

开箱手册-计算机系统

ywy_c_asm

计算学部金牌讲师团

版权声明：本文档允许自由传播分享，仅供学习交流之用，不得用作商业用途，违者后果自负。

前言

- 本教程基于2022年春季学期哈工大计算学部金牌讲师团为2020级的计算机系统课程所开设的期末复习讲座的课件，在原版基础上加入了部分内容，适合作为复习/应付/预习期末考试的资料，主要作应试而用，**并不适合用于实用知识学习**（注意：计统本身是一门十分实用的课程，建议对教材进行深入阅读）。然而，本教程仅覆盖了大部分考试重点内容，还有诸多可以被考的细节没有被覆盖到，这是读者需要注意的。
- 由于作者水平有限，难免会出一些差错，对此提前致以歉意。**此外本教程关于考试内容与重点的划分很大程度上来自个人观点，如果你因此而遭受损失，作者概不负责。**
- 作者在此提供了两次讲座录屏，可以搭配学习：（由于历史原因，这里缺少了15~21以及33节的讲解）
 - 讲座1: https://pan.baidu.com/s/1CMaP5d_6eITdioTShugMtg?pwd=1453 提取码: 1453
 - 讲座2: <https://pan.baidu.com/s/1D0z7JiFua0g0uz-32dNckA?pwd=1453> 提取码: 1453
- 作者还提供了两套配套模拟题：
 - 人肉反编译练习题: https://pan.baidu.com/s/1_J7KVU8NPBpEmerguociRg?pwd=1453 提取码: 1453
 - 期末模拟题: <https://pan.baidu.com/s/1alo3I1RVwFQ6nHxm2glKQw?pwd=1453> 提取码: 1453
- 感谢你的使用，希望它对你有用。

前人经验-对这门课的一些见解

- 珍惜这门课吧！它是你本科生涯里能够遇见的为数不多的好课(以及好教材)了。这门课程名为计算机系统，实际上是在讲x86-64体系结构与linux操作系统的各种机制，通过这一软硬件结合的鲜活实例(也是最常见的实例)来讲述一个计算机系统实际上是如何运行的。在未来，你可能会在《计算机组织与体系结构》、《操作系统》等课程中通过更加抽象与广义的学习了解到与本课程所学截然不同的具体机制，你需要明白本课程讲的仅为一套具体例子，而非广义原理。
- 在开这门课程的同时，你可能还在同时学习《计算机组成原理》这门糟糕的课程，同时开这两门课程是一个极度愚蠢的决定，二者的很多内容高度重合(例如Cache和虚拟内存的题是完全一致的)，并且在一些地方会出现看上去互相冲突的现象(尤其是数据表示那里，唐书完全采取了自己的一套抽象体系)。我的建议是接受现实，在期末考试之前不要混淆两门课程的知识，考完后忘掉计组所讲，但计统讲的真的都是真实有用的知识，这与计组不同。
- 关于考试，历年题很容易就能找到，参考它们的套路实际上足矣。

目录

- 1. [关于进制转换](#)
- 2. [内存中的字节](#)
- 3. [有符号与无符号整数](#)
- 4. [加法的溢出](#)
- 5. [位运算](#)
- 6. [浮点数](#)
- 7. [汇编指令与寻址](#)
- 8. [寄存器](#)
- 9. [标志位与条件转移](#)
- 10. [函数调用与栈帧](#)
- 11. [数组与对齐的结构体](#)
- 12. [switch及其跳转表](#)
- 13. [缓冲区溢出攻击](#)
- 14. [人肉反编译考试解读](#)
- 15. [Y86-64处理器设计](#)
- 16. [程序优化](#)
- 17. [Cache](#)
- 18. [Cache与性能优化](#)
- 19. [从编译到链接](#)
- 20. [符号解析](#)
- 21. [异常](#)
- 22. [虚拟内存机制](#)
- 23. [进程](#)
- 24. [fork与子进程](#)
- 25. [execve与内存映射](#)
- 26. [信号](#)
- 27. [进程的终止与回收](#)
- 28. [并发](#)
- 29. [非本地跳转与“返回n次的函数”](#)
- 30. [shell原理](#)
- 31. [地址翻译的硬件机制](#)

1. 关于进制转换

- 十进制（整数或小数）如何与二进制互相转换？（浮点数计算题会涉及到）
- 十六进制一个位对应4个二进制位，八进制一个位对应3个二进制位（很少用）
- 很多时候高位直接默认0补全

$$1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} + 1 * 2^{-5} \dots = 13.1$$

将 $(13.1)_{10}$ 转换为二进制和十六进制：

13.1的整数部分是13，去掉后变成0.1。

$$13 \div 2 = 6 \dots 1$$

$$6 \div 2 = 3 \dots 0$$

$$3 \div 2 = 1 \dots 1$$

$$1 \div 2 = 0 \dots 1$$

$$\text{则 } (13)_{10} = (1101)_2$$

$0.1 * 2 = 0.2$ ，整数部分是0，去掉后还是0.2。

$0.2 * 2 = 0.4$ ，整数部分是0，去掉后还是0.2。

$0.4 * 2 = 0.8$ ，整数部分是0，去掉后还是0.8。

$0.8 * 2 = 1.6$ ，整数部分是1，去掉后是0.6。

$0.6 * 2 = 1.2$ ，整数部分是1，去掉后是0.2。

.....（开始循环）

将以上所有整数部分组成二进制小数，那么 $(13.1)_{10} = (1101.0001100110011\dots)_2$

每4个二进制位对应一个十六进制位，那么 $(13.1)_{10} = (D.1999\dots)_{16}$

2. 内存中的字节

- 一个字节一般表示为两个16进制位（因此你需要熟练掌握十六进制）
- 小端序（x86-64默认）或大端序决定了内存中字节的解读顺序，顺序很重要！
- 例1. 有变量int x=100000，它在内存中由低到高的字节为：A0 86 01 00
- 例2. 有变量short y=*((short*)((char*)&x) + 1)，那么y的值是什么？0x186
- 例3.

正确理解这种显示方式！对于8位字节A0，低4位是0(0000)，高4位是A(1010)，毕竟这是个十六进制数，A是高位

46. 某C程序(64位模式)的main函数参数argv地址为0x0000413433323110，其内容如下：

30 31 32 33 34 41 00 00 33 31 32 33 34 41 00 00
35 31 32 33 34 41 00 00 00 00 00 00 00 00 00 00
31 43 00 30 00 32 42 00 38 00 31 31 32 32 00 30
32 33 00 61 41 00 31 00 32 00 33 00 31 00 00 31

请写出 程序名：1C，本程序的参数个数 2

按顺序写出各个参数为 0 2B

温馨提示：建议在一纸开卷上（起码）准备字符‘0’、‘a’、‘A’的ASCII码

3. 有符号与无符号整数

- 明确：二者仅仅是对同一种位模式（或者存储的同一种数据）做的不同解读。
- 例如在内存中存储的1个字节0xFF会被signed char解读为有符号数-1，也会被unsigned char解读为无符号数255。

31. (X) C 语言程序中，有符号数强制转换成无符号数时，其二进制表示将会做相应调整。

- 有符号整数的数学意义：符号位位权为负。
$$(b_{MSB}...b_2b_1b_0) = -b_{MSB} * 2^{MSB} + \sum_{i=0}^{MSB-1} b_i 2^i$$
- 如何将一个负数表示为补码形式？先写出其相反数（正数）的二进制表示，再将所有位取反，再进行二进制+1。
- 2 -> 00000000...0010，取反得11111111...1101，加1得11111111...1110

21. int 数 -2 的机器数二进制表示_____。

3. 有符号与无符号整数

注：这里可以参考计组的教程，二者是一样的。

- 可以从有符号数的数学意义出发解释这个转换方法（不考，看看就行）：

证明： 若 x 是以 w 位存储的正数，那么 $\text{not}(x) + 1$ 是 $-x$ 的补码形式。

考虑每一位会有一个**值×位权**的贡献，低 $w - 1$ 位的位权之和是 $2^{w-1} - 1$ ，那么当 x 取反后，低 $w - 1$ 位的贡献和就变成了 $2^{w-1} - 1 - x$ 。（举个例子，比如4位二进制数的位权之和为 $2^4 - 1 = 15$ ，0110的位贡献和是6，那么取反后的1001的位贡献和就是 $15 - 6 = 9$ 。

而此时最高符号位一定由0变成1，产生一个 -2^{w-1} 的贡献，因此 $\text{not}(x)$ 在有符号整数意义下是 $2^{w-1} - 1 - x - 2^{w-1} = -1 - x$ ，再加上1就变成了 $-x$ 。

3. 有符号与无符号整数

- 补码规则使得我们仍然可以用普通的二进制加法来计算有符号数的加减法，因此CPU的加法器不需要知道数据是不是有符号的！

32. (✓) CPU 无法判断参与加法运算的数据是有符号或无符号数。

- 巨坑：C语言中若有符号整数和无符号整数混合运算，结果解读为无符号类型！若混合比较则解读为无符号比较！这是编译器作出的行为。

- 例1. 下列逻辑表达式中哪些是真的？
 $\textcircled{1} 0 > -1$, $\textcircled{2} 0u > -1$, $\textcircled{3} 0 > -1 + 0u$

$\textcircled{1} 0 > -1$ ✓ 有符号比较
 $\textcircled{2} 0u > -1$ ✗ 无符号比较
 $\textcircled{3} 0 > -1 + 0u$ ✗ 无符号比较

无论如何解读，二进制表示始终都是111.....111

- 例2.

2. C 语言程序如下，叙述正确的是 ()

```
#include <stdio.h>
```

```
#define DELTA sizeof(int)
```

```
int main(){
```

```
    int i;
```

```
    for (i = 40; i - DELTA >= 0; i -= DELTA)
```

```
        printf("%d ", i);
```

```
}
```

A. 程序有编译错误

B. 程序输出 10 个数：40 36 32 28 24 20 16 12 8 4 0

✓ C. 程序死循环，不停地输出数值

D. 以上都不对

警惕sizeof坑！

sizeof本身是无符号类型！

表达式本身是无符号类型，永远>=0

4. 加法的溢出

- 不要试图直接用加减运算判断溢出！
- 二进制加法出现进位，并不代表溢出！


没有进位时也会产生溢出！

- 一非负一负（符号位不同）相加不会产生溢出，只有当两个正数/两个负数相加，真实结果太大了或者太小了，超出表示范围，导致结果不对。
- 两个正数相加溢出，机器结果必然为负。两个负数相加溢出，机器结果必然为正。
- 因此看结果的符号位是否和两个符号位相同的加数不同即可。（OF标志位原理）

 练习题 2.30 写出一个具有如下原型的函数：

```
/* Determine whether arguments can be added without overflow */  
int tadd_ok(int x, int y);
```

如果参数 x 和 y 相加不会产生溢出，这个函数就返回 1。

 练习题 2.31 你的同事对你补码加法溢出条件的分析有些不耐烦了，他给出了一个函数 tadd_ok 的实现，如下所示：

```
/* Determine whether arguments can be added without overflow */  
/* WARNING: This code is buggy. */  
int tadd_ok(int x, int y) {  
    int sum = x+y;  
    return (sum-x == y) && (sum-y == x);  
}
```

你看了代码以后笑了。解释一下为什么。

4. 加法的溢出

- 这个不那么直观的性质可以考虑有符号整数的表示范围来证明（看看就好）：

证明1： 一个非负数与一个负数相加不会产生溢出。

设 A 为 w 位有符号非负数， B 为 w 位有符号负数，那么 $A \in [0, 2^{w-1} - 1]$, $B \in [-2^{w-1}, 0)$, $A + B \in [-2^{w-1}, 2^{w-1} - 1)$ ，在有符号 w 位整数的表示范围内，不会溢出。

证明2： 两个正数进行有符号相加溢出，结果必然为负。

设位数为 w ，那么最大能表示的正数为 $2^{w-1} - 1$ ，也就是说 $A + B > 2^{w-1} - 1$ ，而 $A, B \leq 2^{w-1} - 1$ ，那么 $A + B \leq 2^w - 2$ ，即 $A + B \in [2^{w-1}, 2^w - 2]$ ，显然它的二进制表示是一个 w 位二进制数，且第 $w - 1$ 位为1，解读为有符号整数时符号位也就是1，因此为负。

5. 位运算

- 按位的与 (&)、或 (|)、非 (~)、移位 (<<、>>) 要知道是什么，要会算会写！
(要和逻辑的与 (&&) 或 (||) 非 (!) 区分)
- 例1.** 21. 判断整型变量 n 的位 7 为 1 的 C 语言表达式是_____ $(n \& (1 \ll 7)) \neq 0$ 或者 $(n \& 0x80) == 0x80$
- 例2.** 计算 C 语言表达式 $(0x1234567 \& 0x114514) \mid \mid 0x1919810$ 的值。
- 对于右移，一定要搞清楚是算术右移 (SAR) 还是逻辑右移 (SHR)！
- 例3.** 分别计算 C 语言表达式 $(\text{int})0xFFFF0000 \gg 16$ 和 $0xFFFF0000u \gg 16$ 的值。

$0xFFFF0000$
0xFFFFFFFF, 有符号数带着符号位算术右移

$0xFFFF0000u$
0x0000FFFF, 无符号数仅逻辑右移
- 不管是非负数还是负数，算术右移的数学意义都是除以 2 的幂再向下取整。左移同理。

6. 浮点数（重点！）

注：计统和计组这两个课程都讲了浮点数的类似内容，然而计组唐书试图从抽象的角度讲解(出题一般都是随便定义的浮点数格式)，计统基本上只需要理解现实的float和double的格式(IEEE754标准)即可。且唐书讲的一些细节与IEEE754标准冲突，务必不要混淆两门课程的知识。

- 熟悉浮点数格式，知道浮点数的三要素：符号、阶码、尾数，知道规格数和非规格数的区别，理解浮点数的分布，知道机器数怎么存储。建议一定把它们写在小抄上
- 数学意义上，（规格化的） $x = (-1)^s M 2^E$ ， $s \in \{0,1\}$ ， $1 \leq M < 2$ ， E 为整数，那么 s 就是符号， M 就是尾数， E 就是阶码。
- 机器意义上，符号 s 占最高位，尾数 E 表示成二进制小数的形式，仅取小数点后面的若干位（忽略小数点前面的1）。
- 阶码 E 存储为 w 位的整数，存储的机器值 $exp = E + 2^{w-1} - 1$ ，且必须有 $E \in [-2^{w-1} + 2, 2^{w-1} - 1]$ ，一共有 $2^w - 2$ 种阶码（机器值 exp 为全0或全1是特殊的）。
- 对于float， $exp = E + 127$ ，对于double， $exp = E + 1023$

