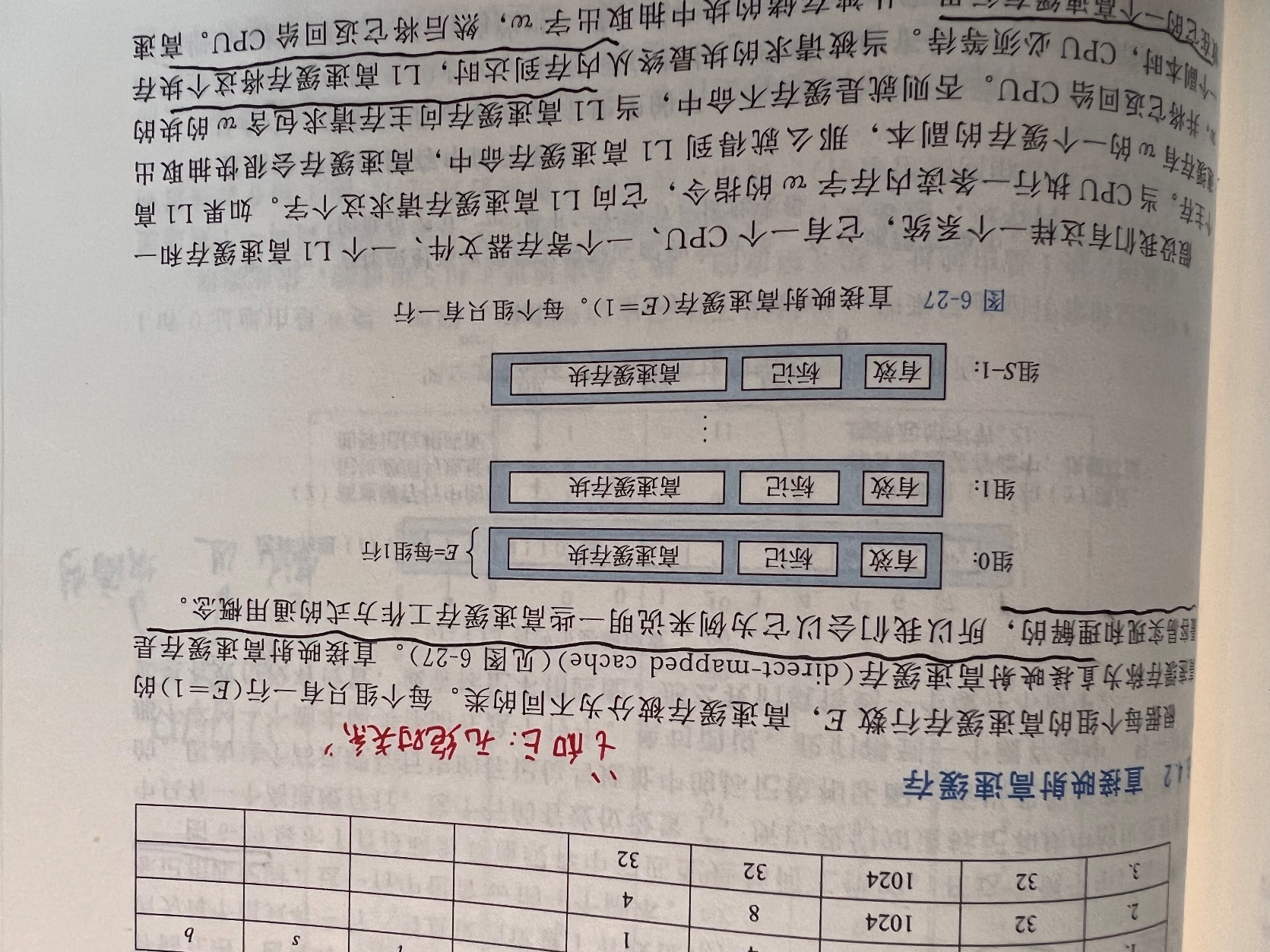
标记位的长度t=m-s-b。

 高速缓存的通用组织形式如下所示：

根据缓存内的组别数和一个组内的高速缓存行的个数，可以分为直接映射高速缓存、组相连高速缓存和全相连高速缓存。

当CPU需要从内存地址为A的内存出读取数据时，首先从cache中去查看该内存地址是否命中。如果命中，直接从cache中读取数据，否则向下一级缓存找寻数据块，替换上级cache的数据块。

