

计算机系统基础I复习大纲



题型

- 选择题：20分
- 分析题：30分
- 实验题：15分
- 综合应用题：35分



CPU时钟周期

机器内部主时钟脉冲信号的宽度，它是CPU工作的最小时间单位。

主频

主频也叫**时钟频率**，用来表示微处理器的运行速度，主频越高表明微处理器运行越快，主频的单位是MHz。

每条指令平均时钟周期数CPI: Cycles Per Instruction

$CPI = \text{执行程序所需的时钟周期数} / IC$

IC: 所执行的指令条数

执行一个程序所需的CPU时间

= 执行程序所需的时钟周期数 $\times$ 时钟周期时间

= 执行程序所需的时钟周期数 / 时钟频率

= $IC \times CPI / \text{时钟频率}$

CPU的性能取决于三个参数：**时钟频率**，**CPI**，**IC**

单选题

1分

计算机系统

设置

假定处理器 P 的主频为 3GHz。CPI 为 1.5，P 运行某个程序的执行时间为 10s，该程序执行的时钟周期数和指令数分别是 (B)。

1. 时钟周期数 = 主频 $\times$ 执行时间

$$= 3 \times 10^9 \times 10$$

$$= 3 \times 10^{10}$$

$$3\text{GHz} = 3 \times 10^9 \text{ Hz}$$

(每秒有 3×10^9 个时钟周期)

A

B

C

D

~ ~ ~ ~ ~

$3 \times 10^9, 2 \times 10^{10}$

$3 \times 10^{10}, 2 \times 10^{10}$

$3 \times 10^{10}, 2 \times 10^9$

CPI = 1.5

每条指令平均需 1.5 个时钟周期

2. 指令数

$$B = \frac{\text{时钟周期数}}{CPI} = \frac{3 \times 10^{10}}{1.5} = 2 \times 10^{10}$$

P: 执行时间 10s

提交



(1) 指令集架构ISA

提供了对实际处理器硬件的抽象。使用这个抽象，机器代码程序表现得就好像运行在一个一次只执行一条指令的处理器上。

(2) 虚拟内存

提供程序存储器的抽象，包括主存和磁盘I/O设备

(3) 文件

提供了对I/O设备的抽象

(4) 进程

提供正在运行的程序的抽象。包括对处理器、主存和I/O设备的抽象表示

(5) 虚拟机

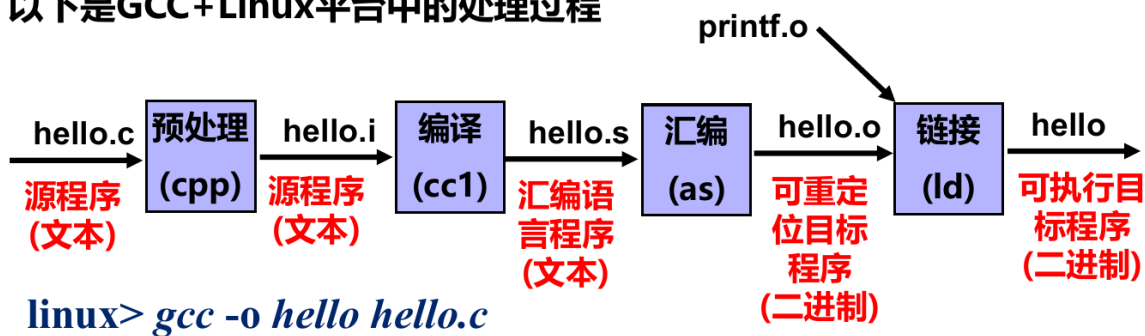
提供了对整个计算机抽象



可执行文件的生成

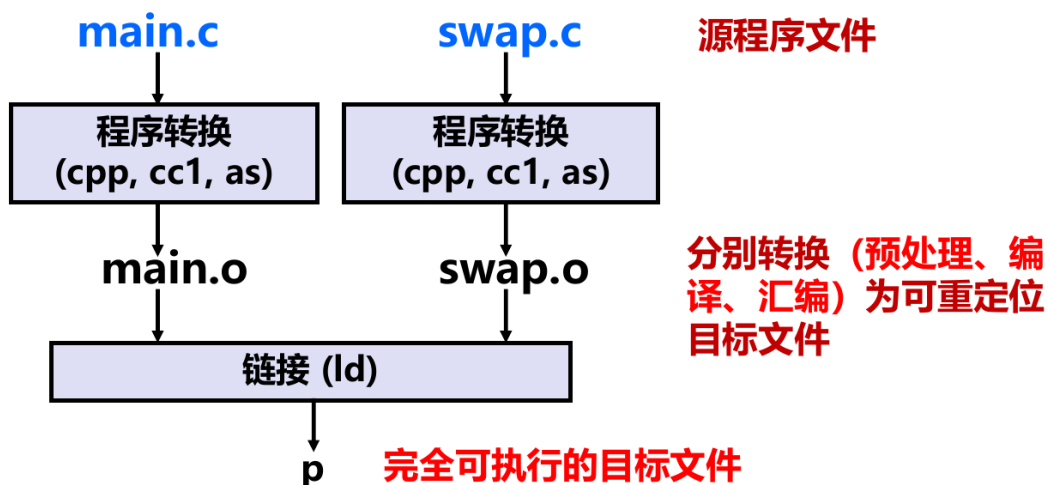
高级语言程序开发过程

以下是GCC+Linux平台中的处理过程



- 使用GCC编译器编译并链接生成可执行程序P:
 - \$ gcc -o p main.c swap.c
 - \$./p

GCC
编译器的
静态
链接
过程



计算机中数据的编码

C声明		字节数	
有符号	无符号	32位	64位
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

基本C数据类型的典型大小(以字节为单位)。
分配的字节数受程序是如何编译的影响而变化。



在几乎所有的机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。

例：unsigned int x=1000; ^{4字节} &i的值: 0x500

32位: ^{高位} 0000 0000 0000 0000 0000 0011 1110 1000 ^{低位} (0x000003e8)

0x503	00
0x502	00
0x501	03
0x500	e8

高位

低位

increasing
byte
address

0x503	e8
0x502	03
0x501	00
0x500	00

小端方式

低位字节存放在低地址单元
高位字节存放在相邻的高地址单元

大端方式

高位字节存放在低地址单元
低位字节存放在相邻的高地址单元

单选题 1分

设置

设intel机器中，有一个int型变量x的地址为0xFFFFC000, x=0x12345678, 则内存单元0xFFFFC001中存放的内容的十六进制表示为 (C)

A 0x12

B 0x34

C 0x56

D 0x78

12	0003	↓ 低
34	0002	
56	0001	
78	0000	

提交



unsigned 都为正
 n位无符号数, 其可表示的值范围为: $0 \sim +2^n - 1$

C数据类型	位数	最小值	最大值
unsigned char	8	0	255
unsigned short	16	0	0-65535
unsigned int	32	0	4294967295
unsigned long	64	0	18446744 073709551615