6. 浮点数（重点！）

描 述	exp	frac	单精度		双精度	
			值	十进制	值	十进制
0	00...00	0...00	0	0.0	0	0.0
最小非规格化数	00...00	0...01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
最大非规格化数	00...00	1...11	$(1-\varepsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\varepsilon) \times 2^{-1022}$	2.2×10^{-308}
最小规格化数	00...01	0...00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
1	01...11	0...00	1×2^0	1.0	1×2^0	1.0
最大规格化数	11...10	1...11	$(2-\varepsilon) \times 2^{127}$	3.4×10^{38}	$(2-\varepsilon) \times 2^{1023}$	1.8×10^{308}

图 2.36 非负浮点数的示例

- 对于非规格化数，机器数exp字段为特殊值0，但此时阶码为 $-2^{w-1} + 2$ 而不是按照正常阶码机器数规则的 $-2^{w-1} + 1$ ！此时尾数默认小数点前面为0。也就是说非规格化数是 $(-1)^s * 2^{2-2^{w-1}} * 0.xxxxxx \dots$
- 为什么要有非规格化数？为了在一些精度较高的计算的场合下表示最接近0的极小数，以及能够表示0（规格化数能表示0吗？）。注意：存在两个数学意义上相等但存储不同的浮点数+0和-0。
21. float 数 0 的机器数有_____2_____个。
- 设尾数的机器数有n位，最小的正非规格化数是 $2^{2-2^{w-1}} * 2^{-n}$ （即尾数为二进制小数0.0000...01），最大的正非规格化数是 $2^{2-2^{w-1}} * (1 - 2^{-n})$ （即尾数为二进制小数0.111...11）。最小的正规格化数为 $2^{2-2^{w-1}} * 1$ （刚好比最大非规格化数大一点），最大的正规格化数为 $2^{2^{w-1}-1} * (2 - 2^{-n})$ （尾数为1.111...11）

这些边界值一定要知道且会算！

6. 浮点数（重点！）

- 最后一种特殊情况，若阶码机器数 exp 为全1二进制数，此时如果尾数机器数为0，那么表示无穷大 inf （视符号位表示为正无穷或负无穷）。如果尾数机器数不是0，那么表示非数 NaN 。所以有很多很多存储值不同的浮点数，但它们都能表示 NaN ！再联系到+0和-0，就有了经典问题：① int 和 float 能表示的数，谁更多？② int 和 float 可以在机器中存储的值的种类，谁更多？
二者一样多，都是32位， 2^{32} 种位模式
- 关于整数与浮点数的转换问题，要明白 float 尾数23位，阶码8位， double 尾数52位，阶码11位。那么 int 转换为 double 可以精确转换（52远大于32），转换为 float 不会溢出（ float 表示范围远大于 int ）但会有精度损失。而浮点数转换为整数则可能会产生溢出！

35. ~~（X）~~ C语言中数值从 int 转换成 double 后，数值虽然不会溢出，但有可能是不精确的。

6. 浮点数（重点！）

- 关于舍入，牢记浮点数默认使用向偶数舍入！！！而不是四舍五入、向0舍入、向下舍入之类的。
33. () 浮点常数编译时缺省舍入规则是四舍五入。 33. (X) 浮点数计算后的舍入规则是四舍五入。
- 啥是向偶数舍入？如果要截取的位后面的部分不是中间值，则采取最接近舍入的策略（类似四舍五入）。如果恰好就是中间值（与四舍五入的本质区别！），那么你需要选择进位或者不进位，使得截取的位的最后一位是偶数。
- **例1.**十进制下的向偶数舍入（保留小数点后1位）：
 - 1.549999->1.5(就近)，**1.55->1.6(偶)**，**1.65->1.6(偶)**，1.650001->1.7
- **例2.**二进制下的向偶数舍入（保留小数点后1位）：
 - 1010.011->1010.1(就近)，**1010.110->1011.0(偶)**

6. 浮点数（重点！）

- 如果出题让你写一个十进制小数（特别爱出0.1）在内存中的浮点数表示，你会算吗？（本质上还是二进制小数的转换，再加上机器数存储格式和舍入规则就行了）
- 例1.

6. C语言中float数据0.1的机器数表示错误的是（ ）
A. 规格化数 B. 不能精确表示 ~~X~~与0.2有1个二进制位不同 D. 唯一的
- 另外一个常见的坑是浮点数加法不满足分配律，例如 $(3.14+1e10)-1e10=0$ ，但 $3.14+(1e10-1e10)=3.14$ 。精度局限性导致的。但浮点数加法在不考虑inf和NaN的情况下是满足单调性的。

实在太大了，3.14对尾数的贡献十分靠后，被截断损失，几乎就是1e10

7. 汇编指令与寻址

- 你需要起码认识如下的汇编指令：mov、add、sub、jmp、条件转移、cmp、test、call、ret、lea、push、pop、nop以及一些位运算指令，以及它们在AT&T格式下的（有的场合下可能会不带）操作数长度后缀b w l q（8、16、32、64位），注意AT&T格式的操作数顺序！（关于这些东西很多地方记不住的建议打小抄）
- 对于操作数，它可以是一个立即数，或者寄存器，或者内存单元，需要能看懂这些操作数的格式：

注：这里的寻址方式是现实中x86所支持的，需要和计组唐书上讲的另一套陈旧的寻址方式加以区分，但两套寻址方式都是你分别要在两个课程中记住的

类型	格式	操作数值	名称
立即数	Imm	Imm	立即数寻址
寄存器	r_a	$R[r_a]$	寄存器寻址
存储器	Imm	$M[Imm]$	绝对寻址
存储器	(r_a)	$M[R[r_a]]$	间接寻址
存储器	$Imm(r_b)$	$M[Imm+R[r_b]]$	(基址+偏移量)寻址
存储器	(r_b, r_i)	$M[R[r_b]+R[r_i]]$	变址寻址
存储器	$Imm(r_b, r_i)$	$M[Imm+R[r_b]+R[r_i]]$	变址寻址
存储器	(r_i, s)	$M[R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(r_i, s)$	$M[Imm+R[r_i] \cdot s]$	比例变址寻址
存储器	(r_b, r_i, s)	$M[R[r_b]+R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(r_b, r_i, s)$	$M[Imm+R[r_b]+R[r_i] \cdot s]$	比例变址寻址

常用于数组
此时R[rb]为数组起始地址，R[ri]为元素索引，s为元素大小(字节数)

常用于结构体数组

图 3-3 操作数格式。操作数可以表示立即数(常数)值、寄存器值或是来自内存的值。比例因子s必须是1、2、4或者8

7. 汇编指令与寻址

这些x86汇编指令应该要知道

- `mov A,B` 将A的值赋给B
- `add A,B` 将A+B的值赋给B
- `sub A,B` 将B-A的值赋给B
- `push A` 压栈, 先将`rsp-=8`, 再令`[rsp]=A`
- `pop A` 弹栈, 先令`A=[rsp]`, 再将`rsp+=8`
- `call A` 函数调用
- `ret` 函数返回
- `lea 内存单元,A` 将内存单元的地址值赋给A (方便的四则运算)
- `cmp A,B` 计算B-A但不保存结果, 比较B和A大小

**rsp始终指向
栈顶元素!**

- `test A,B` 计算A&B但不保存结果, 用于二进制位的检测 (常常用来判符号位)
- `and or xor not` 基础位运算
- `shl` 左移
- `shr` 逻辑右移
- `sar` 算术右移
- `nop` 啥都不干
- `movslq` 将32位值做符号扩展赋给64位寄存器 (常用在int型数组下标的元素访问)
- `movabsq` 将一个64位立即数赋给64位寄存器 (`movq`只能操作可表示为32位的立即数)

7. 汇编指令与寻址

- 说出下列汇编指令的含义，若有错请指出：

- 例1. `movq $0x23333, %rax`

- 例2. `movl 0x40100(,%rbx,4), %eax` 典型的数组访问

- 例3. `movl (%rax,%ebx), %ecx` ✗

- 例4. `addq %rsi, (%rdi)`

- 例4. `addq (%rsi), (%rdi)` ✗

- 例5. `leaq 233(%rax,%rbx,8), %rcx`

- 例6. `xorq %rax,%rax` 常用于寄存器清零

- 例7. `testq %rax,%rax` 常用于判断0或负数

3. 下数值列叙述正确的是 ()

✓ 人. 一条 `mov` 指令不可以使用两个内存操作数

✗ 在一条指令执行期间, CPU 不会两次访问内存

典型反例如把寄存器值add到内存单元上, 先从内存中取出原值再计算加法再把结果存回去

注：从指令系统的角度来看，x86属于CISC指令集，指令执行时的行为可以较为复杂，允许多次访存，我们的计统课程默认以x86为准。而RISC指令集(如MIPS)基本上在执行时不会多次访存，为了简化设计。毕竟，访存是一件比看起来复杂得多且代价巨大的事情来自造CPU人的叹息

8. 寄存器

8. 计算机是 64 位是指 (B)

A. 数据总线 64 根

B. CPU 中通用寄存器是 64 位的

C. 安装的操作系统是 64 位的

D. CPU 中所有寄存器都是 64 位的

- 机器字长=CPU中ALU位数=CPU中通用寄存器位数 (64)
- x86-64中特殊规定, 若指令修改了64位寄存器的低32位, 自动将高32位清零。
D.X86-64 指令"mov\$1,%eax"不会改变%rax 的高 32 位
- rip作为程序计数器PC, 始终指向当前执行的指令的下一条指令地址, 只有转移指令能修改它们的值 (转移指令原理)
~~✗~~ CPU 不总是执行 CS::RIP 所指向的指令, 例如遇到 call、ret 指令时
无论如何这个不可能改变
- rsp永远恰好指向栈顶元素 (的最低字节), 栈底到栈顶是从高地址到低地址增长的! rbp几乎也是专门用来访问栈的

63	31	15	7	0	
%rax	%eax	%ax	%al		返回值
%rbx	%ebx	%bx	%bl		被调用者保存
%rcx	%ecx	%cx	%cl		第4个参数
%rdx	%edx	%dx	%dl		第3个参数
%rsi	%esi	%si	%sil		第2个参数
%rdi	%edi	%di	%dil		第1个参数
%rbp	%ebp	%bp	%bpl		被调用者保存
%rsp	%esp	%sp	%spl		栈指针
%r8	%r8d	%r8w	%r8b		第5个参数
%r9	%r9d	%r9w	%r9b		第6个参数
%r10	%r10d	%r10w	%r10b		调用者保存
%r11	%r11d	%r11w	%r11b		调用者保存
%r12	%r12d	%r12w	%r12b		被调用者保存
%r13	%r13d	%r13w	%r13b		被调用者保存
%r14	%r14d	%r14w	%r14b		被调用者保存
%r15	%r15d	%r15w	%r15b		被调用者保存

图 8-2 整数寄存器。所有 16 个寄存器的低位部分都可以作为字节、字(16 位)、双字(32 位)和四字(64 位)数字来访问

9. 标志位与条件转移

- 理解4个主要标志位CF(进位)、ZF(结果为0)、SF(有符号结果为负)、OF(有符号结果溢出)的意义，明白什么指令能修改标志位，如何通过标志位实现条件转移（以及进一步实现的分支与循环）。
- 需要结合上一章的有符号/无符号整数运算理解！
- 例1. `eax=0xFFFFFFFF`，`addl $1,%eax`后，`CF=1,ZF=1,SF=0,OF=0`
- 例2. `eax=0x7FFFFFFF`，`addl $1,%eax`后，`CF=0,ZF=0,SF=1,OF=1`
- 例3. `eax=负数`，`test %eax,%eax`后，`CF=0,ZF=0,SF=1,OF=0`
- 例4. 哪些指令不会修改标志位？
① `addq %rbx,%rax`，② `leaq (%rbx,%rax),%rax`，
③ `cmpq %rbx,%rax`，④ `jmp label`，⑤ `popq %rax`，⑥ `movq %rbx,%rax`

leaq可以进行四则运算，但不能修改标志位（CPU结构的原因）

9. 标志位与条件转移

- 假设A和B是int型变量，如何通过计算A-B后的标志位（即cmp指令）来判断下列条件：①A == B，②A < B，③(unsigned)A < B，④A > B？
- 若A=B，那么A-B=0，等价于ZF=1（转移指令je、jne的原理）
- 若A<B，那么A-B的结果在有符号意义下为负数，但不能直接判断SF，还要看是否溢出(OF)，如果真的溢出了，如果真的是A<B，那么A是负数，B是正数，溢出后结果一定为正。所以A<B等价于 $(\sim OF \& SF) \mid (OF \& \sim SF) = SF \wedge OF$ 。（j1的原理）
- 若无符号意义下A<B，那么直接做二进制减法会产生借位（进位），等价于CF=1（ja、jb的原理）
- 若A>B，等价于 $\neg(A < B) \& \& (A \neq B)$ ，也就是 $\sim(SF \wedge OF) \& \& \sim ZF$ （jg的原理）

4. 条件跳转指令 JE 是依据 () 做是否跳转的判断
A. ZF B. OF C. SF D. CF

9. 标志位与条件转移

- 一定要明白CPU只是机械地做二进制减法然后根据结果设置进位，并不知道是有符号还是无符号，这只是一种解读，取决于编译器(或程序员)如何选择正确的指令（条件转移、算术/逻辑右移等）或者标志位。

指令	同义名	跳转条件	描述
jmp Label		1	直接跳转
jmp *Operand		1	间接跳转
jz Label	jz	ZF	相等/零
jnz Label	jnz	-ZF	不相等/非零
js Label		SF	负数
jns Label		-SF	非负数
jg Label	jnl	-(SF ^ OF) & -ZF	大于(有符号>)
jge Label	jnl	-(SF ^ OF)	大于或等于(有符号>=)
jl Label	jnge	SF ^ OF	小于(有符号<)
jle Label	jng	(SF ^ OF) ZF	小于或等于(有符号<=)
ja Label	jnb	-CF & -ZF	超过(无符号>)
jae Label	jnb	-CF	超过或相等(无符号>=)
jb Label	jnae	CF	低于(无符号<)
jbe Label	jna	CF ZF	低于或相等(无符号<=)

图 3-15 jump 指令。当跳转条件满足时，这些指令会跳转到一条带标号的目的地址。有些指令有“同义名”，也就是同一条机器指令的别名

41. 针对有符号及无符号整数的加法运算，CPU、编译器、程序员是怎么配合完成不同类型整数的数据表示、数据运算，并如何判断其结果是否超出范围的？

答：C 程序员用 unsigned 或 signed（缺省，可不用）来区分数据类型，常数后加 U 表示无符号数。程序中可以自由进行比较、赋值、运算等。

CPU 并不知道数据类型，只是按位进行加法操作，并按照逻辑规定设置 CF、OF、ZF、SF、PF、AF 等标志位。

编译器会将数据转换成相应的二进制编码（无符号数）或补码（有符号数），如果类型不一致，都转换成无符号数再进行操作。

数组操作之后，编译器根据不同数据类型选择不同的分支转移指令，可按照如上标志位，或无符号数用 JA/JB 等、有符号数用 JG/JL 等判断数据大小，并进行跳转。无符号溢出用 JC、有符号溢出用 JO 判断。

9. 标志位与条件转移: `if`的处理 (重点!)