### 6.4.2 直接映射高速缓存

 直接映射高速缓存的结构中，每个组别中，只有一个缓存数据行。

当发生缓存不命中时，数据替换以数据块为最小单位。

标记位和索引位，共同唯一标识了内存当中的某个块。

当直接映射高速缓存中发生不命中时，需要从存储器结构的下一级取出被请求的块，然后进行替换。

内存块中的不同数据块，是会映射到同一个组别的数据块中的，这也会造成冲突不命中。

P431中，包含冲突不命中的实例。

### 6.4.3 组相联高速缓存

组相连高速缓存，允许一个组内，都有个高速缓存行。

组相连高速缓存，在定位数据时，需要检查组内的每一行的数据的标记位，检查标记位的时间花销会更大。

**·组相连高速缓存中不命中时的行替换**

当发生缓存不命中时，从存储器结构中的下一层，取出数据块，如果当前组别中有空闲的缓存行，直接替换即可，否则，需要替换一个非空的缓存行。

这就涉及到缓存的替换策略，包括LRU(最近最少使用策略)，这也是最常用的替换策略。

关于全相联高速缓存，是组相联高速缓存的特例。

### 6.4.5关于写的问题

当我们需要往主存当中写数据时，有两种方法，首先是在缓存中更新数据后，直接写会低一层的存储结构，称为直写。

或者是等待替换算法要驱逐这个更新过的块时，才将更新后的数据写入下一层。(对于多核来说会存在数据更新不及时的问题)，称为写回。

当出现写不命中时，分为写分配和非写分配，写分配会从低一层的存储结构中把数据加载上来，然后更新。而非写分配绕过上一层，直接写进低一层的存储结构中。

# 第七章 链接

链接过程，我们可以认为是将各个程序段或者是数据片段，收集并组合成一个单一文件的过程。经过预处理、编译和汇编，我们可以得到可重定位文件(.o),链接器负责将各个.o文件，链接组合成一个可执行文件(链接过程可以分为静态链接和动态链接)

## 7.1 编译器驱动程序

GNU编译系统，包括编译、汇编等过程，我们通常使用GCC驱动程序进行编译汇编链接。经历了.c到.i到.o的过程。对于不同的.o文件，链接器将不同的.o文件和一些必要的系统目标文件组合起来，构成一个可执行目标文件。

## 7.2静态链接

对于待链接的可重定位目标文件，是由各种不同的代码和数据节组成的。每一节都是一个连续的字节序列，指令在这一节，初始化后的全局变量，在另外一节，而未初始化的变量又在另外一节，会按照变量性质进行分门别类。

可重定位文件的“可重定位”的函数。编译器生成的可重定位文件，是生成地址从0开始的代码和数据节，我们的链接器将代码段和数据节与我们内存位置关联起来，重定位这些代码和数据节。

可重定位文件，就是可重定位目标文件。

共享文件的格式都是一样的，以ELF文件格式为例，只有在ELF文件的某个字段中，表明这是可重定位目标文件、还是可执行目标文件，还是共享目标文件。

## 7.3 目标文件

**·目标文件类型分类**

目标文件可以分为三种，分别是可重定位目标文件、可执行目标文件和共享目标文件。可重定位目标文件是编译器产生的文件，而可执行目标文件是链接器最后产生的目标文件。共享目标文件可以看成是特殊的可重定位目标文件，可以在加载或者运行时被动态地加载进内存并链接。

目标文件的格式一般采用ELF文件格式(读作L F),该文件格式分为ELF头部、代码和数据节、以及描述代码和数据节的节头部表。

**·静态局部变量的理解**

局部变量原先是存储在栈中的，因此当退出其当前作用域，下次重新加载时，加载的值会被重新初始化。

但是如果我们对其加上“static”，存储在全局变量区，因此在重新调用时，会将上次的值保留住，用上次的值去更新。

并且静态局部变量只会初始化一次。

## 7.4 可重定位目标文件

可重定位目标文件由**ELF头和数据节，以及节头部表**。 ELF header、数据节，以及描述数据节的表。

数据节是数目不定，不同的数据节，存放不同类型的数据，初始化后的全局变量，以及不初始化的全局变量，只读文件，已编译好的程序的机器代码。以及其他与程序有关的数据段。

.data中保存已经初始化的全局变量和静态C变量

.bss中保存未初始化的全局变量和静态C变量，或者是初始化为0的全局或静态变量。

局部变量不保存在数据节中，只保存在栈中。

rodata:存放只读数据。

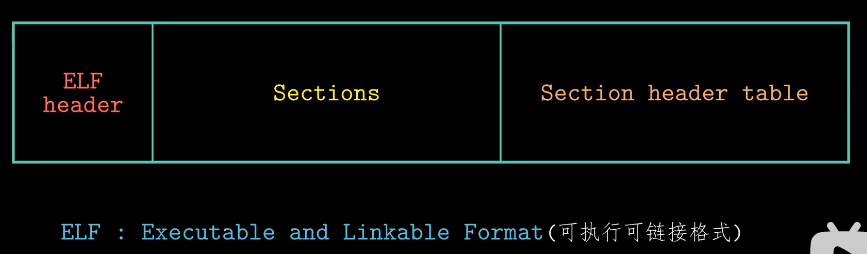
symtab:符号表，符号表是 整个链接过程，是基于符号才能正确完成的。(当某一个可重定位目标文件引用外部的函数或者是变量时，在编写机器代码时，会将引用地址等先置为0，并且在符号表中指出，该地址部分，存在外部函数引用，需要重定位，在linking过程中进行重定位)