补码:

- 补码表示的有符号数, 对于正数来说同原码、反码一样, 但负数的数值位部分为其绝对值按位取反后末位加1所得。

正数补码 = 原码
负数补码 = 正数码取反 + 1

例如: 8位机器数, 则:

+23的补码为 **00010111**

-23的补码为 **11101001**

结论1: 一个负数的补码等于对应正数补码的“各位取反、末位加1”



求补码另一种方法

结论2：一个负数的补码等于模（ 2^n ）减该负数的绝对值。

$$\begin{array}{r}
 -23 \text{ 的补码为 } 100000000 \\
 \quad \quad \quad -00010111 \\
 \hline
 \quad \quad \quad 11101001
 \end{array}$$



例：

① 先求 +1000 的 32 位
 0000 0000 0000 0000 0000 0011 1110 1000
 ② 取反
 int x=-1000; 1111 1111 1111 1111 1111 1100 0001 0111
 ③ 加 1
 32 位: 1111 1111 1111 1111 1111 1100 0001 1000
 (0xFFFFFA18) rsp=0xf5dd1550

FF
FF
FA
18

高 12 位
↓
低

GDB x/4xb 0xf5dd1550

x/1xw 0xf5dd1550

GDB x/4xb \$rsp

0xf5dd1550 :18 fa ff ff

0xf5dd1550 :ffffffa18

b表示单字节，h表示双字节，w表示四字节，g表示八字节

x表示十六进制，d表示有符号十进制，u表示无符号十进制



n位补码所能表示的真值范围为： $-2^{n-1} \sim +2^{n-1} - 1$

C数据类型	位数	$TMin_w$ 最小值	$TMax_w$ 最大值
char	8 = 1字节	-128	127
short	16 = 2字节	-32768	32767
int	32 = 4字节	-2 147 483 648	2 147 483 647
long	64	-9223 372 036 854 775 808	9223 372 036 854 775 807

C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）



无符号数和有符号数的转换

考虑以下C代码：

```

1 int x = -1;
2 unsigned u = 2147483648;
3 printf("x = %u = %d\n", x, x);
4 printf("u = %u = %d\n", u, u);

```

在机器上运行上述代码时，它的输出结果是什么？为什么？

$x = 4294967295 = -1$

$u = 2147483648 = -2147483648$

◆ 2^{31} 的无符号数表示为 "100...0"，被解释为32位带符号整数时，其值为最小负数： $-2^{32-1} = -2^{31} = -2\,147\,483\,648$ 。

处理同样字长的有符号数和无符号数之间相互转换的一般规则是：数值可能会改变，但是位模式不变；

强制类型转换的结果保持位值不变，只是改变了解释这些位的方式



考虑以下C代码：

C语言隐式类型转换

```
int tx; (有) 32位全1
unsigned ux=0xffffffff; (无)
tx=ux; /*Cast to signed */ ☆ 隐式转换, 无符号附给有符号
printf( "tx=%d , ux=%u\n" , tx , ux);
它的输出结果是什么?
```

tx = -1, ux = 4294967295

都是32位全1 但 { tx 有符号 = -1
ux 无符号 = 4294967295
C语言: 位模式被
只解释为十进制



浮点数表示

IEEE浮点标准 $(-1)^s M 2^E$

– Sign 符号位 **s** 代表这个数的正负

– Significand 尾数 **M** 定点小数

– Exponent 指数 **E** 整数

$(-1)^0 1.011101 \times 2^3$

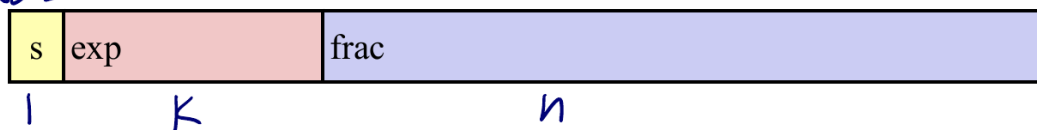
• 编码

– MSB **s** 是符号位 **s**

– exp 编码 **E** (but is not equal to E)

– frac 编码 **M** (but is not equal to M)

符号位





C语言中的浮点变量

$$\text{Bias} = 2^{k-1} - 1 \quad (k \text{ 为 } \text{exp} \text{ 位数})$$

$$\text{Bias}_{\text{float}} = 2^7 - 1 = 127$$

$$\text{Bias}_{\text{double}} = 2^8 - 1 = 1023$$

- 单精度: 32 bits float

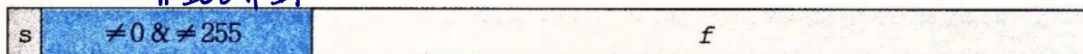


- 双精度: 64 bits double

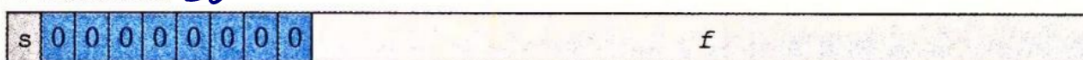


根据 exp 的值, 被编码的值可以分成三种不同的情况

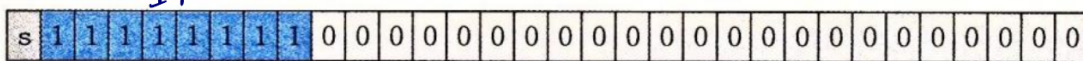
- 规格化的 $\neq 0$ 且 $\neq 255$



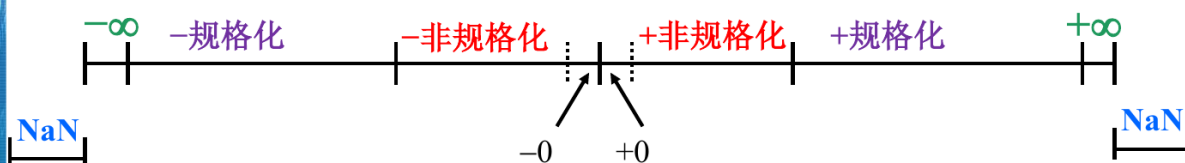
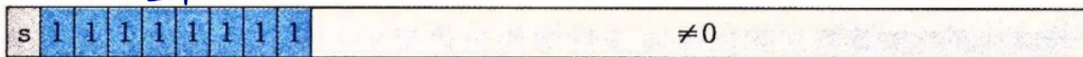
- 非规格化的 全 0