- 一般情况下的`if`, 本质上是通过条件转移指令有选择性地跳过不想执行的指令。

```
if(条件){  
    A;  
}  
else{  
    B;  
}
```

→

```
    如果条件不成立则goto B_start;  
    A;  
    goto B_done;  
B_start:  
    B;  
B_done:
```

有的时候编译器可以把条件转移指令使用的较为灵活, 需要结合实际情况分析!

- 例1. `if((int)eax < 233){`
 指令A;
}

→

```
    cmp1 $233, %eax  
    jge if_done  
    指令A  
if_done:
```

或者

```
    cmp1 $233, %eax  
    j1 do_A  
    jmp if_done  
do_A:  
    指令A  
if_done:
```

如果把`jb`换成`j1`会出现什么错误?

- 例2. `if((unsigned)eax < 233){`
 指令A;
}

→

```
    cmp1 $233, %eax  
    jae if_done  
    指令A  
if_done:
```

或者

```
    cmp1 $233, %eax  
    jb do_A  
    jmp if_done  
do_A:  
    指令A  
if_done:
```

9. 标志位与条件转移: for/while的处理 (重点!)

- 其实和if较为类似。

```
for(初始化;条件;执行动作){  
    循环体;  
}
```



```
    初始化;  
    goto check_label;  
for_start:  
    循环体;  
    执行动作;  
check_label:  
    若条件不满足则 goto for_done;  
    goto for_start; //继续这一轮循环  
for_done:
```

```
do{  
    循环体;  
}while(条件);
```



```
while_start:  
    循环体;  
    若条件满足则 goto while_start;
```

```
while(条件){  
    循环体;  
}
```



```
while_start:  
    若条件不满足则 goto while_done;  
    循环体;  
    goto while_start;  
while_done;
```

- 例1.

```
for(int ebx=0;ebx<6;ebx++){  
    循环体;  
}
```



```
    movl    $0x0, %ebx //初始化循环变量  
    jmp     check_label  
for_start:  
    循环体;  
    addl    $0x1, %ebx //执行动作  
check_label:  
    cmpl    $0x6, %ebx //条件判断  
    jge     for_done  
    jmp     for_start  
for_done:
```

- 例2.

```
for(int ebx=5;ebx>=0u;ebx--){  
    循环体;  
}
```



```
    movl    $0x5, %ebx  
    jmp     check_label  
for_start:  
    循环体;  
    subl    $0x1, %ebx  
check_label:  
    cmpl    $0x0, %ebx  
    jbe     for_done //CF=1 ?  
    jmp     for_start  
for_done:
```

根本停不下来!

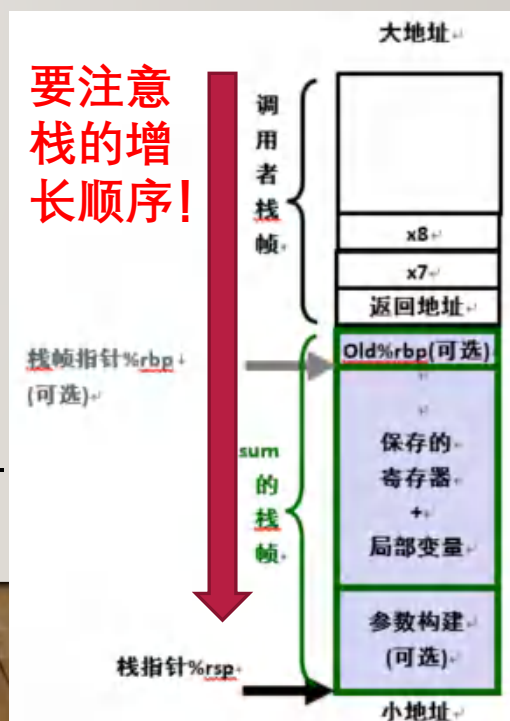
10. 函数调用与栈帧 (重点!)

41. 从汇编的角度阐述: 函数 `int sum(int x1,int x2,int x3,int x4,int x5,int x6,int x7,int x8)`, 调用和返回的过程中, 参数、返回值、控制是如何传递的? 并画出 `sum` 函数的栈帧 (X86-64 形式)。

- 参数传递顺序: `rdi,rsi,rcx,rdx,r8,r9`, 超过6个的部分用栈传递。返回值放在 `rax` 中。
- 指令 `call A` 等价于 `push rip + jmp A`。指令 `ret` 等价于 `pop rip`。因此函数调用必然要满足栈指针守恒性。
- 在教材默认的 `-Og` 优化下, 局部变量一般直接放在寄存器中 (快), 若寄存器放不下, 或者需要取局部变量地址, 或者是局部的数组或结构体, 需要放在栈帧中, 一般用 `rbp` 或者 `rsp` 访问它们。
- 总而言之函数调用要注意3个关键点: 传递控制, 传递数据, 内存的分配与释放。

3. 当函数调用时, (B) 可以在程序运行时动态地扩展和收缩。

A. 程序代码和数据区 B. 栈 C. 共享库 D. 内核虚拟存储区



10. 函数调用与栈帧（重点！）

- 对于栈帧内存储的局部变量的分析也可以是一个重要考点。

41. 简述 C 编译过程对非寄存器实现的 int 全局变量与非静态 int 局部变量处理的区别。包括存储区域、赋初值、生命周期、指令中寻址方式等。

答：

1 分

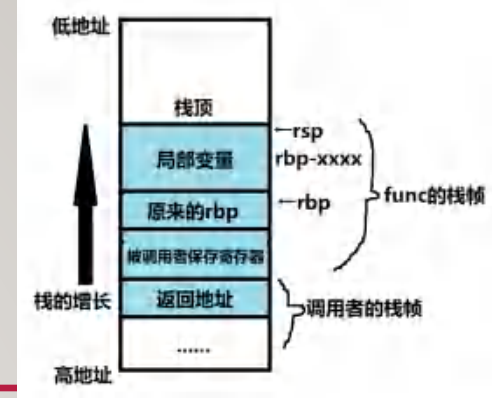
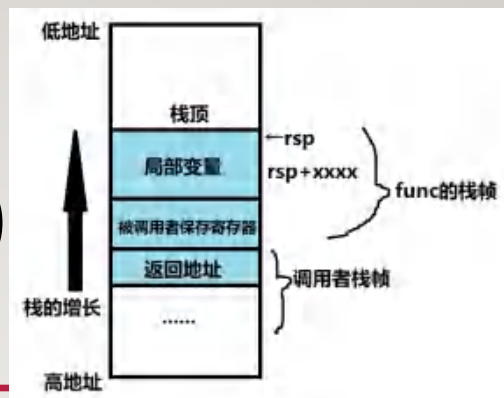
1 分

1 分

1 分

	int 全局变量	int 局部变量
存储区域	数据段	堆栈段
赋初值	编译时 <code>int x=1;</code>	程序执行时，执行数据传送类指令如 <code>MOVL \$1234, 8(RSP)</code>
生命周期	程序整个执行过程中都存在	进入子程序后在堆栈中存在(如执行 <code>subq \$8, %rsp</code>)子程序返回前清除消失
指令中寻址方式	其地址是个常数，寻址如 <code>movl 0x806808C, %eax</code>	通过 <code>rsp/rbp</code> 的寄存器相对寻址方式。如类似 <code>(%rsp)</code> 或 <code>8(%rsp)</code> 或 <code>-8(%rbp)</code> 等

10. 函数调用与栈帧 (重点!)



- 一般而言，函数的框架有以下两种形式：

每个局部变量/数组都对应着一个唯一的offset

```
func:
    pushq    用到的被调用者保存的寄存器
    subq     局部变量所占字节数, %rsp    //在栈上分配空间
    //此时可以使用"offset(%rsp)"的形式访问局部变量
    //函数体, do something...
    mov      返回值, rax(或eax)
    addq     局部变量所占字节数, %rsp    //释放栈帧, 还原rsp
    popq     用到的被调用者保存的寄存器
    ret
```

```
func:
    pushq    用到的被调用者保存的寄存器
    pushq    %rbp
    movq     %rsp, %rbp
    subq     局部变量所占字节数, %rsp    //在栈上分配空间
    //此时可以使用"-offset(%rbp)"的形式访问局部变量
    //函数体, do something...
    mov      返回值, %rax(或eax)
    movq     %rbp, %rsp //还原rsp, 释放栈帧
    popq     %rbp
    popq     用到的被调用者保存的寄存器
    ret
```

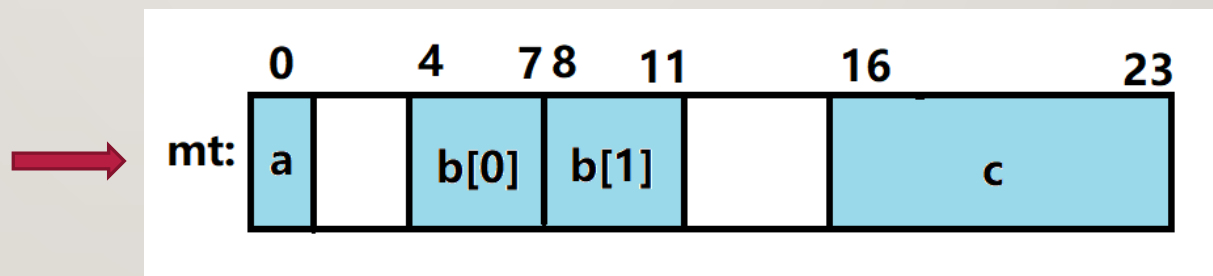
- 不过考试题中遇到的函数一般都比较简单，其中的大多数代码都可以省略。
- rbx、rbp、r12~r15需要被调用者保存（如果被调用者用到它们的话），r10、r11需要调用者保存（如果被用到的话），其余寄存器传递参数或返回值。

11. 数组与对齐的结构体

- 对于一维数组，只需要用“基址+下标*元素大小”寻址即可，例如：
- `movq 0x4031a0(,%rax,8), %rbx` （基址0x4031a0处有long数组a[], `rbx=a[rax]`）
- `movl (%rcx,%rax,4), %ebx` （基址rcx处有int数组b[], `ebx=b[rax]`）
- 对于结构体方法类似，“结构体基址+成员偏移”寻址。由于结构体内部成员类型长度良莠不齐，C语言编译器规定默认情况下长度为K的基本类型成员的偏移必须是K的倍数！（对齐）

例1.

```
struct mytype{
    char a;
    int b[2];
    char* c;
}mt;
```



**sizeof(mytype)=24,
而非17!**

22. C语言程序定义了结构体 `struct noname{char c; int n; short k; char *p;};` 若该程序编译成 64 位可执行程序，则 `sizeof(noname)` 的值是_____。

12.switch及其跳转表

- switch的关键就是其中的`jmpq`间接跳转指令（如果`case`的值过于离散，编译器将不得不一个一个`cmp`+条件跳转，不过一般不至于）
- 这个例子中的跳转表会存在`.rodata`节中的某个位置，0号元素是`case_0`地址，1号元素是`case_1`地址，2号元素是`case_2`地址，每个元素都是8字节的地址。

22. switch 语句的机器级实现中，采用的跳转表存在 elf 文件的 `.rodata` 节。

```
switch(rax){  
    case 0:  
        分支A;  
    case 1:  
        分支B;  
    case 2:  
        分支C;  
    default:  
        默认分支;  
}
```



```
    cmpq    $2, %rax  
    jg      case_default  
    cmpq    $0, %rax  
    jl      case_default  
    jmpq    *跳转表地址(,%rax,8)  
case_0:  
    分支A  
case_1:  
    分支B  
case_2:  
    分支C  
case_default:  
    默认分支
```


6. 缓冲器溢出漏洞防范方法错误的是 (C)。

A. 金丝雀 B. 使用安全函数 C. 加大局部变量占用空间 D. 编译加安全选项

13. 缓冲区溢出攻击

42. 简述缓冲区溢出攻击的原理以及防范方法。

24. 缓冲器溢出漏洞中，是用黑客程序的地址覆盖了 ____ 返回地址 ____ 完成的

- 不知为何几乎每年必考，把这些攻击原理和4条防范方法记住就行.....
- **攻击原理：**有一些需要在栈中开缓冲区（局部数组）的不安全的函数（例如gets、strcpy），它们在将数据复制进缓冲区的时候不会检查是否越界。那么就可以构造长度能够越界的数据，使得栈里包括函数返回地址在内的数据被覆盖，这样就可以替换函数的返回地址为黑客代码的地址！（关键是覆盖了栈里的返回地址）
- **防范方法：**①使用更加安全的函数（例如fgets、strncpy，它们能限制数据长度），②栈随机化策略，通过随机填充字节使栈空间地址在每次运行时都有变化，攻击者更难尝试出返回地址的区域，③基于金丝雀的栈破坏检测，在函数执行开始在栈里的返回地址前插入一个特殊值，返回前检查一下是否被破坏（可以通过编译选项设置），④限制可执行代码区域，通过分页机制将栈等数据区的数据标记为不可执行，防止攻击者插入代码。

14. 人肉反编译考试解读

- 例1. 简单题

46. 有下列 C 函数:

```
long arith(long x, long y, long z)
{
    long t1 = ____ (1) ____;
    long t2 = ____ (2) ____;
    long t3 = ____ (3) ____;
    long t4 = ____ (4) ____;
    ____ (5) ____;
}
```

请填写出上述 C 语言代码中缺失的部分

(1) $x \wedge y$ (2) $t1 + t1 \ll 2 / 5 * t1$ (3) $t2 + y \ll 1$
(4) $t3 - z$ (5) $\text{return } t4$ $4 * t1$ $2 * y$

//注意 64 位参数顺序 rdi、rsi、rdx, 这里 lea 只是把地址送 rax 而已, 相当于计算

函数 arith 的汇编代码如下:

```
arith:
xorq  %rsi, %rdi → t1
leaq  (%rdi, %rdi, 4), %rax → t2
leaq  (%rax, %rsi, 2), %rax → t3
subq  %rdx, %rax → t4
retq
```

(Handwritten notes: t1, t2, t3, t4, and a crossed-out ret)