## 7.5 符号和符号表

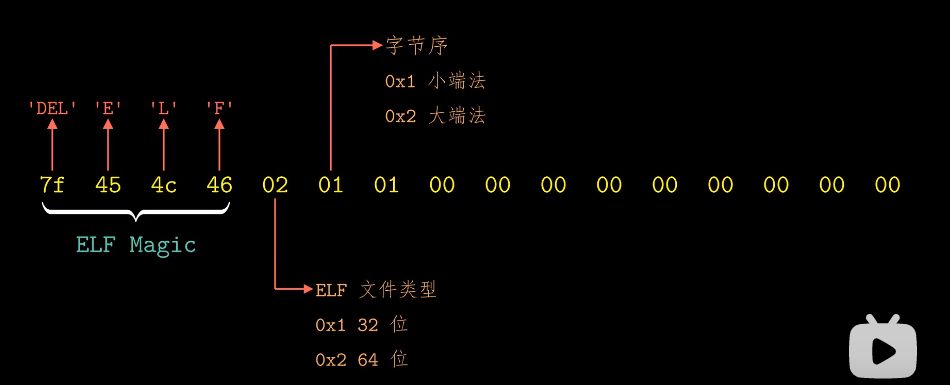
符号表中的符号，和源程序之间会存在对应关系

每个可重定位目标文件模块m中，都会有一个符号表，符号表中会包含模块m定义和引用的符号的信息。在链接器的上下文中，有三种不同的符号。分别是：由模块m定义，并且能够被其他模块引用的全局符号；由其他模块定义，在m中被引用的全局符号；在m中定义和引用的局部符号(限定作用域的全局符号 例如static修饰的全局变量)，注意了在符号表中包含的符号，都是全局变量，是为了统筹各个模块之间的变量定义和引用出现的。

符号表是由汇编器构造的。为链接过程作准备。

**ELF文件格式说明**

ELF头、数据节以及描述数据节的表

 ELFheader开头的16个字节包含了魔数(确认文件类型)

## 7.6符号解析

符号解析可以处理全局变量在不同c文件中的共享使用

当编译器遇到一个不是在当前模块中定义的符号时，他会默认该符号在其他模块中定义。如果链接完毕后，没有在其他模块中找到该符号的定义。则链接失败，会报错。

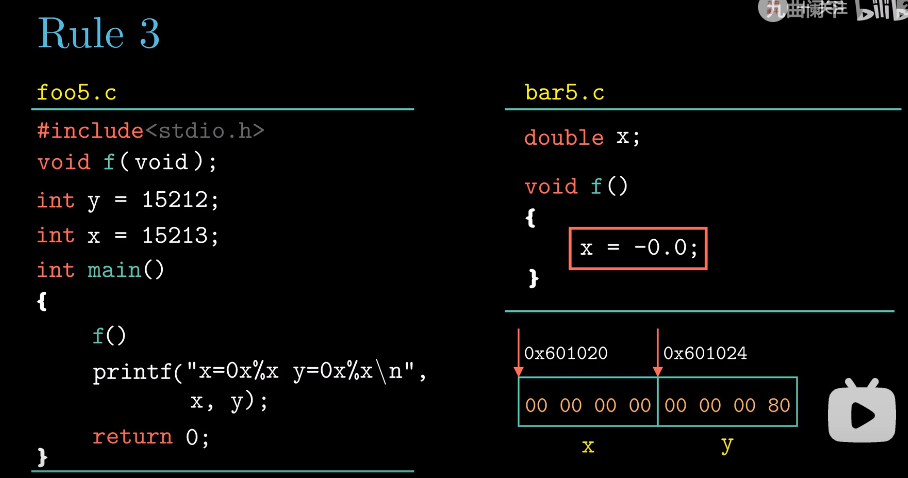
如果是多个可重定位目标文件中定义了同名的全局符号。符号是分为强弱的，分为强符号和弱符号。**强符号包括函数和已经初始化的全局变量**。而弱符号是未初始化的全局变量。强弱信息是隐藏在符号表中的。(强弱主要看变量有无初始化)

当如果在不同的可重定位目标文件中，定义了同名的全局变量，并且都进行了初始化，那么链接器会报错。

链接器不允许多个同名的强符号出现。

但是如果有一个是强符号，另外一个未初始化，那么就是弱符号。链接器会选择强符号。

如果全局变量都未初始化，那么都是弱符号。

 虽然同是定义强符号和弱符号的情况，能通过链接，但是实际使用可能会出现内存越界的情况，不建议不同可重定位目标文件中取相同名字的变量和函数。

如上图所示，当不同模块中，对同一变量名定义的类型不同，赋值起来可能会存在地址溢出的错误，导致对其他变量的错误赋值。

当编译器在翻译某个模块是，遇到了一个弱全局符号x。编译器不清楚该弱全局符号在外部模块中，是否有强全局符号的定义。所以会将该全局符号分配成COMMON，把决定权留给链接器。

