

第一章、 计算机系统漫游

一、 真题考点

◆ 考点 1：程序的执行过程

编译系统：预处理器、编译器、汇编器和链接器

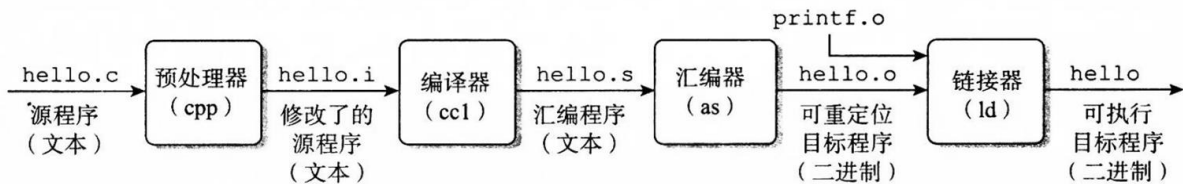


图 1-3 编译系统

◆ 考点二：计算机系统的抽象概念

1. 文件是对 I/O 设备的抽象表示
2. 虚拟内存是对主存和磁盘 I/O 设备的抽象表示
3. 进程则是对处理器、主存和 I/O 设备的抽象表示。
4. 在处理器里，指令集架构提供了对实际处理器硬件的抽象。[2020 考点]
5. 虚拟机，它提供对整个计算机的抽象，包括操作系统、处理器和程序。

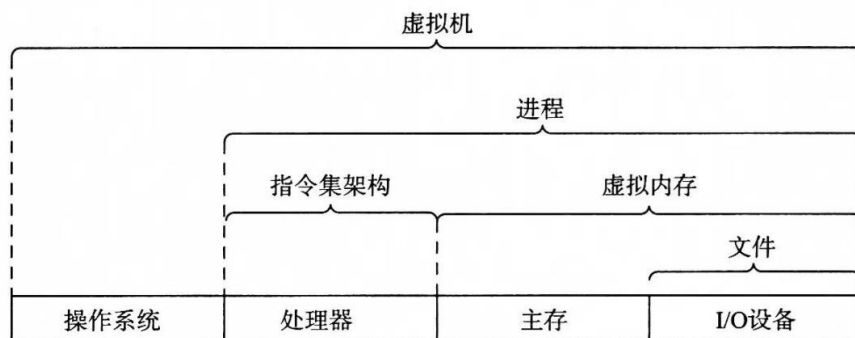


图 1-18 计算机系统提供的一些抽象。计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性

◆ 考点三：虚拟内存概述

1. 堆：代码和数据区后紧随着的是运行时堆。当调用像 `malloc` 和 `free` 这样的 C 标准库函数时，堆可以在运行时动态地扩展和收缩。
2. 栈：用户栈在程序执行期间可以动态地扩展和收缩。特别地，每次我们调用一个函数时，栈就会增长；从一个函数返回时，栈就会收缩。

◆ 考点四：Amdahl 定律

若系统执行某应用程序需要时间为 T_{old} 。假设系统某部分所需执行时间与该时间的比例为 α ，而该部分性能提升比例为 k 。即该部分初始所需时间为 αT_{old} ，现在所需时间为 $(\alpha T_{old}) / k$ 因此，总的执行时间应为

$$T_{new} = (1 - \alpha) T_{old} + (\alpha T_{old}) / k = T_{old} [1 - \alpha + \alpha / k]$$

由此，可以计算加速比 $S = T_{old} / T_{new}$ 为

$$S = 1 / (1 - \alpha + \alpha / k)$$

二、 真题演练

(一) 选择题

1. `hello.c` 文件在 () 生成 `hello.o` 文件
 A. 预处理阶段 B. 编译阶段 C. 汇编阶段 D. 链接阶段
2. 计算机操作系统抽象表示时 () 是对处理器、主存和 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
3. 操作系统通过提供不同层次的抽象表示来隐藏系统实现的复杂性, 其中 () 是对实际处理器硬件的抽象。
 A. 进程 B. 虚拟存储器 C. 文件 D. 指令集架构 (ISA)
4. 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. `ccl` C. `as` D. `ld`
5. 当函数调用时, () 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. 内核虚拟存储器
6. 当调用 `malloc` 这样的 C 标准库函数时, () 可以在运行时动态的扩展和收缩。
 A. 堆 B. 栈 C. 共享库 D. 内核虚拟存储器
7. 虚拟内存是对主存和磁盘 I/O 设备的抽象表示, 即是对 () 的抽象。
 A. 进程 B. 程序存储器 C. 文件 D. 指令集架构 (ISA)