- 无穷大 全 1



- NaN 全 1



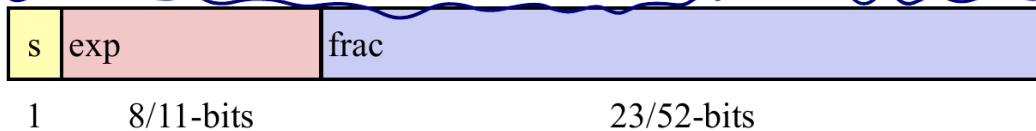


Normalized numbers (规格化数)

计算机系统基础I

规格化数: $\pm 1.xxxxxxxx_{two} \times 2^{Exponent}$

$$v = (-1)^s M 2^E$$



- $exp \neq 000...0$ and $exp \neq 111...1$ (非0也非1)

- $exp = E + Bias$

– exp : 无符号数

– $Bias = 2^{k-1} - 1$, $k=8$ 或 11 是 exp 的位数

- 单精度: 127 ($exp: 1...254$, $E: -126...127$)

- 双精度: 1023 ($exp: 1...2046$, $E: -1022...1023$)

- $M = 1.xxx...x_2$ $xxx...x$: $frac$

– Minimum when $frac=000...0$ ($M = 1.0$)

– Maximum when $frac=111...1$ ($M = 2.0 - \epsilon$)

规定: 规格化尾数最高位总是1, 所以隐含表示, 省1位
单精度: $frac$ 用23位, 但能精度表示24位有效数字
双精度: $frac$ 用52位, 但能表示53位有效数字
机不存1, 省下给 $frac$ 用一位
实际尾数 $m = 1.xxx$ 存储 $frac = xxx$



Normalized numbers (规格化数)

计算机系统基础I

规格化数: $\pm 1.xxxxxxxx_{two} \times 2^{Exponent}$



exp范围

8/11-bits

指数M把小数加回, 小数 $frac$

23/52-bits

$E = exp - Bias$

单精度: $(-1)^s \times (1 + frac) \times 2^{(exp-127)}$

双精度: $(-1)^s \times (1 + frac) \times 2^{(exp-1023)}$

32位浮点数格式的规格化数的表示范围

$$v = (-1)^s M 2^E$$



规格化数举例

- Value: float $F = 15213.0$;

$$15213_{10} = 11101101101101_2$$

$$= 1.1101101101101_2 \times 2^{13}$$

$$v = (-1)^s M 2^E$$

$$E = \text{Exp} - \text{Bias}$$

- Significand

$$M = 1.1101101101101_2$$

$$\text{frac} = 11011011011010000000000_2$$

- Exponent

$$E = 13$$

$$\text{Bias} = 127$$

$$\text{Exp} = 140 = 10001100_2$$

① 先2进制然后格式化成1.x.x.x

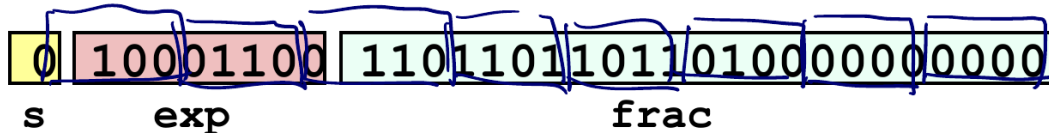
② 根据格式化成: E, frac

③ 算 $\text{exp} = E + \text{bias} \rightarrow$ 二进制

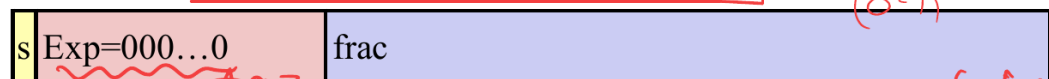
④ frac 补到23位 (左补0)

⑤ 拼接 s Exp frac
4位一分 然后得十六进制

- Result: (1+8+23)



非规格化数: $\pm 0.xxxxxxxx_{\text{two}} \times 2^{\text{Exponent}}$ (补最低位到0 调零)



1 8/11-bits

23/52-bits

- Condition: $\text{exp} = 000...0$

- $E = 1 - \text{Bias}$ 注意规格化是 $E = \text{exp} - \text{Bias}$

$$v = (-1)^s M 2^E$$

$$E = 1 - \text{Bias}$$

- 单精度: $E: -126$

- 双精度: $E: -1022$

- $M = 0.xxxx...x$ 规格化 $M = 1.x \times (\text{隐含} 1)$ $xxx...x_2$: bits of **frac**

- $\text{exp} = 000...0$, $\text{frac} = 000...0$

- 代表数值 0, 不区分 +0 和 -0

- $\text{exp} = 000...0$, $\text{frac} \neq 000...0$

- 接近 0.0 的小数

eg 0.000000000000000000000001

$s = 0$ $\text{frac} = 0.0...1$

$M = 0.0...1 = 2^{-23}$

$E = -126$

$v = 0.000...01 \times 2^{-126}$

$= 2^{-23} \times 2^{-126}$

$= 2^{-149}$

In FP, 除数为0的结果是 $\pm\infty$, 不是溢出异常. (整数除0为异常)

• 可以利用 $+\infty/-\infty$ 作比较。例如: $X/0 > Y$ 可作为有效比较

- $\text{exp} = 111\dots 1$ ^{exp 全1 特殊值} $\left\{ \begin{array}{l} \text{frac 全0} \rightarrow \pm\infty \\ \text{frac 非0} \rightarrow \text{NaN 非数} \end{array} \right.$
- 情况1: $\text{exp} = 111\dots 1, \text{frac} = 000\dots 0$

– 代表 ∞

– E.g., $1.0/0.0 = -1.0/-0.0 = +\infty, 1.0/-0.0 = -\infty$

- 情况2: $\text{exp} = 111\dots 1, \text{frac} \neq 000\dots 0$

– Not-a-Number (NaN) ^{不等于任何数, 自身, 参与任何运算仍NaN}

– E.g., $\text{sqrt}(-1), \infty - \infty, \infty \times 0$ ^{数浮上极限}

X-86指令概述

指令存放

多字节操作数连续存放, 顺序依据小端法规则 (Little Endian),

即: 低位字节存放在低地址单元, 高位字节存放在相邻的高地址

单元。

^{下低指令}
40110a: ^{大字节} b8 78 56 34 12
40110f: ^{小字节} bb 78 56 34 12

操作码
告诉CPU这是mov指令
目标是eax/ebx

mov \$0x12345678,%eax (4字节版)
mov \$0x12345678,%ebx

[↑]
把数字 0x12345678 装到寄存器%eax

每个mov指令占5字节 | 1字节操作码 + 4字节立即数

地址

机器码(5字节)

汇编指令(人类读)