但是当某一个全局符号被赋初值了，那就认为是强全局符号，那么编译器就知道，必须引用该全局符号，就可以将该全局符号分配成bss。

### 7.6.2 与静态库链接

在linux中，静态库以特殊的文件格式存放在磁盘当中，我们有理由相信，在window系统下，也是存放在磁盘当中。

静态库中，存在许多可能会使用到的目标模块，比如说printf.o等，链接器会复制静态库中被引用的目标模块，并一起打包生成可执行文件。

相关的函数可以被独立编译成独立的目标模块，然后封装成一个单独的静态库文件。

### 7.6.3链接器如何使用静态库来解析引用

当我们需要对可重定位文件和库文件进行链接时，链接过程如下所示：

假设先链接main.o文件，我们在main.o文件中会发现一些未定义的符号，比如说printf.o(这在静态库中)，因此我们会先假定这些符号都存在，并存放在一个集合中，除此之外，还会对全局变量进行集合的存放。接着去链接库文件，会在链接库中，去寻找当前假定存在的符号。如果链接到最后，发现还存在假定存在的符号，则报错。

但是如果这样进行链接，对于可重定位的目标文件和库文件的输入先后顺序，就有很高的要求，通常需要先输入main.o 再输入库文件。

链接器会将引用了，但是尚未定义的全局变量，存放在一个单独的集合当中。当里面的全局变量不断被定义后，逐渐清空该集合。

## 7.7 重定位

在完成符号解析后，链接去需要对编译器生成的代码和数据节进行重定位。与内存地址相互关联。

**重定位操作：链接器合并输入模块，并为每个模块重新分配运行时的地址。**

**因为当模块引用外部变量时，对应的机器代码的引用地址，可能是0，具体的地址还没有确定，具体地址的确定是由链接器来确定的。**

重定位分为两步：分别是重定位节和符号定义，第二步是重定位节中的符号引用

**·重定位节和符号定义**

重定位实现对链接得到的新的聚合节，赋予新的运行内存地址。当重定位完成后，程序中的每条指令和全局变量，都有唯一运行的内存地址(这都是逻辑地址吧，还是线性地址)。

经过这一步，每条指令和全局变量都有唯一运行的内存地址，接下来就需要用正确的内存地址，去更新原先各模块中存在的“暂时地址”。也就是重定位节中的符号引用。

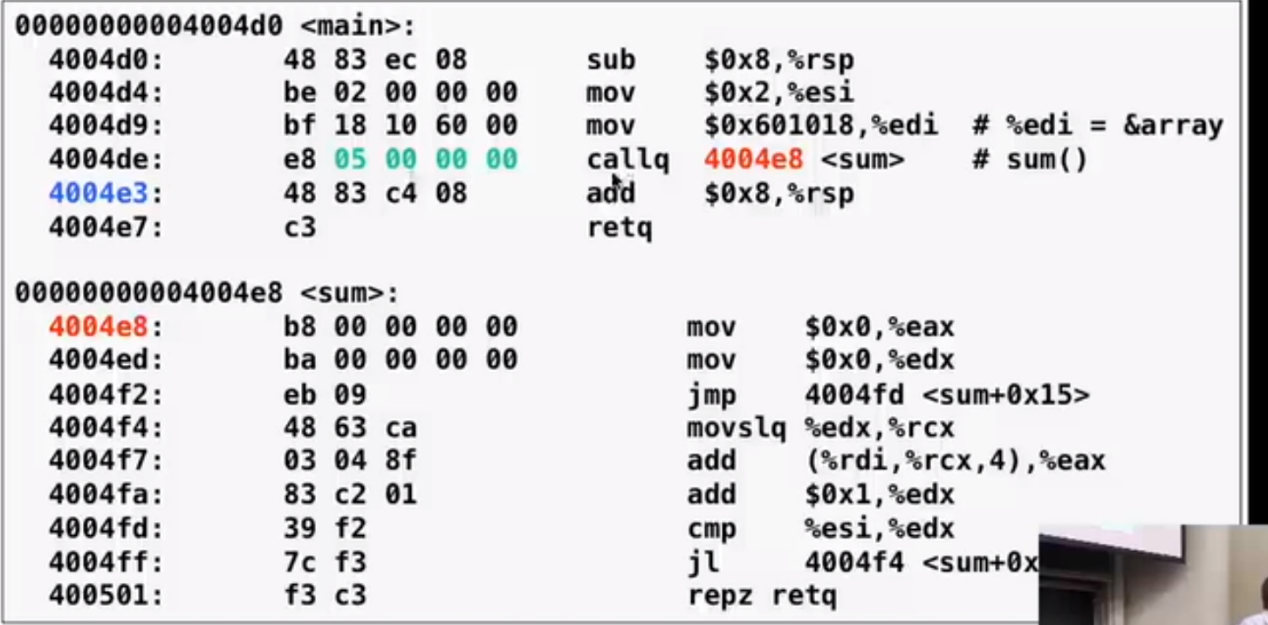
**·重定位节中的符号引用**

在这一步过程中，链接器修改代码节和数据节中，对每个符号的引用，使得他们指向正确的地址。这一部分依赖于可重定位目标模块中，称为重定位条目的数据结构。

在重定位之前，目标模块中的函数地址，只是它在模块中的偏移量。(有个疑问，当某一个.c文件中，调用了其他c文件的函数，那么在进行call调用时，汇编该怎么写？)

### 7.7.1 重定位条目

当汇编器产生可重定位目标文件时，对于一些外部定义的函数或者全局变量，汇编器并不知道其存放的位置。因此，当汇编器遇到对最终位置未知的目标引用时，会生成重定位条目，该条目会告诉链接器在合并成可执行文件时，如何修改这个引用。