14. 人肉反编译考试解读

参数p本身显然是指针（因为访问了它指向的内存单元），但它指向的东西较为怪异，在32位下用eax存，在64位下用rax存，这是什么类型？当然还是指针！

- 例2. 这题的关键在于分析eax/rax的怪异行为

47. 某 C 函数(函数体只有一条 C 语句)的 64 位与 32 位的反汇编结果分别如下：

```
4005d6: push    %rbp
4005d7: mov     %rsp,%rbp
4005da: mov     (%rdi,-0x8(%rbp)),%rax
4005de: mov     -0x8(%rbp),%rax
4005e2: mov     (%rax),%rax
4005e5: lea     0x4(%rax),%rcx
4005e9: mov     -0x8(%rbp),%rdx
4005ed: mov     %rcx,(%rdx)
4005f0: mov     (%rax),%eax
4005f2: pop     %rbp
4005f3: retq
```

eax说明返回

请写出 函数 f 的返回值类型 值是32位int，参数 p 的类型 int指针int**

函数体的唯一一条 C 语句 return *((*p)++);

```
804849b:  push    %ebp
804849c:  mov     %esp,%ebp
804849e:  mov     0x8(%ebp),%eax
80484a1:  mov     (%eax),%eax
80484a3:  lea     0x4(%eax),%ecx
80484a6:  mov     0x8(%ebp),%edx
80484a9:  mov     %ecx,(%edx)
80484ab:  mov     (%eax),%eax
80484ad:  pop     %ebp
80484ae:  ret
```

先让eax/rax=*p，说明p仍然指向一个指针，最终eax=**p，说明p是二重int指针int**

先取p指向的地址指向的int，再将p指向的地址加4，等价于令p指向的int指针进行一个++操作！

15.Y86-64处理器设计

50. 写出 Y86-64 CPU 顺序结构设计中 addq 指令各阶段的微操作。为 Y86-64 CPU 增加一条指令“mraddq rA,D(rB)”，能够将内存数据加到寄存器 rA。请参考 mrmovq, addq 指令，合理设计 mraddq rA,D(rB)指令在各阶段的微操作，并给出设计理由。(15 分)

指令 mraddq rA,D(rB)的编码规则如下

字节 0		字节 1		字节 2...9			
C	0	rA	rB	D			

考试必考一道这样的Y86微操作设计题，参照教材上这些示例的套路即可。微操作是比较抽象的，可以设计的较为随意。

- 在本课程描述的Y86-64处理器中，一条指令的处理需要通过6个阶段：取指、译码、执行、访存、写回、更新PC。这里是非流水的处理器，一条指令必须完完整整地通过6个阶段后，才能开始取下一条指令。关于流水线涉及到的考点参考计组的复习教程，都是差不多的。考试不会让你设计流水线。

irmovq rA, rB	2	0	rA	rB
lrmovq V, rB	3	0	F	rB
rmovq rA, D(rB)	4	0	rA	rB
mrmovq D(rB), rA	5	0	rA	rB
OPq rA, rB	6	0	fn	rA
jXX Dest	7	0	fn	Dest

阶段	OPq rA, rB	irmovq rA, rB	lrmovq V, rB
取指	icode, ifun ← M ₁ [PC] rA, rB ← M ₂ [PC+1] valP ← PC+2	icode, ifun ← M ₁ [PC] rA, rB ← M ₂ [PC+1] valP ← PC+2	icode, ifun ← M ₁ [PC] rA, rB ← M ₂ [PC+1] valC ← M ₃ [PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	加上指令长度
执行	valE ← valB OP valA Set CC	valE ← 0 + valA	valE ← 0 + valC
访存	只能在这个阶段访问存储器(不算取指的话)		
写回	R[rB] ← valE	R[rB] ← valE	R[rB] ← valE
更新 PC	PC ← valP	PC ← valP	PC ← valP

根据指令格式，取出包含在指令中的字段

从通用寄存器文件中取操作数值进行运算，得到指令执行结果

将结果写回寄存器文件

将PC设置为下一条指令地址
(只能在非流水线处理器中这样设计)

阶段	irmovq rA, D(rB)
取指	icode, ifun ← M ₁ [PC] rA, rB ← M ₂ [PC+1] valP ← M ₃ [PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]
执行	valE ← valB + valC
访存	M ₃ [valE] ← valA
写回	
更新 PC	PC ← valP

计算地址

阶段	pushq rA	popq rA
取指	icode, ifun ← M ₁ [PC] rA, rB ← M ₂ [PC+1] valP ← PC+2	icode, ifun ← M ₁ [PC] rA, rB ← M ₂ [PC+1] valP ← PC+2
译码	valA ← R[rA] valB ← R[rsp]	valA ← R[rsp] valB ← R[rsp]
执行	valE ← valB + (-8)	valE ← valB + 8
访存	M ₃ [valE] ← valA	valE ← M ₃ [valA]
写回	R[rsp] ← valE	R[rsp] ← valE R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

阶段	jXX Dest	call Dest	ret
取指	icode, ifun ← M ₁ [PC] valC ← M ₂ [PC+1] valP ← PC+9	icode, ifun ← M ₁ [PC] valC ← M ₂ [PC+1] valP ← PC+9	icode, ifun ← M ₁ [PC] valP ← PC+1
译码		valB ← R[rsp]	valA ← R[rsp] valB ← R[rsp]
执行	只能在执行时处理操作码 Cnd ← Cond(CC, ifun)	valE ← valB + (-8)	valE ← valB + 8
访存		M ₃ [valE] ← valP	valM ← M ₃ [valA]
写回		R[rsp] ← valE	R[rsp] ← valE
更新 PC	PC ← Cnd?valC:valP	PC ← valC	PC ← valM

16. 程序优化 (这些方法在考试时需要能列举出)

- 将不必要重复使用的代码移出循环，减少运算量
- 减少过程调用，使用inline函数
- 用简单计算代替复杂计算(如移位代替乘除法)
- 公共子表达式重用，减少重复计算
- 使用局部变量作为累积量
- 用条件表达式(条件数据传送指令)取代分支结构
- 循环展开，减少循环跳转次数
- 消除不必要的内存引用，减少访存
- 多个累积变量，利于指令级并行
- 使用SIMD指令

17.Cache

- 程序具有时间局部性(一个存储单元会在不久后再次使用)和空间局部性(使用一个存储单元后其临近的存储单元会在不久之后被使用)。

- 存储器层次结构:



上一级是下一级的缓存

10. 位于存储器层次结构中的最顶部的是 (A)。
A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存

- 缓存的3种不命中：冷不命中(缓存中一开始一定没有对应的块)，冲突不命中(访问了与该块映射到同一位置的其它块导致该块被替换出去，但缓存未满不存在容量问题)，容量不命中(访问的内存区域太大而缓存容量有限，块被迫频繁替换)。

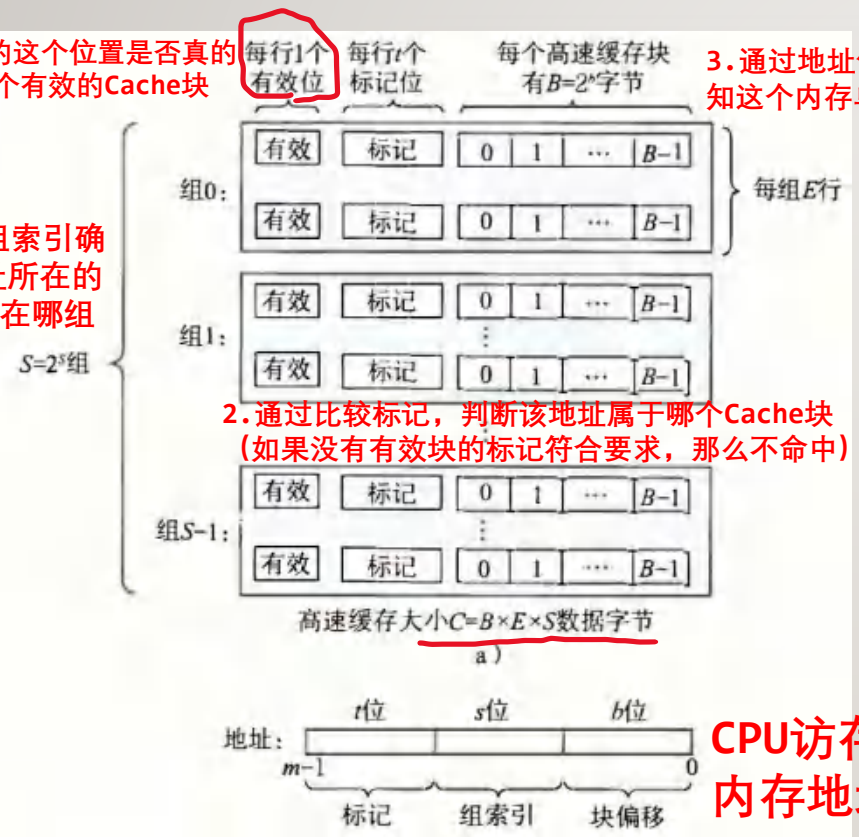
17.Cache

48. 某CPU的L1 cache容量32kb, 64B/块, 采用8路组相连, 物理地址47位。试分析其结构参数B、S、E分别是多少? 地址0x00007f6635201010访问该L1时, 其块偏移C0、组索引C1、标记C2分别多少? (5分)

块大小 $B=64$, 块内偏移 $b=6$ 位
组数 $S=32KB/(64B*8)=64$, 组索引 $s=6$ 位
标记 $t=47-6-6=35$ 位

Cache中的这个位置是否真的对应了一个有效的Cache块

1. 通过组索引确定该地址所在的Cache块在哪组



2. 通过比较标记, 判断该地址属于哪个Cache块 (如果没有有效块的标记符合要求, 那么不命中)

3. 通过地址低位的块内偏移, 得知这个内存单元在块的哪个位置

基于这种“比较地址”模式访问的存储器称为相联存储器, 若Cache的组中有超过1行(块), 那么便是组相联Cache, 否则就是直接映射Cache, 很容易发生冲突不命中, 但访问速度快(只做1次比较)。若整个Cache只有1组, 对Cache的访问完全进行标记比较, 便是全相联Cache, 不命中率最低(但硬件实现代价最高, 速度最慢)。

一般所说的“ n 路组相联Cache”指的是每组有 n 行(块), 即 $E=n$ 。考试时会用这个让你来回倒腾块大小、容量、组数、组索引位数、标记位数、地址位数等参数的计算

CPU访存时给出的物理内存地址格式(重要!)

18.Cache与性能优化

44. 简述程序的局部性原理，如何编写局部性好的程序？

答：局部性原理：1分 程序倾向于使用与最近使用过数据的地址接近或是相同的的数据和指令。时间局部性：最近引用的项很可能在不久的将来再次被引用，如代码和变量等；空间局部性：与被引用项相邻的项有可能在不久的将来再次被引用。

2分让通用或共享的功能或函数—最常见情况运行得快：对于多线程/多处理器/多核。

2分尽量减少每个循环内部的缓存不命中数量。反复引用变量是好的(时间局部性)

—寄存器—编译器：把为1的 参考模式是好的(空间局部性)—缓存是连续块

一旦从内存中装入数据对象，尽可能多的使用它，使得程序中时间局部性最大。

- 程序需要尽可能增大自身局部性，减少Cache不命中率，提升访存效率。(尤其多维数组)

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

访问数组不连续，空间局部性差，应该交换内外循环

调整内外循环顺序

```
code/mem/matmult/mm.c
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++) {
3         sum = 0.0;
4         for (k = 0; k < n; k++)
5             sum += A[i][k]*B[k][j];
6         C[i][j] += sum;
7     }
```

```
code/mem/matmult/mm.c
1 for (k = 0; k < n; k++)
2     for (i = 0; i < n; i++) {
3         r = A[i][k];
4         for (j = 0; j < n; j++)
5             C[i][j] += r*B[k][j];
6     }
```

50. 程序优化：矩阵 $c[n,n] = a[n,n] * b[n,n]$ ，采用 48 题 I7 CPU。块 64B。

```
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
    {
        c[i][j]=0;
        for(int k=0;k<n;k++)
            c[i][j]+=a[i][k]*b[k][j];
    }
```

此题考试必考

请针对该程序进行速度优化，写出优化后的程序，并说明优化的依据。

对矩阵分块，使得同一块被频繁访问，时间局部性更好

```
for(int i=0; i < n; i += B)
    for(int j=0; j < n; j += B)
        for(int k=0; k < n; k += B)
            for(int i1=0; i1 < B; i1++)
                for(int k1=0; k1 < B; k1++)
                    for(int j1=0; j1 < B; j1++)
                        C[i + i1][j + j1] += A[i + i1][k + k1] * B[k + k1][j + j1];
```

19. 从编译到链接

1. 在 Linux 系统中利用 GCC 作为编译器驱动程序时，能够将汇编程序翻译成可重定位目标程序的程序是（ ）

A. cpp

B. cc1

C. as

D. ld

- C语言程序从源代码到运行的全过程：①C预处理器(cpp)进行预处理，②C编译器(cc1)进行编译，得到汇编语言文件.s，③汇编器(as)进行汇编，翻译为二进制指令数据，得到可重定位目标文件.o，④链接器(ld)将目标文件和其它目标文件进行链接，得到可执行目标文件，⑤用户在shell里输入命令，shell调用加载器将可执行目标文件加载到内存，执行程序。
- 链接是对不同代码模块的组合，可以在编译时、加载时、执行时进行。在加载前进行的链接是静态链接。
- 静态链接的两大任务：符号解析，重定位。

20. 符号解析

```
/* main2.c */
/* $begin main2 */
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("x = [%d %d]\n", z[0], z[1]);
    return 0;
}
/* $end main2 */

/* addvec.c */
/* $begin addvec */
int addcnt = 0;

void addvec(int *x, int *y,
            int *z, int n)
{
    int i;

    addcnt++;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
/* $end addvec */
```

10. 链接过程中，赋初值的非静态全局变量名，属于（A）

A. 强符号 B. 弱符号 C. 可能是强符号也可能是弱符号 D. 以上都错

45. 请指出 addvec.c main2.c 中哪些是全局符号？哪些是强符号？哪些是弱符号？以及这些符号经链接后在哪个节？（5分）

- 可重定位目标文件中声明的3种符号：（对符号的引用需要知道其实际地址）
 - 全局符号：自己定义，可以被其它模块引用。对应非静态函数和全局变量。
 - 对于全局符号，函数和已初始化的全局变量是强符号，未初始化全局变量是弱符号。
 - 外部符号：其它模块定义，自己引用。对应extern函数与变量。
 - 局部符号（本地符号）：自己定义，其它模块不可引用。对应static函数与变量。