- 寻址方式

- 根据指令给定信息得到操作数

- 操作数所在的位置 *寻址方式*

- ① – 指令中：立即寻址

eg
\$0x10

- ② – 寄存器中：寄存器寻址

%rax

- ③ – 存储单元中（属于存储器操作数，按字节编址）：

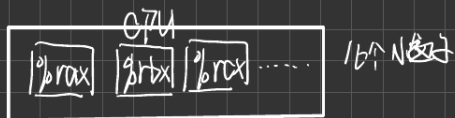
在内存里

(%rax), 8(%rax), %rdx(4)

- ④ – 端口中

寄存器

CPU 内部小盒子, 用来临时存数据



每个盒子有一个名字: 比如 %rax

能装 8 个字节的数据

eg %rax 装着: 0x123456789ABCDEF (16 位十六进制)

CPU 做运算时只能操作寄存器里的数据 | 所以要经常从内存搬进寄存器, 算完再搬回内存

% 是什么: AT&T 汇编语法的标记, 告诉这是寄存器名, 区别于普通数

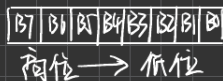
eg: %rax → 寄存器 rax

rax → 不规范写法

0x100 内存地址或数字

#0x100 → 立即数 (常数)

一个盒子里



不同名字访问不同数据 (eg: 0x123456789ABCDEF)

%rax → 全部 = 0x123456789ABCDEF (8 字节)

%eax → 低 4 字节 = 0x9ABCDEF (低 4 字节)

%ax → 低 2 字节 = 0xCDEF (低 2 字节)

特殊 { %ah → 第 2 字节 = 0xCD (第 2 字节)

%al → 最低字节 = 0xEF (最低字节)

指令 命令 CPU

eg: movq %rbx, %rax
动词 源 目标

= 把 %rbx 值复制到 %rax 里
左流向右 (ATT)

四、看到指令怎么“读”

第一步：找动词

Code

```
movq %rbx, %rax    ← mov (搬动)    + q (8字节)
addq %rbx, %rax    ← add (加)       + q (8字节)
subq %rbx, %rax    ← sub (减)       + q (8字节)
```

decq 减 1

动词带后缀 (b/w/l/q) 告诉你“操作几字节”:

- b = 1字节
- w = 2字节
- l = 4字节
- q = 8字节



第二步：找操作数

操作数有 3 种形式:

写法	是什么	例子
%xxx	寄存器 (取盒子中的值)	%rax
\$xxx	立即数 (就是这个常数)	\$5、\$0x100
(...) 或 数字+(...)	内存 (去内存某地址取值)	(%rax)、8(%rax)

第三步：执行

按动词把事情做了。

eg: movq \$5 %rax

搬 8 字节, 立即数 5 装进 %rax 寄存器

movq (%rbx) %rax

去内存里 rbx 指向地方取值

把 rbx 当成地址去内存那个地址读 8 字节 放到 rax

addq %rbx %rax

把 rbx 加到 rax 上 (rbx 不变 rax 变)



x86-64 整数寄存器

计算机系统基础I

16个通用整数寄存器

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp
%rip	%eip

寄存器
不能直接
访问

寄存器
(地址指针)

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d



计算机系统基础I

general purpose

%eax	%ax	%ah	%al
%ecx	%cx	%ch	%cl
%edx	%dx	%dh	%dl
%ebx	%bx	%bh	%bl
%esi	%si		
%edi	%di		
%esp	%sp		
%ebp	%bp		
%eip	%ip		

accumulator
(mostly obsolete)

counter

data

base

source
index

destination
index

stack
pointer

base
pointer

16-bit virtual registers
(backwards compatibility)



寄存器操作数寻址

计算机系统基础I

$$\text{通用公式 } EA = Imm + R[r_b] + R[r_i] \times S$$

Imm: 立即数偏移(常数)

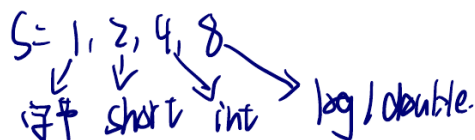
S: 比例因子 1, 2, 4, 8, -

r_b : 基址寄存器

r_i : 变址寄存器

写法	有效地址	寻址方式
Imm	$M[Imm]$	绝对寻址
(r_a)	$M[R[r_a]]$	间接寻址
$Imm(r_b)$	$M[Imm + R[r_b]]$	(基址 + 偏移量) 寻址
(r_b, r_i)	$M[R[r_b] + R[r_i]]$	变址寻址
$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	变址寻址
$(, r_i, S)$	$M[R[r_i] \cdot S]$	比例变址寻址
$Imm(, r_i, S)$	$M[Imm + R[r_i] \cdot S]$	比例变址寻址
(r_b, r_i, S)	$M[R[r_b] + R[r_i] \cdot S]$	比例变址寻址
$Imm(r_b, r_i, S)$	$M[Imm + R[r_b] + R[r_i] \cdot S]$	比例变址寻址

都是通用寻址方式



寻址方式举例

寄存器是立即数直接取常数值本身

不特指寄存器也不特指内存(0xb4, 9, rax)

寄存器是去内存找值

计算机系统基础I

寄存器是 $\rightarrow$ 寄存器寻址
内存是 $\rightarrow$ 绝对寻址

地址	值	寄存器	值
0x100	0xff	rax	0x100
0x104	0xab	rcx	0x1
0x108	0x13	rdx	0x3
0x10c	0x11		

寄存器寻址 + 寄存器内找 X

操作数	值	操作数	值
%rax (寄存器寻址)	0x100	9(%rax, %rdx)	0x11
0x104 (绝对寻址)	0xab	260(%rcx, %rdx)	0x13
\$0x108 (立即寻址)	0x108	0xFC(, %rcx, 4)	0xff
(%rax) (寄存器寻址)	0xff	$= M[0xFC + 0x1 \times 4] = M[0xb4] = 0xff$	
4(%rax) (寄存器寻址)	0xab	$= M[0xb4 + 0x1 \times 4] = M[0xb8] = 0xab$	
		9(%rax, %rdx)	0x11
		$= M[9 + 0x100 + 0x3] = M[0x10c] = 0x11$	

rax 地址

$M[4 \times 0xb4] = M[0xb8] = 0xab$
 $= M[R[rax]] = M[0xb4] = 0xff$



1 假设下面的值存放在指定的内存地址和寄存器中：

地址	值
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

寄存器	值
%rax	0x100
%rcx	1
%rdx	0x3

包括其他指令，如LEA，逻辑运算、移位指令等

指令	目的	值
addq %rcx, (%rax)	$R[rcx] = 0x100$ 内存地址: 0x100	$0x1 + M[0x100] = 0x1 + 0xFF = 0x100$
subq %rdx, 8(%rax)	$M[8 + 0x100] = M[0x108] = 0xAB$ 内存地址: 0x108	$0x100 - 0x3 = 0x9D$
imulq \$16, (%rax, %rdx, 8)	$M[0x100 + 8 \times 0x3] = M[0x118] = 0x11$ 内存地址: 0x118	$0x100 \times 16 = 0x1000$
incq 16(%rax)	$M[16 + 0x100] = M[0x110] = 0x13$ 内存地址: 0x110	$0x100 + 16 = 0x110$
decq %rcx	$\%rcx$	$0x1 - 1 = 0x0$
subq %rdx, %rax	$\%rax$	$0x100 - 0x3 = 0x9D$