## 7.8 可执行目标文件

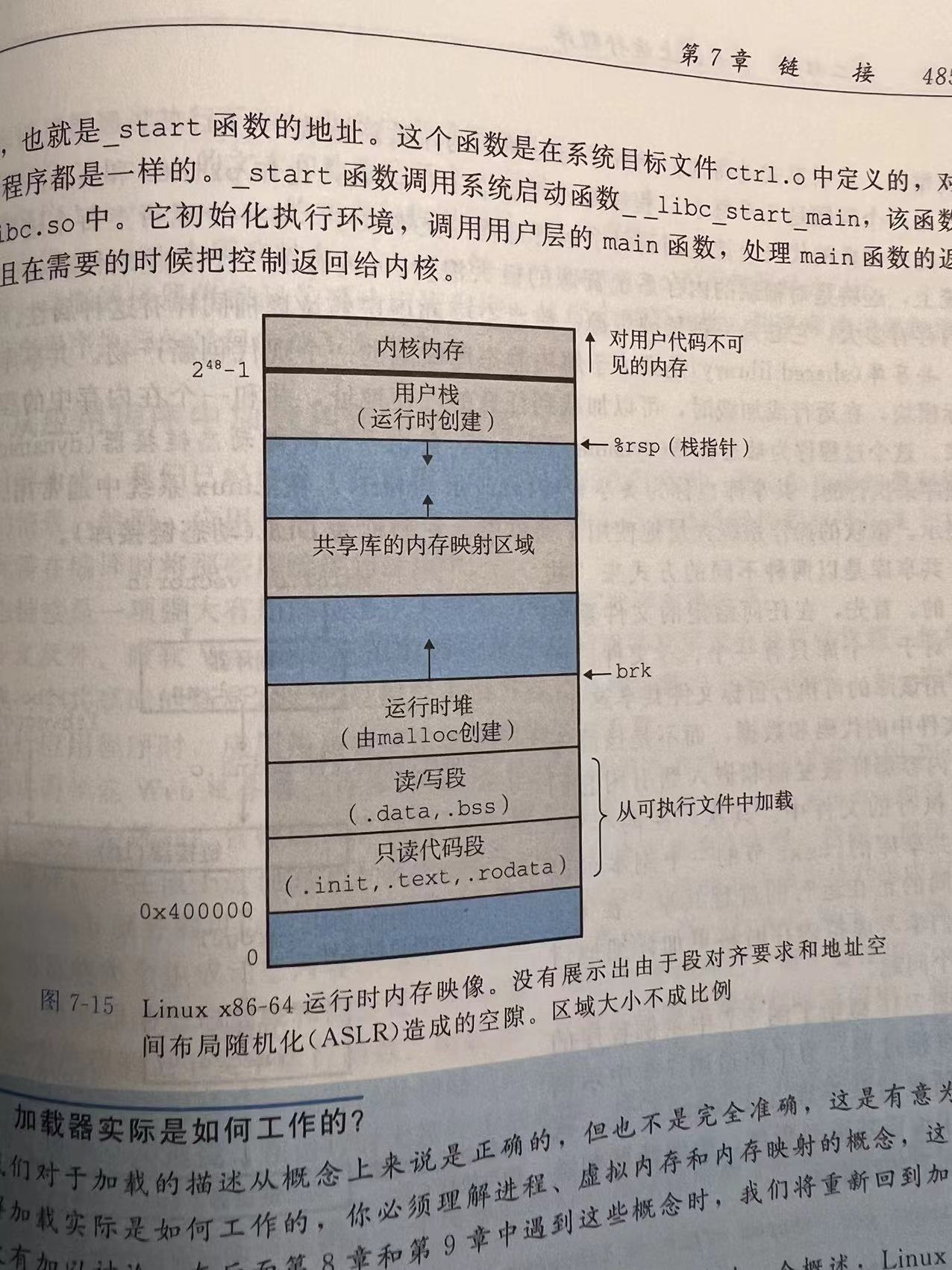
## 7.9 加载可执行目标文件

每个Linux程序都有一个运行的内存映像。我们可以认为，每个程序占据全部的内存。

我们的程序，一般来说，是从0x400000处开始。

0x400000向上，是只读代码块和读写块。是从可执行目标文件中加载出来的。然后是运行时产生的堆。堆之后，是为共享库提供的内存映射区域。再之后是用户栈。栈总是从高地址向低地址扩展，最高地址为2^48-1。地址为2^48及以上的部分，是为内核中的代码和数据保留的，是操作系统驻留在内存中的部分。

但是栈顶的位置不一定是2^48-1,只是最大地址可以达到这个大小。

 并且由于代码段和数据段之间有对其要求，所以二者之间是有间隙的。并且根据安全要求，程序运行时，这些区域的地址都会变，但是相对位置不会发生变化。

## 7.10 动态链接共享库

静态库有些缺点，静态库和所有的软件一样，需要定期维护和更新。对于常见的共享文件，如果每个可执行性文件都要copy一份，是对内存资源的巨大浪费。

为了解决静态库的缺陷，操作系统提供了一种共享库的技术。

共享库在运行或加载时，可以被加载到任意的内存地址，并且能和在内存中的程序链接起来，这个过程称为动态链接。

# 第八章 异常控制流

## 8.1 异常

异常控制流的分类：

中断、陷阱、故障、终止

中断是异步的，而陷入、故障和终止，都是同步的。这里同步和异步的概念，主要是指异常来源是来自于CPU内部还是CPU外部，CPU处理指令产生的结果。

**中断**

CPU会在执行完当前的指令后，再去响应中断，响应完成后，再去执行下一条指令

**陷阱**

陷阱是一种故意触发的异常，他是执行一条指令的结果。陷阱最重要的用途，是为用户程序和操作系统之间，提供调用接口，称为系统调用。