模块1

全局符号

```
int global_var;

int func(int a){
    return a * 2;
}
```

模块2

外部符号

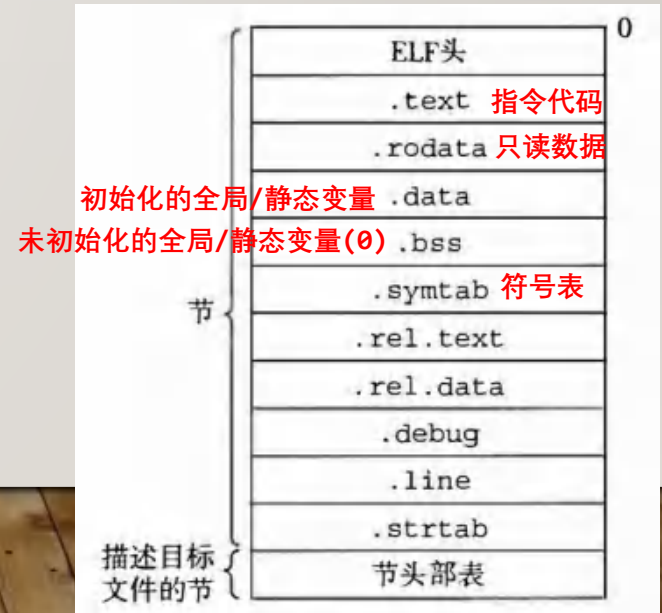
```
extern int global_var;
extern int func(int);

int func1(int b, int c){
    static int d=0;
    d = d + func(b) + c;
    return d;
}
```

局部符号

static变量存储在.data/.bss内存区，需要符号


纯粹的局部变量在栈/寄存器里存储，不涉及符号



20. 符号解析

- 在符号解析时，对于多个同名符号，需要在引用这个名字时进行选择。ld不允许多个同名强符号，若存在一个强符号则选择强符号。否则就从多个弱符号里任选一个。
- 静态库**是多个可重定位目标文件的集合，链接时从静态库中提取被引用符号的模块进行链接，这对链接时的命令行顺序有较高要求。
 - 链接器顺序扫描命令行输入的目标文件与静态库，扫描到目标文件时，可以使用之前得到的所有模块定义的符号解析，将该目标文件中暂时无法解析的外部符号放入U。扫描到静态库时，通过U得知该静态库的哪些目标文件需要参与连接，提取它们。

在命令行中，静态库应该被放在目标文件之后，而静态库之间可能还存在需要解决的依赖(引用)关系

 练习题 7.3 a 和 b 表示当前目录中的目标模块或者静态库，而 $a \rightarrow b$ 表示 a 依赖于 b，也就是说 b 定义了一个被 a 引用的符号。对于下面每种场景，请给出最小的命令行(即一个含有最少数量的目标文件和库参数的命令)，使得静态链接器能解析所有的符号引用。

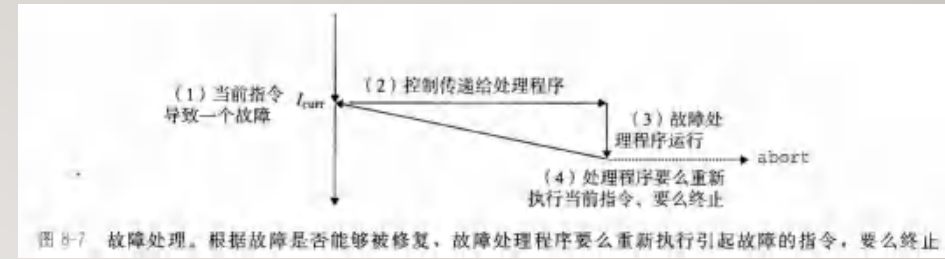
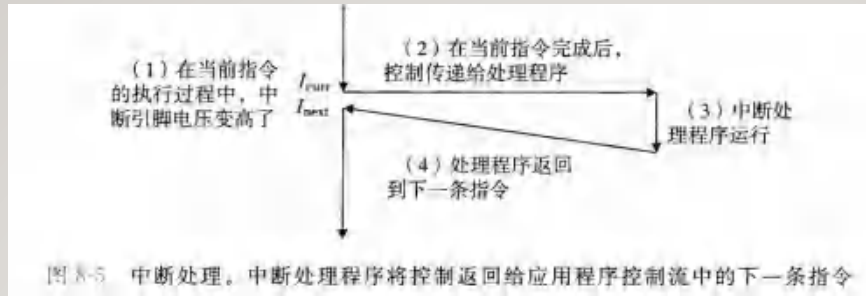
A. $p.o \rightarrow libx.a$ **ld p.o libx.a**

B. $p.o \rightarrow libx.a \rightarrow liby.a$ **ld p.o libx.a liby.a**

C. $p.o \rightarrow libx.a \rightarrow liby.a$ 且 $liby.a \rightarrow libx.a \rightarrow p.o$ **ld p.o libx.a liby.a libx.a**

这里p.o不需要再写一遍.....因为它不是静态库，没有“选择性提取”这回事

21. 异常



- 四种异常：中断（异步）、陷阱（同步）、故障（同步）、终止（同步）。一定要搞清楚异步和同步的区别！

18. 同步异常不包括 ()
A. 终止 B. 陷阱 C. ~~中止~~ D. 故障

19. 属于异步异常的是 ()
A. 中断 B. 陷阱 C. 故障 D. 终止

- 异常发生时，自动将特权级切换到内核态，此时能够访问一切系统资源！执行内核的异常处理程序，异常处理程序结束后切换回用户态。

15. 进程从用户模式进入内核模式的方法不包括 ()
A. 中断 B. 陷阱 C. ~~系统调用~~ D. 故障

25. CPU 在执行异常处理程序时其模式为 内核模式/超级用户模式。

34. () 异常处理程序运行在内核模式下，对所有的系统资源都有完全的访问权限。
✓

- 对于中断和陷阱，异常处理结束后返回执行下一条指令。对于终止，不返回（直接寄掉）。对于故障，可能不返回（太严重，寄掉），或者处理程序处理好了故障，返回后重新执行当前指令（如缺页故障）。

9. 下列异常中经异常处理后能够返回到异常发生时的指令处的是 ()
A. I/O 中断 B. 陷阱 C. ~~故障~~ D. 终止

5. 下列异常中可能从异常处理返回也可能不返回的是 ()
A. I/O 中断 B. 陷阱 C. 故障 D. 终止

19. 关于异常处理后返回的叙述，错误的叙述是 ()
A. 中断处理结束后，会返回到下一条指令执行
B. 故障处理结束后，会返回到下一条指令执行
C. 陷阱处理结束后，会返回到下一条指令执行
D. 终止异常，不会返回

21. 异常

- 每个异常都有一个编号，在主存的某个特殊位置有个**异常表**，当某个编号的异常发生时，就去异常表中找对应的处理程序地址，进行带有特权级切换的间接转移。
- **中断**：外部硬件（鼠标键盘时钟等）突然发来信号，计算机必须立刻处理它们。
- **陷阱**：使用类似call（但更复杂）的int n主动触发，是主动从用户态进入内核态的几乎唯一方法。**系统调用**就是最典型的陷阱（int 0x80），要区分系统调用的编号（表示0x80号异常处理程序的子功能）和异常号（0x80）！
17. 调用 read() 函数产生 () 异常。
A. 故障 B. 陷阱 C. 异步异常 D. 进程切换
37. (X) Linux 系统调用中的功能号 n 就是异常号 n 。
- **故障**：这条指令并不能被正常执行（例如访存时缺页），可能需要内核程序进行修复（加载页），修复不了就不返回，修复完成就重新执行一次该指令。
- **终止**：产生严重问题，没啥好说的。

22. 虚拟内存机制

- 物理地址：数据真正存在主存DRAM中的位置，通过地址信号线发送给主存。
- 虚拟地址（线性地址）：程序员用来访问一个内存单元的整数地址。
- 逻辑地址：“段地址:偏移地址”，linux下实质上就是虚拟地址。
- 虚拟内存机制本质上是利用主存和磁盘，实现了一个从虚拟地址到物理地址的映射。它可以：①作为缓存，高效使用主存，仅在主存中缓存活动区域。②作为内存管理，为每个进程提供一致的地址空间，并且保护其不被其它进程破坏。
- 虚拟内存机制是由分页机制实现的。

39. (✓) 虚拟内存系统能有效工作的前提是软件系统具有“局部性”。

21. 在计算机存储层次结构中，主存是磁盘的缓存。

16. 操作系统提供的抽象表示中，(B)是对主存和磁盘 I/O 设备的抽象表示。
A. 进程 B. 虚拟存储器 C. 文件 D. 虚拟机

22. 虚拟内存机制

- 我们把物理主存分割成若干个大小一致（一般4KB，可在CPU的控制寄存器中设置）的物理页，同样地，虚拟地址空间也会分割成大小与物理页一致的虚拟页。任意一个虚拟页有三种状态：未分配的（这个页没有意义）、缓存的（对应一个物理页）、未缓存的（存在磁盘上）。

5. 记录内存物理页面与虚拟页面映射关系的是（ ）
A. 磁盘控制器 B. 编译器 C. 虚拟内存 D. 页表

- 每个虚拟页都有个页号（就是起始地址的高若干位），通过这个页号可以去主存中的页表里查询它对应的页表条目PTE，PTE里记录了这个虚拟页是否是缓存的，对应的物理页（如果是缓存的）或者磁盘块（如果未缓存）在哪，是否只读/读写/不可执行，访问所需的特权级，是否采取写时复制策略等等，页表上的这些东西都是操作系统设置的，这些能够使得操作系统对虚拟页进行安全可靠且高效的管理，并且实现虚拟页到物理页的映射。

25. 虚拟内存系统借助____页表____这一数据结构将虚拟页映射到物理页。

15. 虚拟内存系统中的虚拟地址与物理地址之间的关系是（ B ）
A. 1对1 B. 多对1 C. 1对多 D. 多对多

12. 虚拟内存页面不可能处于（ ）状态
A. 未分配、未载入物理内存 B. 未分配但已经载入物理内存
C. 已分配、未载入物理内存 D. 已分配、已经载入物理内存

22. 虚拟内存机制

17. 虚拟内存发生缺页时，缺页中断是由（ D ）触发
A. 内存 B. Cache L1 C. Cache L2 D. MMU

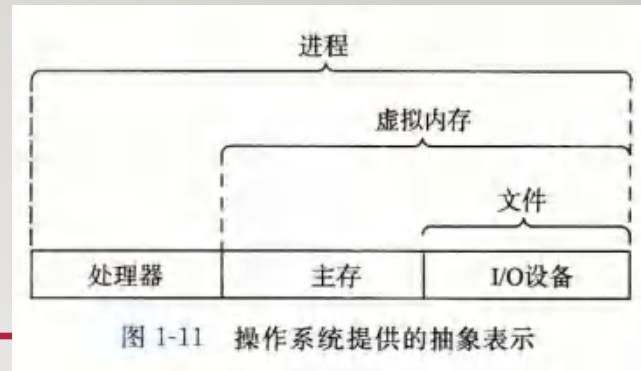
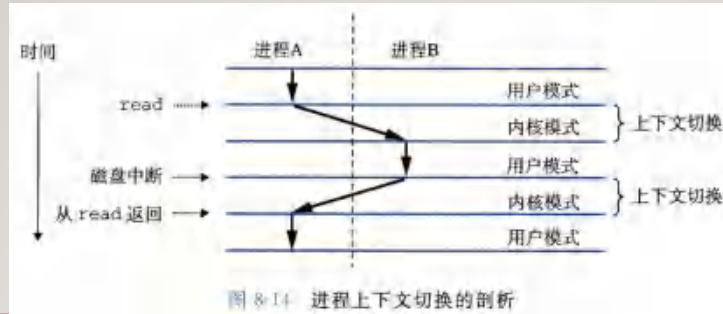
14. 虚拟内存发生缺页时，正确的叙述是（ ）触发的
A. 缺页异常处理完成后，重新执行引发缺页的指令
B. 缺页异常处理完成后，不需要重新执行引发缺页的指令
C. 缺页异常都会导致程序退出
D. 中断由 MMU 触发

30. 虚拟内存发生缺页时，MMU 将触发_____。

- 当一条访问内存的指令执行时，这条指令会将虚拟地址提供给CPU，CPU会通过地址翻译单元MMU查询虚拟地址对应的虚拟页号（取虚拟地址的高若干位）在主存页表中对应的PTE，如果，这个页未分配，或者违反了PTE上的访问权限管理，那么MMU将触发一般保护故障（段错误，不返回的故障）。如果这个页分配但未缓存，那么MMU将触发缺页故障，即页不命中，内核的处理程序采取页替换算法将主存里的某个物理页替换到磁盘里，并从磁盘内将目标页加载到这个物理位置，并修改页表，返回后重新执行内存访问指令。总之，MMU要通过页表得到虚拟地址对应的物理页，再访问物理页上对应页偏移（即地址的低若干位）的内存单元。
- 道理和Cache类似，并且相当于一种全相联的缓存！（因为任意一个虚拟页都可以对应到任意一个物理页上，只要设置页表。这与直接映射不同）

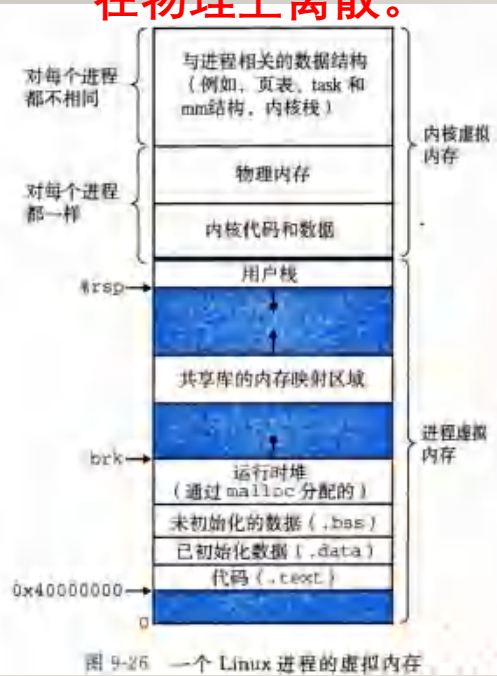
38. （ ）如果系统中程序的工作集大小超过物理内存大小，虚拟内存系统会产生抖动：页面不断地换进换出，导致系统性能暴跌。 **时间局部性差**

23. 进程



彼此独立，但可以共享部分页面。
看似连续，实则在物理上离散。

- 进程是一个执行中程序的实例，它是对处理器、主存、IO设备的抽象，运行中的程序在表面上好像通过进程独占它们。
- 通过基于分页的虚拟内存机制，操作系统能够为每个进程准备独立的虚拟地址空间。
3. 在进程的虚拟地址空间中，用户代码不能直接访问的区域是（D）
A. 程序代码和数据区 B. 栈 C. 共享库 D. 内核虚拟内存区
- 每个进程都具有上下文，决定了进程的程序运行状态，包括各种通用寄存器、状态寄存器、用户栈指针、页表（决定了虚拟地址空间）、文件表（进程打开的文件）、进程表（进程信息）等。
12. 下列不属于进程上下文的是（C）
A. 页全局目录 pgd B. 通用寄存器 C. 内核代码 D. 用户栈
9. 内核为每个进程保存上下文用于进程的调度，不属于进程上下文的是（A）
A. 全局变量值 B. 寄存器 C. 虚拟内存一级页表指针 D. 文件表
10. 内核为每个进程维持一个上下文，不属于进程上下文的是（D）
A. 寄存器 B. 进程表 C. 页表 D. 调度程序
- 系统内核会通过异常（时钟中断等）使得当前占用处理器的进程暂时休眠，保存进程的上下文，通过调度程序选择另一个进程，恢复该进程的上下文然后运行。这便是进程切换。某些需要长时间等待的系统调用（read、sleep等）或者硬件中断（磁盘、定时器（**主要**））都能引起进程切换。进程切换必须有下一个进程运行！