单选题 2分

计算机系统 设置

假设R[ax]=FFE8H, R[bx]=7FE6H,
执行指令“subw %bx, %ax”后,
寄存器内容和各标志的变化为

① 转二进制

$ax = 0xFFE8 = 1111\ 1111\ 1110\ 1000$

$bx = 0x7FE6 = 0111\ 1111\ 1110\ 0110$

② 减法 = 加上 -bx 补码 [CPU不直接加法] -bx = bx 取反 + 1

-bx 补码 (取反加1)

$1000\ 0000\ 0001\ 1001$

$1000\ 0000\ 0001\ 1010$

∴ -bx 补码 0x801A

③ $1111\ 1111\ 1110\ 1000$ (ax)
 $1000\ 0000\ 0001\ 1010$ (-bx)

$11000\ 0000\ 0000\ 0010$

∴ 取低16位 (进位不取)

$ax = 0x8002$

CF=0 (加法进位) 减法没进 ZF (零标志)=0 SF=1 (最高位为1)

A R[ax]=8002H, OF=0, SF=1
CF=0, ZF=0

B R[ax]=8002H, OF=1, SF=1
CF=0, ZF=0

C R[bx]=8002H, OF=0, SF=1
CF=0, ZF=0

D R[bx]=8002H, OF=1, SF=1
CF=0, ZF=0

进位标志
OF=0 (进位)
ax 取反 bx 正
结果不为0 不为0

提交



分三类:

(1) 根据单个标志的值转移

(2) 按无符号整数比较转移

(3) 按带符号整数比较转移

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 A > B
10	jae/jnb label	CF=0 OR ZF=1	无符号整数 A ≥ B
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 A < B
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 A ≤ B
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 A > B
14	jge/jnl label	SF=OF OR ZF=1	带符号整数 A ≥ B
15	jl/jnge label	SF≠OF AND ZF=0	带符号整数 A < B
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 A ≤ B

以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。



跳转指令的编码

计算机系统基础I

```

1    movq    %rdi, %rax  把参数从 %rdi 复制到 %rax
2    jmp     .L2         无条件跳转到 L2 (第一次跳转, 对应下面第 10 行 jmp)
3    .L3:
4    sarq    %rax        算术右移 (除以 2)
5    .L2:
6    testq   %rax, %rax  测试 %rax 是否为 0 或正数
7    jg      .L3         条件跳转, 如果满足 jg 条件, 就跳回上面 .L3 处
8    rep; ret

```

指令要求:

汇编器产生的 “.o” 格式的反汇编版本如下:

```

1  0: 48 89 f8          mov     %rdi,%rax
2  3: eb 03             jmp     8 <loop+0x8>
3  5: 48 d1 f8          sar     %rax
4  8: 48 85 c0          test    %rax,%rax
5  b: 7f f8             jg      5 <loop+0x5>
6  d: f3 c3             repz retq

```

在内存中找地址 本段代码 (操作码) 偏移量



- 传递控制

过程的调用和返回

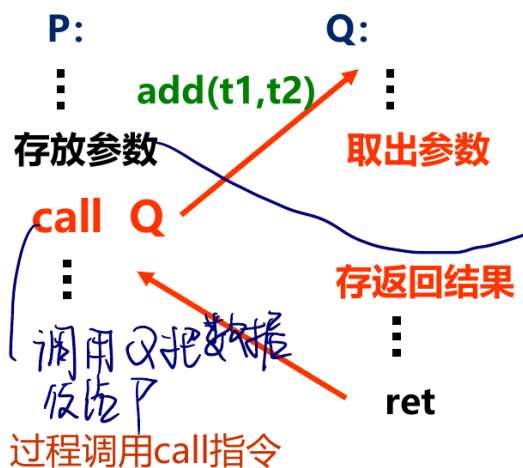
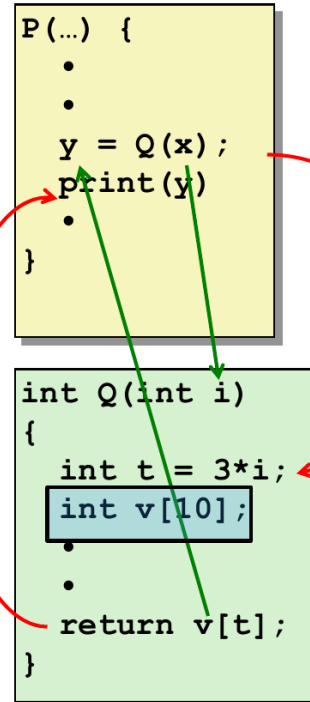
- 传递数据

参数的传递和返回值

- 内存的分配和释放

为局部变量分配空间及释放

x86-64 的过程实现包括过程调用和返回指令和一些对机器资源(例如寄存器和程序内存)使用的约定规则。



直接调用: call 标号

间接调用: call *operand

先把传进来的参数放到指定寄存器或堆栈里

(1) P保存返回地址

(2) 跳转到Q

过程返回ret指令

(1) 从堆栈栈顶取出P的返回地址

(2) 跳转到返回地址, 即call Q指令后面的指令继续执行。



数据传送 [P调又传参, 优先用寄存器, 速读!] 计算机系统基础I

寄存器

~~x86-64~~ 号数

- 通过寄存器最多传递6个整型(例如整数和指针)参数

8字节

%rdi
%rsi
%rdx
%rcx
%r8
%r9

第一个参数

第二个参数

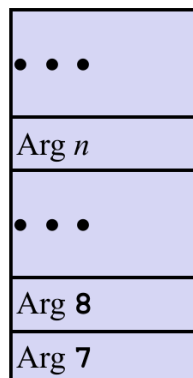
返回值 执行完又返回

%rax

第6个参数

堆栈

- 需要的时候分配堆栈空间



超过6个存堆栈

1. 必须提前在代码里为这些超过的参数分配(好空间)

操作数宽度 (字节)	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL



x86-64 栈帧

栈生长方向: 向低地址

寄存器 %rsp (栈指针) 极其关键

永远指向当前栈帧的栈顶(即地址最低)