**故障**

故障是由错误情况引起的，不过是有可能被系统修复的。发生故障后，修复完成后，操作系统仍然处理发生故障的那条指令。当错误无法修复，则进行终止。一个经典的故障例子，就是缺页。

**终止**

不可恢复的异常。

当处理器检测到异常事件发生时，处理器会从执行应用程序，触发异常们根据异常表的表目，切换到对应的异常处理程序，根据异常类型，决定是否返回。

系统为每种异常类型都分配了异常编号。根据异常编号，从异常表中，检索对应的异常处理程序，处理异常。

异常处理程序是运行在内核态的，所以他对所有的系统资源都具有访问权限

异常处理过程，类似于过程调用，因此也会存在入栈出栈的操作，并且部分上下文信息也会压入栈中。是否返回到原先出现异常的那条指令，或者执行下一条指令，则由异常类型和处理结果决定。(缺页故障会返回出现异常的那条指令，而中断则会执行对应的下一条指令)。

如果控制从用户程序转移到内核，则当前进程的上下文，都会被压入内核栈当中，而不是用户栈当中。(内核也不是全部常驻内存的)。

## 8.2 进程

### 8.2.5 上下文切换

上下文是由程序或者说进程，正确运行所需的状态组成的。

上下文是内核重启一个被抢占的进程，所需的状态。他由一些对象值组成，包括寄存器的值、程序计数器、用户栈、状态寄存器、和各种内核数据结构，包括描述地址空间的页表，包含当前进程信息的进程表，以及包含进程已打开文件的信息的文件表。

当发生上下文切换时，环境信息是保存在内核栈当中对吧？(是的吧)

## 8.4 进程控制

### 8.4.1 获取进程ID

每个进程，都有唯一一个的正数(非0)的进程ID。

(当我运行hello world函数时，是否也有一个进程ID?)