38. （√）进程在进行上下文切换时一定会运行内核函数。

19. 进程上下文切换不会发生在如下（D）情况
A. 当前进程时间片用尽 B. 外部硬件中断
C. 当前进程调用系统调用 D. 当前进程发送了某个信号

24. fork与子进程

子进程中的“返回”实际上是子进程“获得新生”！

- Linux中除了根进程以外的进程都是由某个进程通过fork()自我复制而来的。在成功的情况下fork()调用一次，返回两次（父子进程中各返回一次），通过fork()的返回值区分父子进程，并且它们也有不同的进程编号。

26. fork 后创建的子进程与父进程不同的信息是_____进程 ID_____。

- fork()会将子进程的虚拟页映射到父进程的页上，有的内存区二者共享（例如共享库），更多的内存区理论上应由进程分别私有，linux使用写时复制策略，使用这些页暂且在PTE上标记为只读，当其中一个进程尝试修改该页时，引发故障，内核为进程复制一份私有的页用来修改。这样既保证了内存空间的独立，又尽量地通过共用页提高了空间效率。

38. ~~(X)~~ fork 的子进程与其父进程同名的全局变量始终对应同一物理地址。

27. ~~(X)~~ 当执行 fork 函数时，内核为新进程创建虚拟内存并标记内存区域为私有的写时复制，意味着新进程此时获得了独立的物理页面。

- 针对带有fork的程序，我们应当能够分析父子进程的行为，并画出进程图。（会出大题！）

24. fork与子进程-分析题

49. C 程序如下，请画出对应的进程图，并回答父进程和子进程分别输出什么？

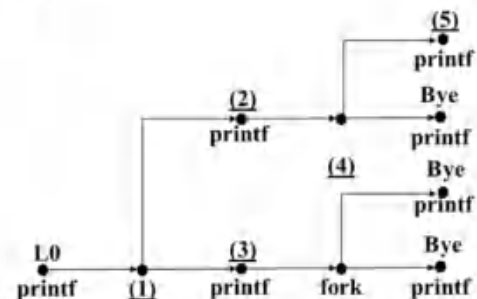
```
int main()
{
    int x = 1;
    if(Fork() != 0)
        printf("p1: x=%d\n", ++x);
    printf("p2: x=%d\n", --x);
    exit(0);
}
```

```
1  int main()
2  {
3      int status; A. 这个程序会产生多少输出行?
4      pid_t pid; B. 这些输出行的一种可能的顺序是什么?
5
6      printf("Hello\n");
7      pid = Fork();
8      printf("%d\n", !pid);
9      if (pid != 0) {
10         if (waitpid(-1, &status, 0) > 0) {
11             if (WIFEXITED(status) != 0)
12                 printf("%d\n", WEXITSTATUS(status));
13         }
14     }
15     printf("Bye\n");
16     exit(2);
17 }
```

48. C 程序 fork2 的源程序与进程图如下：

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

注意嵌套
的fork!



请写出上述进程图中空白处的内容

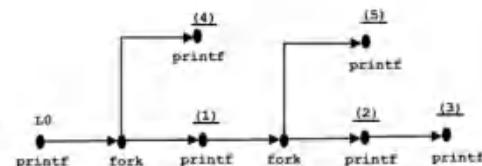
(1) fork (2) L1 (3) L1

(4) fork (5) Bye

48. C 程序 forkB 的源程序与进程图如下：

```
void forkB()
{
```

```
    printf("L0\n");
    if(fork() != 0) {
        printf("L1\n");
        if(fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



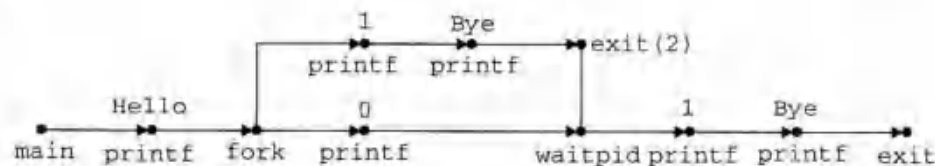
请写出上述进程图中空白处的内容

(1) L1 (2) L2 (3) Bye

(4) Bye (5) Bye

A. 只简单地计算进程图(图 8-49)中 printf 顶点的个数就能确定输出行数。在这里，有 6 个这样的顶点，因此程序会打印 6 行输出。

B. 任何对应进程图的拓扑排序的输出序列都是可能的。例如：Hello、1、0、Bye、2、Bye 是可能的。



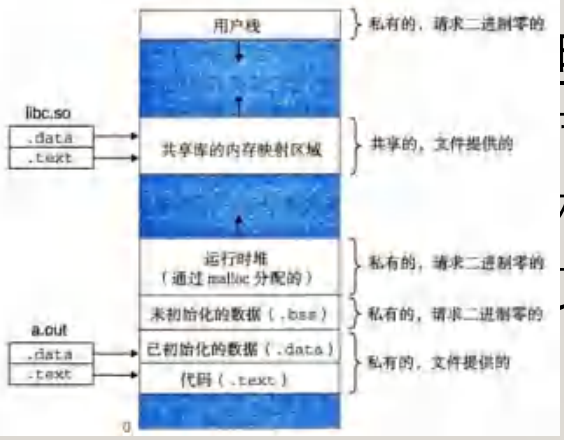
25. execve与内存映射

- 系统调用execve将一个可执行文件加载到当前进程的上下文中，替换掉原来的程序。execve调用成功后不返回（因为原程序无了，原有的用户区域被execve删掉了），也不创建新的进程。

13. 下列函数中属于系统调用且在调用成功后，不返回的是()
A. fork B. execve C. setjmp D. longjmp

38. (✓) execve 加载新程序时会覆盖当前进程的地址空间，但不创建新进程。

- execve实际上并不会直接把可执行文件的内容复制到物理内存，而是使用了内存映射机制，代码和数据区直接映射到可执行文件中的.text、.data、.rodata等节（普通文件的内存映射，并且写时复制），等到用的时候缺页了再加载。.bss、栈、堆都将映射到匿名文件，它们不会从磁盘中加载页，而是缺页时由内核直接填充二进制0。栈和堆的初始长度都是0（空），.bss的长度由可执行文件指定。



16. 程序语句“execve(“a.out”, NULL, NULL);”在当前进程中加载并运行可执行文件 a.out 时, 所做的叙述是 ()
A. 为代码、数据、.bss 和栈创建新的, 私有的, 写时复制的区域结构
B. .bss 区域是请求二进制零的, 映射到匿名文件, 初始长度为 0
C. 堆区域也是请求二进制零的, 映射到匿名文件, 初始长度为 0
D. 栈区域也是请求二进制零的, 映射到匿名文件, 初始长度为 0

26. linux 虚拟内存区域可以映射到普通文件和__匿名文件__, 这两种类型的对象中的一种。

- 可见，execve做了如此多的工作，可执行文件加载时不可能就占这么几页内存！

20. Hello World! 执行程序占 3186 字节，运行时占(D)页内存
A. 1 B. 2 C. 4 D. > 4

17. “Hello World”执行程序很小不到 4k，在其首次执行时产生缺页中断次数 ()
A. 0 B. 1 C. 2 D. 多于 2 次

25.execve与内存映射

```
int main(int argc, char **argv)
{
    int i, secs;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <n>\n", argv[0]);
        exit(0);
    }
    secs = atoi(argv[1]);
    for (i=0; i < secs; i++)
        sleep(1);
    exit(0);
}
```

./myspin 秒数

- `execve`会将参数列表`argv[]`以及环境变量列表`envp[]`传递给加载的程序，程序可以使用带参数的`main`接受它们。这两个列表内的元素都是字符串指针，以NULL结尾。
- `argv`实际上相当于用户在shell内输入的命令行按空格分隔而成的字符串们，例如命令行“`./me arg1 arg2`”得到的`argv[]`就是“me”，“arg1”，“arg2”，`argv[0]`为程序名。

46. 某C程序(64位模式)的main函数参数argv地址为0x000413433323110，其内容如下：

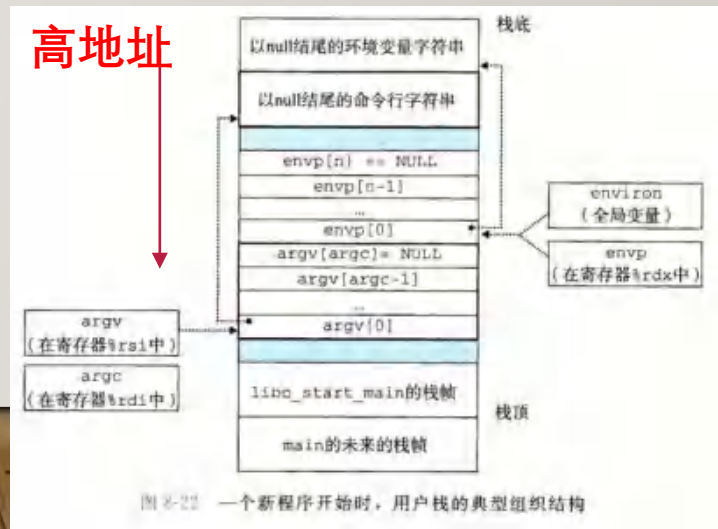
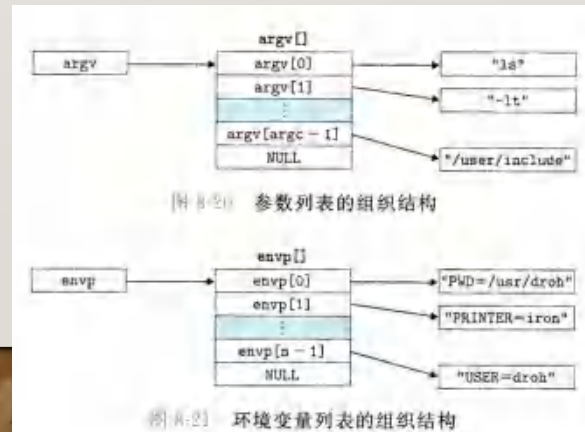
```
30 31 32 33 34 41 00 00 33 31 32 33 34 41 00 00
35 31 32 33 34 41 00 00 00 00 00 00 00 00 00 00
31 43 00 30 00 32 42 00 38 00 31 31 32 32 00 30
32 33 00 61 41 00 31 00 32 00 33 00 31 00 00 31
```

请写出 程序名：_____，本程序的参数个数_____

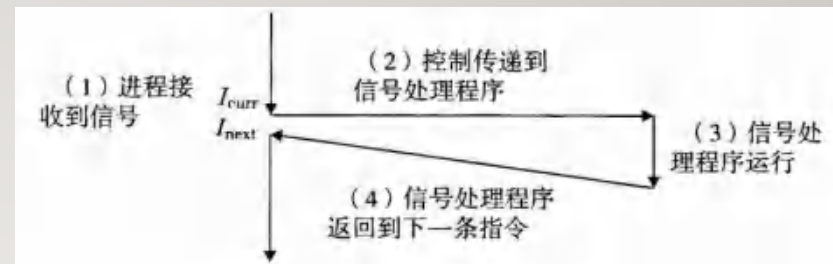
按顺序写出各个参数为_____

14. 如下数据在内存中地址最高的是 (B)

A. 命令行参数 B. 环境变量 C. main 的栈帧 D. 当前子程序栈帧



26. 信号



- 信号是linux下进程与进程或者内核与进程的消息传递，它与硬件没什么关系，完全是软件层面实现的。信号的发送必须借助内核！当一个信号被发送到进程A时，此时控制流一定在内核，内核会为进程A打上标记。当内核打算切换回进程A时，发现它被发送了待处理的信号，那么会执行默认操作，或者在用户态调用进程指定的信号处理程序。信号不可排队（或者计数），只能说被发送或者未被发送。
- 当一个信号被阻塞的时候，如果这种信号在某时刻发送到该进程，那么它不会被立刻处理，等信号解除阻塞的时候再被处理。这段时间内发来的多个这种信号视为一个（不排队！）。
- 应用程序可以使用系统调用kill()发送信号，使用signal()为信号指定处理程序，使用sigprocmask()阻塞指定的信号。用户可以在终端里用kill命令发送信号。

30. 向指定进程发送信号的 linux 命令是_____。

26. 信号

- 很多信号是异步的（比如Ctrl+C发送的SIGINT、其它进程发来的信号），但这种异步性是依赖于中断的。实际上，任何信号都需要依赖于异常，毕竟一定有内核与用户的切换。
- **例1.** 进程A正在运行，此时人类按下Ctrl+C，引发键盘中断（*异步异常*），内核的处理程序识别到Ctrl+C，由内核向进程A发送信号SIGINT，默认处理行为使得进程A终止。
- **例2.** 进程A正在运行，“此时”另一个进程B通过kill向进程A发送信号SIGUSR1，在进程A看来，它刚执行完语句1，正想执行语句2，控制就莫名其妙的转入SIGUSR1的处理程序了。**实际上的情况是**，进程A执行完1后，发生定时器中断，内核将进程A切换到其它进程。等到进程B运行时，它通过kill令内核代为执行“B给A发信号”这个动作，等到一段时间后，内核要切换到进程A，发现进程A有待处理的信号，那么就先切换到用户态，但控制传递给处理程序。
- **例3.** 44. 程序执行 `int x=y/c` 语句时，当 `c=0` 时程序执行结果是什么？并请结合异常、信号的概念及处理机制解释原因。

16. C 程序执行到整数或浮点变量除以 0 可能发生（ D ）
A. 显示除法溢出错直接退出 B. 程序不提示任何错误
C. 可由用户程序确定处理方法 D. 以上都可能

26. 信号

- 我们应该起码认识这些信号，考试的时候可能会考它们的默认行为与作用：
- SIGINT: 终止进程，可以由Ctrl+C向前台进程发送。
- SIGTSTP: 停止进程，进程处于暂停状态。可以由Ctrl+Z向前台进程发送。
- SIGCONT: 使停止的进程开始继续运行。
- SIGCHLD: 当子进程停止或终止时向父进程发送，默认什么都不做。
- SIGSEGV: 当发生内存非法访问（一般保护故障，段错误）时由内核发送给进程。默认直接终止进程。
- SIGFPE: 浮点异常，或者发生了整数除0故障，由内核发送给进程，默认直接终止。

16. 程序运行中按下Ctrl-Z,会发生 (B) =
A. 当前进程终止 B. 当前进程停止 C. 父进程停止 D. 父进程终止

17. 每个信号类型都有一个预定义的默认行为,可能是 (D)
A. 进程终止 B. 进程挂起直到被SIGCONT重启 C. 进程忽略该信号 D. 以上都是

18. 一个子进程终止或者停止时,操作系统内核会发送 (D) 信号给父进程。
A. SIGKILL B. SIGQUIT C. SIGSTOP D. SIGCHLD

27. 进程的终止与回收

20. Linux 进程终止的原因可能是 (D)

A. 收到一个信号 B. 从主程序返回 C. 执行 exit 函数 D. 以上都是

- 进程终止时会产生一个退出状态码，一般会有3种情况：①main函数return了一个状态码，②程序调用exit传入状态码，③进程接收到一个信号并执行终止的默认行为，此时也会自动设置退出状态码。

10. 导致进程终止的原因不包括 ()

A. 收到一个信号 B. 执行 wait 函数 C. 从主程序返回 D. 执行 exit 函数

- 当一个进程终止后，它的内存资源并不释放，仍然留在内存中，称为僵死进程，它需要被父进程使用waitpid或者wait回收，如果父进程长时间工作的话必须及时回收终止的子进程，避免浪费资源。

37. (x) 进程一旦终止就不再占用内存资源。

36. (✓) 子进程即便运行结束，父进程也应该使用 wait 或 waitpid 对其进行回收。

- waitpid会让当前程序暂时等待任意一个（或者指定的）子进程停止或终止，若为终止的子进程则对其进行回收，释放资源，并将退出状态传递给父进程。

- waitpid很多情况下用在SIGCHLD的信号处理中，但要注意不要来一个信号就仅回收一次！

这个SIGCHLD的处理程序可能会造成未回收的僵死子进程！