计算机系统基础I

内容

- 返回地址
- 调用另一个过程设置的参数7~n
- 局部变量

(1) 寄存器不足够存放所有的本地数据。

(2) 局部变量使用地址运算符 '&', 必须为它产生一个地址。eg: $\text{int}^*p = \&x$

(3) 局部变量是数组或结构

- 被保存的现场寄存器

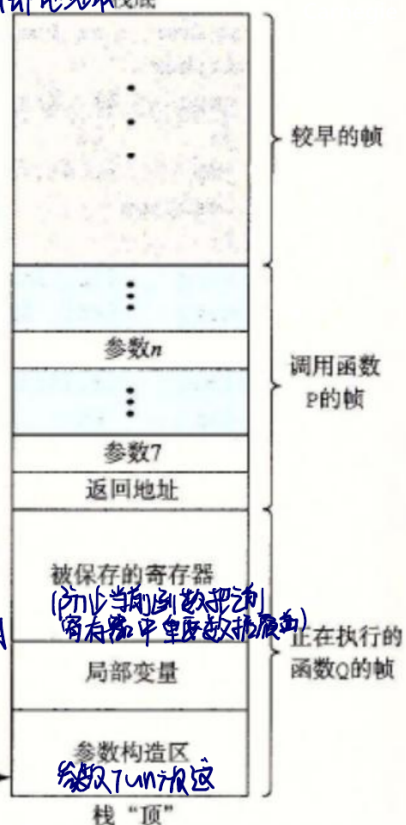
X86-64 过程只分配自己所需要的栈帧部分

实际上, 许多函数甚至根本不需要栈帧。

若所有的局部变量都可以保存在寄存器中

, 而且该函数不会调用任何其他函数。

2. 这函数不调用其他函数



低地址 栈指针 %rsp

参数构造区 参数7~n构造

栈“顶”



数组的分配和访问

$T\ A[L];$

- 用标识符A来作为指向数组开头的指针: $Type\ T^*$
- 用0~N-1的整数索引来访问该数组元素
- 第i ($0 \leq i < N$) 个元素的地址计算公式: $\&A[0] + sizeof(T) * i$

首地址 一个元素的字节数

main ()

```
{
    static short num[4] = { 2, 9, -1, 5 },
    static short *pn[ ] = { num[0], num[1] };
    static short s[2] = { 0, 0 };
```

二维数组

4字节 指向数组

一维数组

short 占2字节

指针 占4字节 (32位)

```
08049300 <num>: 02 00 09 00 ff ff 05 00 03 00 08 00 02 00 fa ff
08049310 <pn>: 00 93 04 08 00 00 00 00 08 93 04 08 00 00 00 00
08049320 <s>: 00 00 00 00
```

小端序: 低字节, 高字节

中间0...是
由于编译器
做内存对齐



结构的分配和访问

将不同类型的数凑在一起

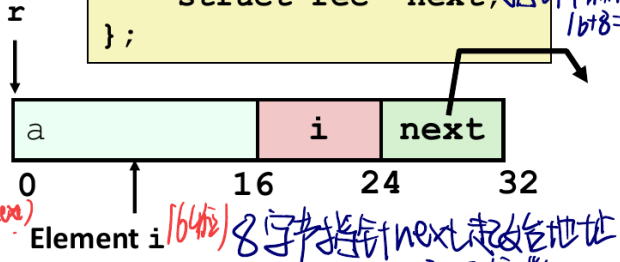
机器级管理结构体的原则:

- ① 结构体在内存中也是连续排列的
- ② 通过相对于结构体起始地址的偏移量来访问结构体

• C Code

```
void set_val
(struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```



Register	Value
%rdi	r
%rsi	val

偏移 8 位数
16+4=20
但偏移 21, 22
23
N=24 字节