### 8.4.2 创建和终止进程

# 第九章 虚拟内存

早期PC使用物理寻址，CPU直接获得要访问的程序代码或者是数据的内存的真实地址。

但是后来，CPU的PC获取的是虚拟地址，通过MMU,将虚拟地址映射到内存真实的物理地址。

虚拟内存的好处。第一可以将内存看成是存储在磁盘上的地址空间的高速缓存。第二，可以保护不同进程的地址空间不被其他进程破坏。第三：他为每个地址空间提供了一致的地址空间，从而简化内存管理。

对于以下话的理解：“从N = 2^n个地址的地址空间中，生成一个虚拟地址”

答：虚拟地址空间是一个集合，从0000000 到11111111 ，共同组成虚拟地址空间。而对其中任意一个，都是我们可能会生成的虚拟地址，比如说 0111\_1110 。虚拟地址的前a位和后n-1-a位，有其特殊的含义。

虚拟内存被组织为一个由存放在磁盘丧的N个连续的字节大小的单元组成的数组。在磁盘上，被分割成页。每个页存在两种状态，是否被缓存到内存当中。这种缓存也可以认为是一种映射关系。

这种映射关系，需要一张页表存放，每个进程都会有一张页表。页表的处理单元是页。进程是按照页分的，而存放到内存当中时，也是按照页存放的。页的存放是不连续的。

缓存命中：当上一次存储系统向下一级存储系统查找某一数据时，如果查到了，则称为缓存命中。如果没查到，则称为缓存不命中。缓存不命中后，则需要从下一级的存储系统中继续向下查找。

由于DRAM的缓存不命中的代价很高，因此虚拟页的大小是比较大的。并且页的替换算法也是比较精密的。

## 9.3 虚拟内存作为缓存的工具

9.3.2 页表

虚拟内存系统必须通过某种方法，判断一个虚拟页是否缓存在DRAM中的某个地方。如果是，系统还必须确定这个虚拟页存放在哪个物理页中，如果不命中，系统还必须判断该虚拟页应该存放在磁盘的哪个位置，并在物理内存中，选择一个牺牲页，并将虚拟页从磁盘幅值到DRAM中，替换这个牺牲页。

这种替换是由操作系统管理的，需要MMU中的地址翻译硬件，和一个存放在物理内存中的叫做页表的数据结构。

页表将虚拟页映射到物理页，每次地址翻译时，都会将一个虚拟地址转换成物理地址时，都需要读取页表。

缓存不命中的现象叫做缺页。当CPU需要读取某一个虚拟页VP3的某一个数据(是通过偏移量算出来的)，那么MMU在页表中找，发现该页VP3没有被映射到内存中，因此会触发一个缺页异常。该缺页异常调用缺页异常中的缺页异常处理程序。在内存当中选择某物理页，用来存放VP3的内容，比如说我们选用VP4,如果VP4已经被修改过了，则会将VP4的内容重新读回磁盘中。

局部性分为时间局部性和空间局部性

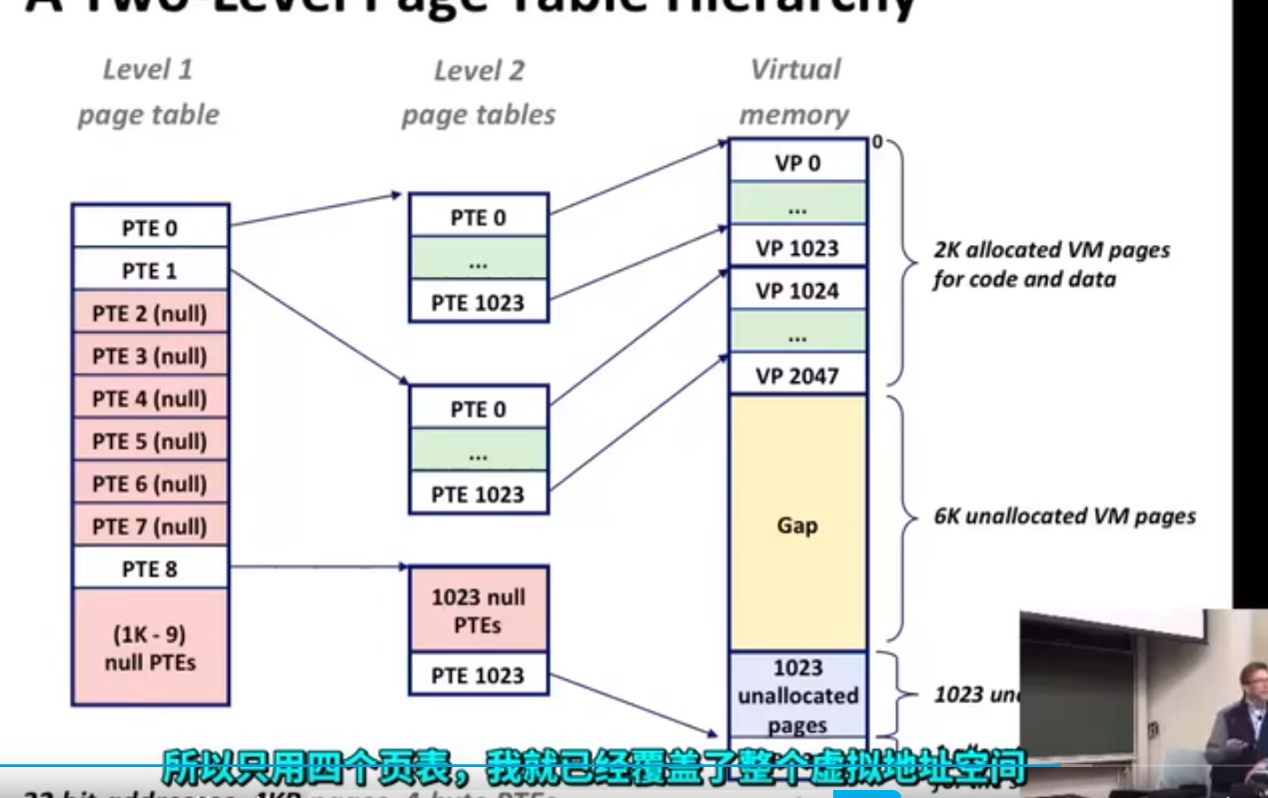
时间局部性是指：一条指令或者是一个数据的多次访问，集中在一个较短的时间中。空间局部性是指：当前指令和邻近访问的几条指令，都集中在一个较小区域内。因此可以保证在进程运行时，这种页面交换不会频繁出现。