```
void handler1(int sig)
{
    int olderrno = errno;

    if ((waitpid(-1, NULL, 0)) < 0)
        sig_error("waitpid error");
    Sig_puts("Handler reaped child\n");
    Sleep(1);
    errno = olderrno;
}
```

```
void handler2(int sig)
{
    int olderrno = errno;

    while (waitpid(-1, NULL, 0) > 0) {
        Sig_puts("Handler reaped child\n");
    }
    if (errno != ECHILD)
        sig_error("waitpid error");
    Sleep(1);
    errno = olderrno;
}
```


28. 并发

18. 三个进程其开始和结束时间如下表所示，则说法正确的是()

进程	开始时刻	结束时刻
P1	1	5
P2	2	8
P3	6	7

- A. P1、P2、P3 都是并发执行 B. 只有 P1 和 P2 是并发执行
C. 只有 P2 和 P3 是并发执行 D. P1 和 P2、P2 和 P3 都是并发执行

- 一个逻辑流（程序）在执行时间上与另一个逻辑流重叠，那么它们并发地运行。特别地，如果这两个流运行在不同的处理器核上，那么它们是并行的。
- 不同的进程是并发的，信号处理程序和主程序也是并发的（主要由于其异步性），如果它们共享相同的资源，容易造成数据竞争，需要小心处理。特别是信号处理程序，它与主程序共享相同的全局变量。
- 并发运行的两个进程如果都进行输出操作，由于进程切换，输出顺序将是不确定的，可能需要分析所有可能情况。信号处理程序也会有类似情况。
- 由于信号处理程序的并发性，需要通过信号阻塞等方式避免与信号处理程序产生数据竞争，并且在信号处理程序中使用可重入的（不会使用全局数据的）异步信号安全的函数（Unix I/O-系统提供的read/write，而非标准库的scanf/printf）。

36. (X) 进程是并发执行的，所以能够并发执行的都是进程。

40. () 相比标准 I/O，Unix I/O 函数是异步信号安全的，可以在信号处理程序中安全地使用。

11. 异步信号安全的函数要是可重入的（如只访问局部变量）要么不能被信号处理程序中断，包括 I/O 函数 ()

- A. printf B. sprintf C. write D. malloc

28. 并发

LAB4里用到的使用信号阻塞 来避免数据竞争的保护方式

```
/* block signals before operate on jobs list */
if (sigemptyset(&mask) < 0)
    unix_error("sigemptyset error");
if (sigaddset(&mask, SIGCHLD))
    unix_error("sigaddset error");
if (sigaddset(&mask, SIGINT))
    unix_error("sigaddset error");
if (sigaddset(&mask, SIGTSTP))
    unix_error("sigaddset error");
if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
    unix_error("sigprocmask error");

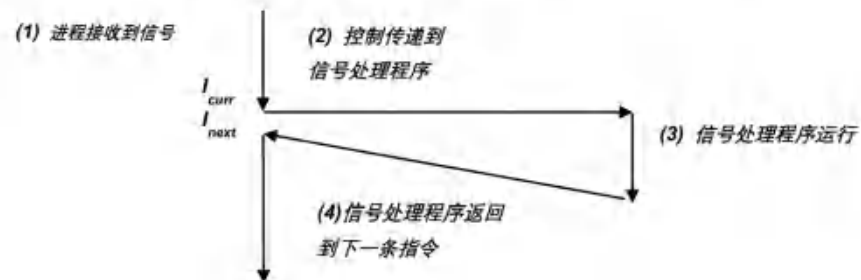
struct job_t* ptr=getjobpid(jobs,pid); /* Find the job */

/* unblock signals */
if(sigprocmask(SIG_UNBLOCK, &mask, NULL)<0)
    unix_error("sigprocmask error");
```

七、附加题（共 10 分）

51. Linux 如何处理信号？应当如何编写信号处理程序？谈谈你的理解。

（5 分）Linux 通过软中断（陷阱）方式实现信号机制，包括信号发送和信号接收两个阶段。处理过程如下



信号处理程序的编写原则：5 分

- 1) 处理程序尽可能简单
- 2) 在处理程序中只调用异步信号安全的函数
- 3) 确保其他处理程序不会覆盖当前的 `errno`
- 4) 阻塞所有信号保护对共享全局数据结构的访问
- 5) 用 `volatile` 声明全局变量
- 6) 用 `sig_atomic_t` 声明标志

29. 非本地跳转与“返回n次的函数”

28. 程序执行到 A 处继续执行后，想在程序任意位置还原到执行到 A 处的状态，通过 非本地跳转/long jmp 进行实现。

- 非本地跳转和操作系统没关系（**不是系统调用**），纯粹是用户作出的单凭C语言语法无法做到的开挂式跳转行为，是用户级的异常控制流。setjmp在第一次调用时向某个jmp_buf中保存当前状态（当前代码地址、栈指针、寄存器等），之后在某个时刻（必须是这个函数或者它嵌套调用的函数的代码，为什么？），程序通过longjmp跳到那个jmp_buf指定的代码地址并恢复状态，表现为那个地方的setjmp返回了一个longjmp指定的参数。sigsetjmp和siglongjmp还会多保存一个信号阻塞集合。

try-catch的实现原理

- 总结一下，正常的函数调用一次返回一次，系统调用execve和exit在调用成功后不返回，非系统调用函数longjmp从不返回，系统调用fork在调用成功后返回两次，非系统调用函数setjmp会返回 ≥ 1 次。

```
int main()
{
    switch(setjmp(buf)) {
        case 0:
            foo();
            break;
        case 1:
            printf("Detected an error1 condition\n");
            break;
        case 2:
            printf("Detected an error2 condition\n");
            break;
        default:
            printf("Unknown error condition\n");
    }
    exit(0);
}

/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}
```

28. 非本地跳转中的 setjmp 函数调用一次，返回 多次 次。

29. 运行一次，可返回多次 (≥ 2) 的函数是 set jmp。

一道好题

51. 一段 C 语言程序如下:

```
#include "csapp.h"
int counter = 0;
jmp_buf buf;
void handler_alrm(int sig){
}
void handler_usr1(int sig) {
    counter +=1;
    siglongjmp(buf,1);
    counter +=2;
}
int main(void)
{
    signal(SIGUSR1, handler_usr1);
    signal(SIGALRM, handler_alrm);
    if (!sigsetjmp(buf,1)) {
        sleep(10);
        printf("A");
        counter +=3;
    } else {
        printf("B");
    }
    exit(0);
}
```

在信号处理程序中用 **siglongjmp** 而不是 **longjmp**，是为了还原原来的信号阻塞状态，因为信号处理程序调用时会阻塞当前信号（尽管不是本题重点.....）

假设：上述 C 语言程序运行后，进程 ID 为 12345，且程序执行完语句 `sigsetjmp` 后收到信号，且 `printf()` 不会被信号中断。

(1) 如另一个程序给正在运行的进程 12345 发送了 1 个 **SIGALRM** 信号，屏幕输出是什么？在退出 `main` 函数之前一刻，变量 `counter` 的数值是多少？

输出：A

counter 的数值是：3

(2) 如另一个程序给正在运行的进程 12345 发送了 1 个 **SIGUSR1** 信号，可能的屏幕输出有哪些，在退出 `main` 函数之前一刻，变量 `counter` 的数值是多少，并对每种情况作出解释。

输出：B

counter 的数值是：1

输出：AB

counter 的数值是：3

答案错误吗？

输出：AB

counter 的数值是：1

输出：AB

counter 的数值是：4

30.shell原理

44. 结合 fork, execve 函数, 简述在 shell 中加载和运行 hello 程序的过程。

43. 简述 shell 的主要原理与过程。

- (其实就是LAB4干的事.....)
- shell不断地循环读取用户输入的命令行并进行解析。
- **核心:** shell运行可执行文件时, 先fork出一个子进程, 再在子进程中execve。execve传入的argv为命令行以空格分隔而成的字符串列表。
- shell本身维护一个作业列表 (与操作系统无关), 用户输入的每个命令行都对应一个新的作业, 这个命令行涉及到的所有进程都在同一个进程组中 (由操作系统维护), 终端通过对进程组中所有进程发信号来实现对这个作业的操作。
- 如果命令行以"&"结尾, 这个作业为后台作业, shell会直接继续接收用户输入而不去等它。否则就是前台作业, shell会等待它直到停止/终止为止。
- shell还应该将用户输入Ctrl+C或Ctrl+Z等产生的信号转发至前台作业的进程组。

31. 地址翻译的硬件机制

- 一般情况下，CPU通过指令中给出的地址（或者取指时的PC）直接得到的都是虚拟地址，需要通过地址翻译单元MMU将其翻译为物理地址进而访问物理主存。
- 这个过程我们要结合硬件-CPU、TLB、Cache、物理主存、磁盘(不怎么涉及)的行为以及数据传输进行分析。这些涉及地址的整数字段我们应该认识：VA(Virtual Address, 虚拟地址)、PA(Physical Address, 物理地址)、VPN(Virtual Page Number, 虚拟页号)、VPO(Virtual Page Offset, 虚拟页偏移)、PPN(物理页号)、PPO(物理页偏移)、TLBT(TLB Tag, TLB标记)、TLBI(TLB Index, TLB索引)。
- 地址翻译时必须去页表中查询这个虚拟页对应的页表条目PTE，从而得到对应的物理页。CPU一般采用多级页表（为了提高空间效率），并且使用比Cache速度更快的TLB缓存从主存中页表取出的PTE。

47. Intel I7 CPU 的虚拟地址 48 位，物理地址 52 位。

每一页面 4KB，分析如下项目：

虚拟地址中的 VPN 占 36 位；

其一级页表为 512 项。

31. 地址翻译的硬件机制

12. 某 CPU 使用 32 位虚拟地址和 4KB 大小的页时，需要 PTE 的数量是 (C)

A. 16 B. 8 C. 1M D. 512K

- 对于 Intel Core i7 (建议将教材上提到的关于它的功能单元、Cache、TLB 等具体参数打小抄)，虚拟地址 48 位，物理地址 52 位，物理页和虚拟页大小(它们始终相等)在 Linux 下被设置为 4KB (页偏移 12 位)，采取四级页表，物理地址的低 12 位为偏移 PPO，高 40 位为页号 PPN，虚拟地址的低 12 位为偏移 VPO，高 36 位为页号 VPN，VPN 会被均等分割为 4 个 9 位字段 VPN1~4，表示每一级页表里指向下一级 PTE 的索引 (所以每级页表里有 $2^9=512$ 个 PTE)。CPU 中的控制寄存器 CR3 保存页表的物理基址。

12. Intel X86-64 的现代 CPU，采用 (C) 级页表

A. 2 B. 3 C. 4 D. 由 BIOS 设置确定

47. 假设：某 CPU 的虚拟地址 14 位，物理地址 12 位；页面大小为 64B；TLB 是四路组相联，共 16 个条目；L1 数据 Cache 是物理寻址，直接映射，行大小为 4 字节，总共 16 个组。分析如下项目：

(1) 虚拟地址中的 VPN 占 8 位；物理地址的 PPN 占 6 位。

30. Intel I7 CPU 的各级页表的元素个数为 512。

46. Intel I7 CPU 的虚拟地址 48 位，虚拟内存的每一页面 4KB，物理地址 52 位，cache 块大小 64B，物理内存按字节寻址。其内部结构如下图所示，依据此结构，分析如下项目：

某指令 A 的虚拟地址为 0x804849b，则该地址对应的 VPO 为 0x 49b；

若指令 A 的物理地址为 0x86049b，则该地址对应的 PPN 为 0x 860 (40 位)；

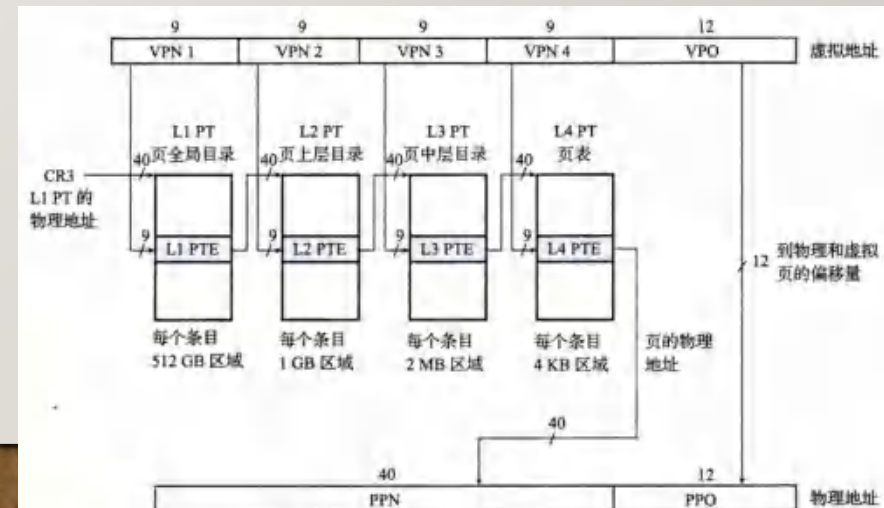


图 8-25 Core i7 页表翻译 (PT: 页表, PTE: 页表条目, VPN: 虚拟页号, VPO: 虚拟页偏移, PPN: 物理页号, PPO: 物理页偏移量。图中还给出了这四级页表的 Linux 名字)

页是地址对齐的

31. 地址翻译的硬件机制

49. 以 Intel i64 位现代处理器为例，简述加快页表 PTE 访问，大大降低页表占用空间的相关技术。
答：采用 TLB 加快页表 PTE 的访问：采用高速缓冲存储器作为页表的 Cache，TLB 中保存最近常用的虚拟页号对应的页表条目 PTE（含物理页号）。虚拟页数较少的进程页表可以完全在 TLB 中。
采用多级页表大大降低页表占用的空间：由于所有页表中大量的连续 PTE 条目都是未分配的，Intel 64 位 CPU 采用 4 级页表后，一级页表的表基条目其内容为 NULL，同样二级、三级页表也是如此。这样只有已分配页表条目采用 4、3、2、1 级页表，大大节省了页表空间。

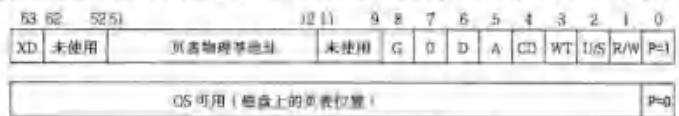
13. 下面叙述错误的是
A. 虚拟页面的起始地址%页面大小恒为 0;
B. 虚拟页面的起始地址%页面大小不一定是 0;
C. 虚拟页面大小必须和物理页面大小相同;
D. 虚拟页面和物理页面大小是可设定的系统参数;

38. (X) Intel 64 位系统页表中 PTE 的物理页号 PPN 是 64 位的，占 8 个字节。

• Core i7 的 PTE 是 8 字节 64 位 的，其中包含了下一级页表或页的物理地址，或者当页未缓存时页的磁盘位置，以及一些与内存管理密切相关的位字段。（内存管理的原理）

• i7 的一级数据 TLB 中有 16 个组，每组 4 行（路），每行一个 8 字节的 PTE（等价于块），这个 TLB 能够缓存 64 个 PTE，每个 36 位 VPN 的低 4 位为组索引 TLBI，映射到 TLB 的组，高 32 位为标记 TLBT，原理、计算、命中过程和 Cache 是一样的。

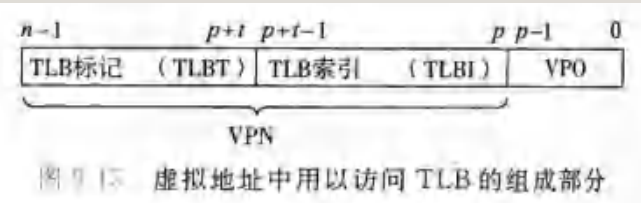
图 9-24 给出了第四级页表中条目的格式。当 P=1，地址字段包括一个 40 位 PPN，它指向物理内存中某一页的基地址。这又强调了一个要求，要求物理页 4KB 对齐。



字段	描述
P	子页表在物理内存中（1），不在（0）
R/W	对于子页，只读或者读写访问权限
U/S	对于子页，用户或超级用户（内核）模式访问权限
WT	子页的直写或写回缓存策略
CD	能/不能缓存
A	引用位（由 MMU 在读和写时设置，由软件清除）
D	修改位（由 MMU 在读和写时设置，由软件清除）
G	全局页（在任务切换时，不从 TLB 中驱逐出去）
Base addr	子页物理基地址的最高 40 位
XD	能/不能从这个子页中取指令

引发缺页故障
实现写时复制
优化页面替换
防范缓冲区溢出攻击
(限制代码执行区域)

有时关于 i7 的题目中给出的缓存容量可能稍有出入，但其它参数默认情况下需要按教材提到的为准



28. Intel I7 的 CPU 其 TLB 的每行的存储块 Block 是 8 字节。

47. 假设：某 CPU 的虚拟地址 14 位，物理地址 18 位；页面大小为 80B，TLB 是四路组相联，共 16 个条目；L1 数据 Cache 是物理寻址，直接映射，行大小为 4 字节，总共 16 个组。分析如下项目：

(2) TLB 的组索引位数 TLBI 为 2 位。

15. CPU 一次访存时, 访问了 L1、L2、L3 Cache 所用地址 A1、A2、A3 的关系 (B)

- A. $A1 > A2 > A3$ B. $A1 = A2 = A3$ C. $A1 < A2 < A3$ D. $A1 = A2 < A3$

11. CPU 访问 TLB、Cache 时使用的地址分别是 (B)

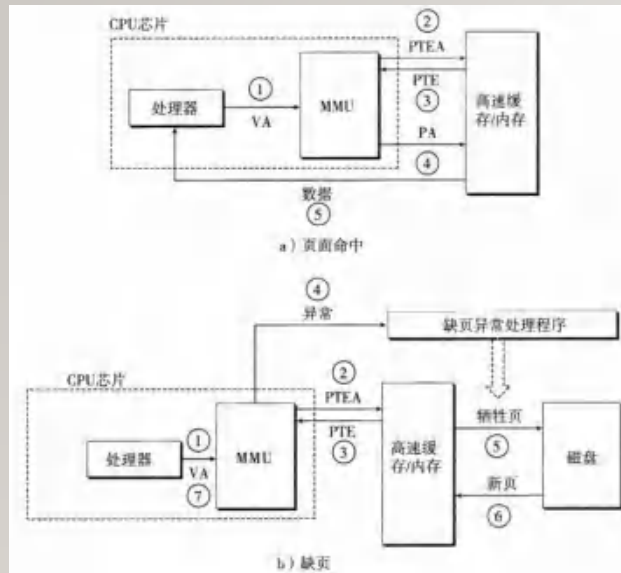
- A. 虚拟地址、虚拟地址 B. 虚拟地址、物理地址
C. 物理地址、虚拟地址 D. 物理地址、物理地址

31. 地址翻译的硬件机制-过程分析 (重点!)

这几个书上的图务必要理解，
考试时会直接拿它们考！

←注意Cache是物理地址访问的，
无论L1L2L3

(这两个图里直接把TLB跟Cache混一块了，大家明白意思就好.....)



在任何既使用虚拟内存又使用 SRAM 高速缓存的系统中，都应该使用虚拟地址还是使用物理地址来访问 SRAM 高速缓存的问题。尽管关于这个折中的详细讨论已经超出了我们的讨论范围，但是大多数系统是选择物理寻址的。使用物理寻址，多个进程同时存在高速缓存中有存储块和共享来自相同虚拟页面的块成为很简单的事情。而且，高速缓存无需处理保护问题，因为访问权限的检查是地址翻译过程的一部分。

图 9-14 展示了一个物理寻址的高速缓存如何和虚拟内存结合起来。主要的思路是地址翻译发生在高速缓存查找之前。注意，页表条目可以缓存，就像其他的数据字一样。

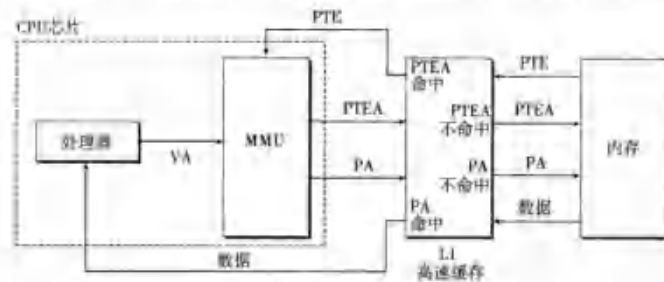


图 9-14 将 VM 与物理寻址的高速缓存结合起来 (VA: 虚拟地址, PTEA: 页表条目地址; PTE: 页表条目; PA: 物理地址)

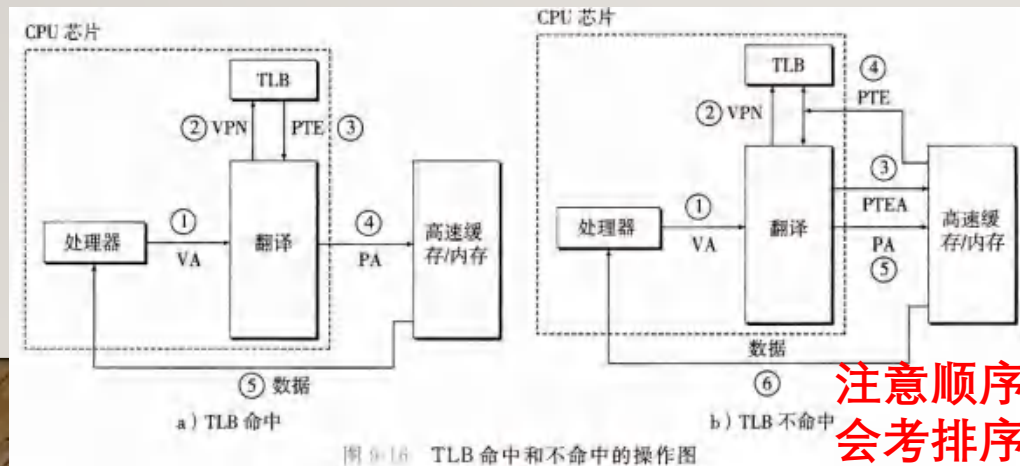


图 9-16 TLB 命中和不命中中的操作图

注意顺序，
会考排序

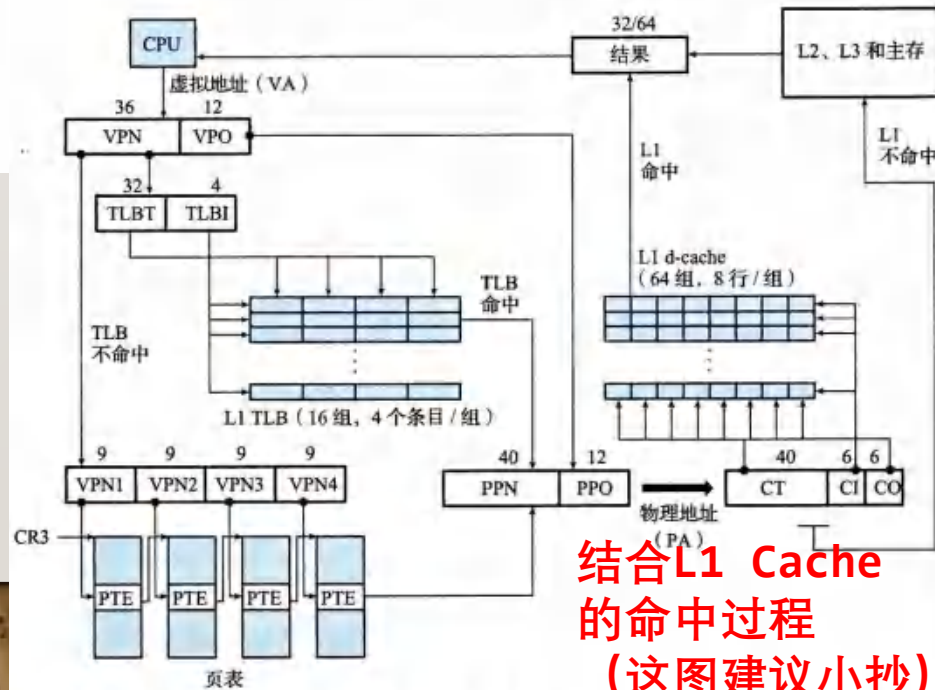


图 9-22 Core i7 地址翻译的概况。为了简化，没有显示 i-cache、i-TLB 和 L2 统一 TLB

结合L1 Cache
的命中过程
(这图建议小抄)

往往会跟Cache结合起来

The diagram illustrates the memory access process in a system with a TLB, Translation, and L1 Cache. The components and their interactions are as follows:

- CPU**: The central processing unit, shown in a red box on the left.
- TLB**: Translation Lookaside Buffer, shown in a white box at the top.
- 翻译 (Translation)**: The translation process, shown in a green box in the center.
- L1 高速缓存 (L1 Cache)**: The first-level cache, shown in a grey box on the right.
- 内存 (Memory)**: The main memory, shown in a grey box on the far right.

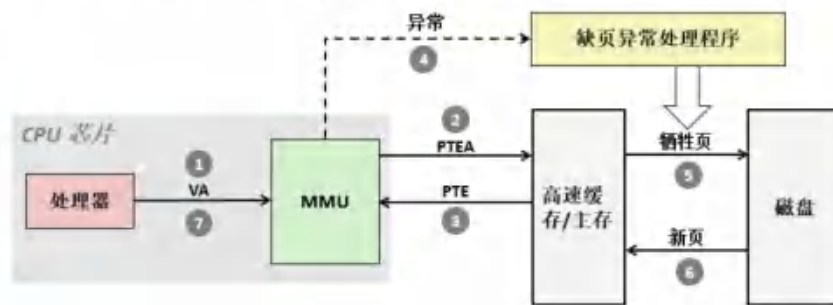
The process flow is indicated by arrows and labels:

- The **CPU** sends a request labeled **(A) VA** to the **翻译** block.
- The **翻译** block sends a request labeled **(B) VPN** to the **TLB**.
- The **TLB** sends a response labeled **(C) PTE** back to the **翻译** block.
- The **翻译** block sends a request labeled **(D) PTE** to the **L1 高速缓存**.
- The **L1 高速缓存** has three states:
 - PTE 命中 (PTE Hit)**: The cache contains the PTE, and it sends a response labeled **(G) PTE** back to the **翻译** block.
 - PTE 不命中 (PTE Miss)**: The cache does not contain the PTE, and it sends a response labeled **(H) PTEA** back to the **翻译** block.
 - PA 不命中 (PA Miss)**: The cache does not contain the physical address (PA), and it sends a response labeled **(I) PA** back to the **翻译** block.
- The **翻译** block sends a request labeled **(F) PA** to the **L1 高速缓存**.
- The **L1 高速缓存** sends a response labeled **(J) 数据 (Data)** back to the **翻译** block.
- The **翻译** block sends a request labeled **(K) 数据** back to the **CPU**.

The **L1 高速缓存** also interacts with the **内存** (Memory) for data retrieval, as indicated by the arrow labeled **(J) 数据** from the cache to the memory.

- (1) $A \Rightarrow B \Rightarrow C \Rightarrow F \Rightarrow K$
 (2) $A \Rightarrow B \Rightarrow E \Rightarrow D \Rightarrow F \Rightarrow K$
 (3) $A \Rightarrow B \Rightarrow E \Rightarrow H \Rightarrow G \Rightarrow (D) \Rightarrow F \Rightarrow I \Rightarrow J \Rightarrow (K)$; D, K 缺少也正确

42. 结合下图，简述虚拟内存地址翻译的过程。



19. 执行一条指令最不幸时需要访问 (D) 次各类存储器
A. 2 B. 4 C. ≤ 8 D. > 8

极端情况 addq %rax, (%rbx) 啥都不命中

EOF