```

        jmp .L22
.L11: movslq 16(%rdi), %rax      # i = M[r+16]
        movl  %esi, (%rdi,%rax,4) # M[r+4*i] = val
        movq  24(%rdi), %rdi     # r = M[r+24]
.L22:  testq  %rdi, %rdi        # Test r
        jne   .L11              # if !=0 goto loop
```

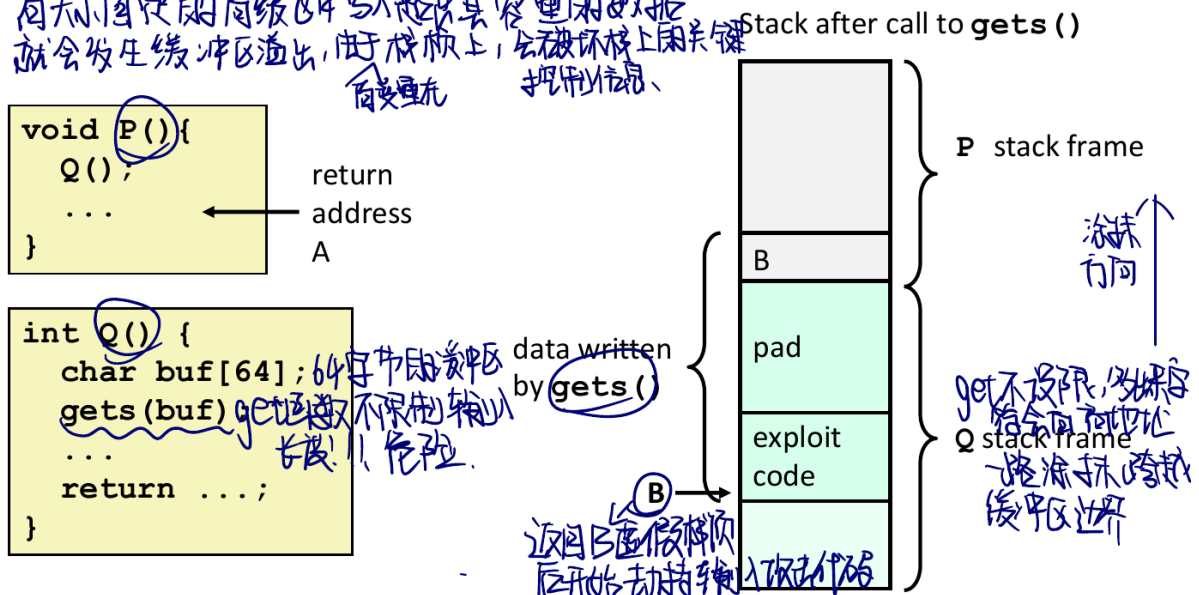
结构体指针 16 字节地址值

继续向后
循环



代码注入攻击 Code Injection Attacks

向大小固定的缓冲区写入超过其容量的数据，就会发生缓冲区溢出，由于栈帧上，会破坏程序上关键信息、



- 输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码(exploit code)，
- 攻击代码的地址覆盖原返回地址。
- 执行ret 指令跳转到攻击代码



浮点指令与代码

过程中的浮点代码

在x86 -64 中，XMM 寄存器用来向函数传递浮点参数，以及从函数返回浮点值。具体规则：

(1) XMM 寄存器%xmm0~ %xmm7 最多可以传递8 个浮点参数。按照参数列出的顺序使用这些寄存器。可以通过栈传递额外的浮点参数。

(2) 函数使用寄存器%xmm0来返回浮点值。

(3) 当函数包含指针、整数和浮点数混合的参数时，指针和整数通过通用寄存器传递，而浮点值通过XMM 寄存器传递

例：double fl(int x, double y, long z);

x 存放在%edi 中，y 存放在%xmm0 中，而z存放在%rsi 中

double fl(float x, double *y, long *z);

x 放在%xmm0 中，y 放在%rdi 中，而z 放在%rsi 中

通用寄存器前两位第一位是寄存器编号，第二位是寄存器类型



- 可重定位目标文件 (.o)
 - 其代码和数据可和其他可重定位文件合并为可执行文件
 - 每个.o 文件由对应的.c文件生成 (经过预处理、编译、汇编)
 - 每个.o文件代码和数据地址都从0开始 (不能直接运行, 需和其他.o文件合成一个完整可执行文件)
- 可执行目标文件 (默认为a.out) (由多个.o文件合并生成)
 - 包含的代码和数据可以被直接复制到内存并被执行
 - 代码和数据地址为虚拟地址空间中的地址
- 共享的目标文件 (.so) (不需编译时加载, 运行时加载)
 - 特殊的可重定位目标文件, 能在装入或运行时被装入到内存并自动被链接, 称为共享库文件
 - Windows 中称其为 *Dynamic Link Libraries* (DLLs) (动态链接库)



可重定位目标文件格式(.o)

ELF 头

- ✓ 定义了ELF魔数、版本、小端/大端、操作系统平台、目标文件的类型、机器结构类型、节头表的起始位置和长度等

.text 节

- ✓ 编译后的代码部分

.rodata 节

- ✓ 只读数据, 如 printf 格式串 ("hello"), switch 跳转表等

.data 节

- ✓ 已初始化的全局变量 static 局部变量

.bss 节

- ✓ 未初始化全局变量, 仅是占位符, 不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率 int i (静态)

ELF 头
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.rel.txt 节
.rel.data 节
.debug 节
.line 节
.strtab 节
Section header table (节头表)



链接操作的步骤

计算机系统基础I

链接器 Linker 把各个.o 文件链接成可执行文件, 做以下两件事

① Step 1. 符号解析 (Symbol resolution)

– 程序中有定义和引用的符号 (包括变量和函数等)

- `void swap() {...} /* 定义符号 swap */`
- `swap(); /* 引用符号 swap */`
- `int *xp = &x; /* 定义符号 xp, 引用符号 x */`



编译器将定义和引用的符号存放在一个符号表 (symbol table) 中。

链接器将每个符号的引用都与一个确定的符号定义建立关联

② Step 2. 重定位

- 将多个代码节与数据节分别合并为一个单独的代码段和数据段
- 计算每个定义的符号在虚拟地址空间中的绝对地址
- 将可执行文件中符号引用处的地址修改为重定位后的地址信息

回头将写着“相对零地址”的符号引用处全部强行修改为这个刚算出来的绝对虚拟地址



符号和符号解析

计算机系统基础I

每个可重定位目标模块 m 都有一个符号表, 它包含了在 m 中定义的符号。

有三种链接器符号:

① Global symbols (模块内部定义的全局符号)

- 由模块 m 定义并能被其他模块引用的符号。例如, 非static 函数和非static 的全局变量 (指不带static 的全局变量)

如, main.c 中的全局变量名 buf

② External symbols (外部定义的全局符号)

- 由其他模块定义并被模块 m 引用的全局符号

如, main.c 中的函数名 swap

③ Local symbols (本模块的局部符号)

- 仅由模块 m 定义和本地引用的符号。例如, 在模块 m 中定义的带static 的函数和全局变量

如, swap.c 中的static 变量名 bufp1



链接器局部符号不是指程序中的局部变量 (分配在栈中的临时性变量) 链接器不关心这种局部变量



main.c

```
int buf[2] = {1, 2};
extern void swap();

int main()
{
    swap();
    return 0;
}
```

swap.c

```
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

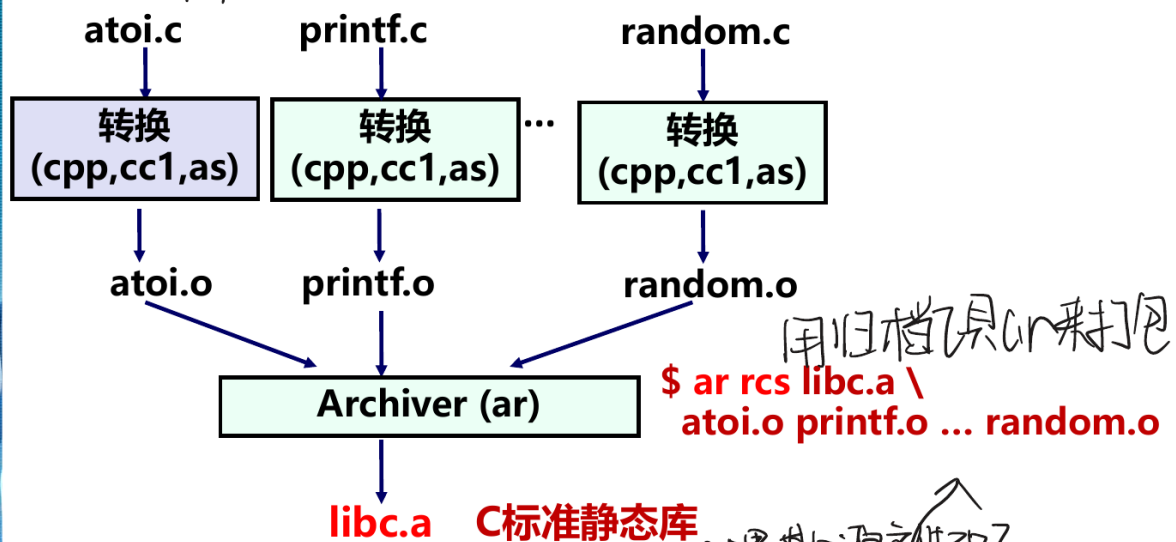
void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

哪些是全局符号？哪些是外部符号？哪些是局部符号？



静态库的创建

每个常用的.o文件打包成个单独的文件，.a结尾



- Archiver (归档器) 允许增量更新，只要重新编译需修改的源码并将其.o文件替换到静态库中。

在gcc命令行中无需明显指定C标准库libc.a(默认库)