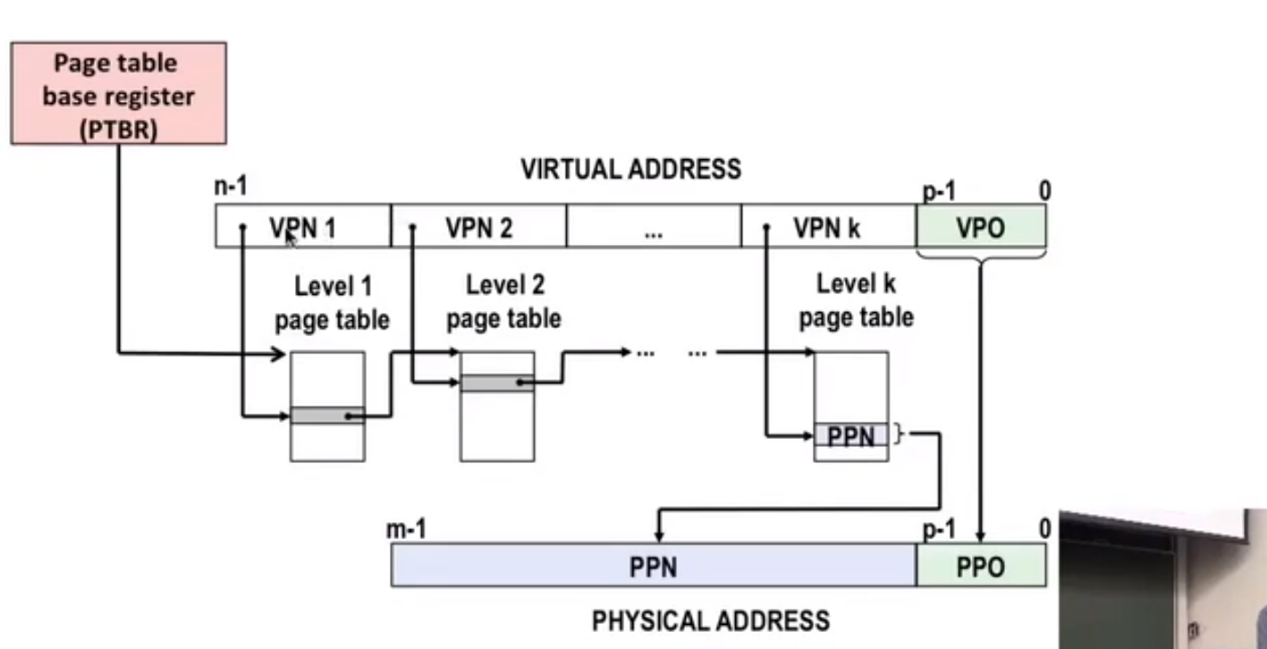
## 9.4 虚拟内存作为内存管理的工具

简化内存管理，可以保护内存。

虚拟内存的VPN ，可以定位到PPN





多级页表如何节省内存空间

Execve不添加新进程，只是在当前进程上，添加一段新程序。

# 第十章 系统级IO

# 第十一章 网络编程

# 第十二章 并发编程

# 存疑

1.

2.机器指令 机器周期

3.寻址方式有哪些

4.时钟周期、机器周期和指令周期

时钟周期好理解，由晶振提供的定时。

指令周期，读取并执行一条指令所需要的时间，通常由好几个机器周期构成。

机器周期：读一条指令最少的时间，一般为12倍的时钟周期。

5.逻辑地址、线性地址、物理地址

逻辑地址好理解。逻辑地址是程序产生的，和段相关的段内偏移(引入段的概念)

线性地址是段内偏移地址+段基址，就成为线性地址。

如果启用了分页机制，则线性地址会再映射到物理地址上。