链接顺序问题

静态链接器处理命令参数，
从左到右扫描文件，维护三个集合

计算机系统基础I

- 假设调用关系如下：
func.o 调用 libx.a 和 liby.a 中的函数
libx.a 调用 libz.a 中的函数
libx.a 和 liby.a 之间、liby.a 和 libz.a 相互独立
则以下几个命令行都是可行的：
 - ✓ gcc -static -o myfunc func.o libx.a liby.a libz.a
 - ✓ gcc -static -o myfunc func.o liby.a libx.a libz.a
 - ✓ gcc -static -o myfunc func.o libx.a libz.a liby.a
- 假设调用关系如下：
func.o → libx.a 和 liby.a 中的函数
libx.a → liby.a 同时 liby.a → libx.a
则以下命令行可行：
 - gcc -static -o myfunc func.o libx.a liby.a libx.a
 - gcc -static -o myfunc func.o liby.a libx.a liby.a



可执行目标文件格式

计算机系统基础I

把符号解析和重定位都做完

与可重定位文件稍有不同：

- ELF头中字段 **e_entry** 给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0
- 多一个**程序头表**，也称**段头表** (segment header table)，是一个结构数组
- 多一个**.init节**，用于定义 **_init** 函数，该函数用来进行可执行目标文件开始执行时的初始化工作
- 少两个**.rel节** (无需重定位)
(所有相地址行修改为虚拟空间中绝对地址 (不需重定位信息))

ELF 头	
程序头表	只读 (代码) 段
.init 节	只读 (代码) 段
.text 节	只读 (代码) 段
.rodata 节	只读 (代码) 段
.data 节	读写 (数据) 段
.bss 节	读写 (数据) 段
.symtab 节	无需装入到存储空间的信息
.debug 节	无需装入到存储空间的信息
.strtab 节	无需装入到存储空间的信息
.line 节	无需装入到存储空间的信息
Section header table (节头表)	无需装入到存储空间的信息



C程序如下所示，其中f1(n)函数用于计算 $f(n) = \sum_{i=0}^n 2^i = 2^{n+1} - 1$

=11.....1B
(有n+1个1)

```
#include "stdio.h"
int f1( int n )
{ int i;
  int sum = 1, power = 1;
  for ( i = 0; i <= n - 1; i++ )
  { power *= 2;
    sum += power;
  }
  return sum;
}
void main()
{ int n,sum;
  scanf("%d",&n);
  sum=f1(n);
  printf("sum=%d\n",sum);
}
```



x86-64 反汇编代码

计算机系统基础I

内存地址
↓
00401176 <f1>:
401176: f3 0f 1e fa
40117a: ba 01 00 00 00
40117f: b9 01 00 00 00
401184: b8 00 00 00 00
401189: eb 07
40118b: 01 d2
40118d: 01 d1
40118f: 83 c0 01
401192: 39 f8
401194: 7c f5
401196: 89 c8
401198: c3

机器码
↓

寄存器表:
第一个
8字节
%rdi
%rsi
%rdx
%rcx
%r8
%r9

寄存器参数表 (顺序)
输入参数 n: %edi / %rdi
(有n+1个1) (4字节)
返回值 sum: %eax / %rax
(return 一行)

```
endbr64
mov $0x1,%edx %edx = power = 1
mov $0x1,%ecx %ecx = sum = 1
mov $0x0,%eax %eax = i = 0
jmp 401192 <f1+0x1c> 条件循环判断
add %edx,%edx power *= %edx = %edx + %edx
add %edx,%ecx sum += power
add $0x1,%eax i++
cmp %edi,%eax 比较 %edi (参数n) 和 %eax (i)
jl 40118b <f1+0x15> 如果 i < n 跳回循环体
mov %ecx,%eax sum 复制到 %eax 作返回值
ret
```

4字节
%edi
%ecx
%edx
%ecx
%r8d
%r9d

C语言指针, long - 8字节 (开头)
int - 4字节 (开头)



0401199 <main>:

401199:	f3 0f 1e fa	endbr64
△ 40119d:	48 83 ec 18	sub \$0x18,%rsp
4011a1:	64 48 8b 04 25 28 00	mov %fs:0x28,%rax
4011a8:	00 00	
4011aa:	48 89 44 24 08	mov %rax,0x8(%rsp)
4011af: 31 c0		xor %eax,%eax
4011b1:	48 8d 74 24 04	lea 0x4(%rsp),%rsi
4011b6:	48 8d 3d 47 0e 00 00	lea 0xe47(%rip),%rdi
○ 4011bd:	e8 be fe ff ff	call 401080 <__isoc99_scanf@plt>
4011c2:	8b 7c 24 04	mov 0x4(%rsp),%edi
○ 4011c6:	e8 ab ff ff ff	call 401176 <f1>
4011cb:	89 c2	mov %eax,%edx
4011cd:	48 8d 35 33 0e 00 00	lea 0xe33(%rip),%rsi
4011d4:	bf 01 00 00 00	mov \$0x1,%edi
4011d9:	b8 00 00 00 00	mov \$0x0,%eax

★ 改变 %rsp 两种方法

① 显式改变: △, 用于栈向下生长, 方法 0x18 = 在栈上预留 24 字节
用于存放临时变量 n, 及安全保障

② 隐式改变: ○, 所有 call 指令, CPU 会自动把当前返回地址压入栈顶。



程序部分调试信息如下:

```
(gdb) i r rip
rip      0x401196
(gdb) i r rsp
rsp      0x7fffffffde18
```

调用 scanf 输入变量 n 指定 0x4(%rsp)

```
(gdb) x/4xg $rsp
0x7fffffffde18: 0x0000000000004011cb
0x7fffffffde28: 0xfc957cf2acabe600
```

0x4 高 4 字节对应 01
0x0000000000000000
0x0000000000000000

1. 基于调试信息, 输入的 n 的值是多少? (十进制表示)

4

2. 此时 eax, edi, ecx 寄存器的值分别是多少? (十进制表示)

4 4 31

第一个输入参数, 运行期间被

循环变量 i, i = 0, 1, 2, 3



3. 执行下述两条指令的作用是什么？（对照C语句解释）

401192: 39 f8

cmp %edi,%eax

401194: 7c f5

jl 40118b <f1+0x15>

for循环的边界条件判断与跳转控制

比较 %eax(i) 与 %edi(n)
j: 若 $i < n$ ($i \leq n$) 则跳回
否则退出, 向下继续执行

4. 在给出的main()的部分反汇编代码中, 哪些指令会改变rsp寄存器的内容? 选取其中一条指令, 基于调试信息, 说明该指令执行后rsp寄存器的内容是什么?

前序

5. 在main()和f1()中都定义了sum变量, 为什么这两个sum变量的访问没有冲突?

☆ 因为这两个sum都为局部变量

都被分配在main自身独立堆栈帧空间内

f1()sum在运行期间被编译器直接映射到 %ecx中临时变量

○ 由于它们分别应用完全隔离的内存区域与硬件寄存器资源, 因有不同作用域与生命周期, 访问时绝不会冲突