(二) 填空题

1. 翻译过程由_____组成。
2. 文件是对_____的抽象表示, 虚拟内存是对_____的抽象表示, 进程则是对_____的抽象表示。

(三) 简答题

1. 公司的市场部向你的客户承诺, 下一个版本的软件性能将改进 2 倍, 但只有 80% 的系统能够被改进, 那么这部分需要被改进多少才能达到整体性能目标?

第二章、 信息的表示和处理

一、 真题考点

◆ 考点一：大端方式和小端方式

最低有效字节在最前面的方式，称为小端法(little endian)，使用小端法的机器：Linux、Window 等，大多数 Intel 兼容机都只用小端模式。

最高有效字节在最前面的方式，称为大端法(big endian)，使用大端法的机器：Sun

假设变量 x 的类型为 int，位于地址 0x100 处，它的十六进制值为 0x01234567。地址范围 0x100~0x103 的字节顺序依赖于机器的类型：

| | | | | | |
|-----|-------|-------|-------|-------|-----|
| 大端法 | | | | | |
| | 0x100 | 0x101 | 0x102 | 0x103 | |
| ... | 01 | 23 | 45 | 67 | ... |
| 小端法 | | | | | |
| | 0x100 | 0x101 | 0x102 | 0x103 | |
| ... | 67 | 45 | 23 | 01 | ... |

注意，在字 0x01234567 中，高位字节的十六进制值为 0x01，而低位字节值为 0x67。

◆ 考点二：有符号数和无符号数的转换及 C 语言标准

1. 有符号数和无符号数的转换

对于大多数 C 语言的实现，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

2. C 语言中的有符号数和无符号数

当执行一个运算时，如果它的一个运算数是有符号的而另一个是无符号的，那么 C 语言会隐式地将有符号参数强制类型转换为无符号数，并假设这两个数都是非负的，来执行这个运算。

| 表 达 式 | 类 型 | 求 值 |
|--------------------------------|-----|-----|
| 0 == 0U | 无符号 | 1 |
| -1 < 0 | 有符号 | 1 |
| -1 < 0U | 无符号 | 0* |
| 2147483647 > -2147483647-1 | 有符号 | 1 |
| 2147483647U > -2147483647-1 | 无符号 | 0* |
| 2147483647 > (int) 2147483648U | 有符号 | 1* |
| -1 > -2 | 有符号 | 1 |
| (unsigned) -1 > -2 | 无符号 | 1 |

◆ 考点三：无符号加法和有符号加法的溢出条件

1. 无符号加法溢出条件

原理：检测无符号数加法中的溢出

对在范围 $0 \leq x, y \leq UMax_w$ 中的 x 和 y ，令 $s \doteq x +_w y$ 。则对计算 s ，当且仅当 $s < x$ (或者等价地 $s < y$) 时，发生了溢出。

2. 有符号加法的溢出条件

原理：检测补码加法中的溢出

对满足 $TMin_w \leq x, y \leq TMax_w$ 的 x 和 y ，令 $s = x +_w y$ 。当且仅当 $x > 0, y > 0$ ，但 $s \leq 0$ 时，计算 s 发生了正溢出。当且仅当 $x < 0, y < 0$ ，但 $s \geq 0$ 时，计算 s 发生了负溢出。

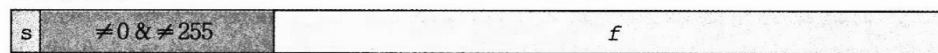
◆ 考点四：IEEE 浮点标准

IEEE 浮点标准用 $(-1)^s \times M \times 2^E$ 的形式来表示一个数：在单精度浮点格式(C 语言中的 float)中， s 、 exp 和 $frac$ 字段分别为 1 位、 $k=8$ 位和 $n=23$ 位，得到一个 32 位的表示。在双精度浮点格式(C 语言中的 double)中， s 、 exp 和 $frac$ 字段分别为 1 位、 $k=11$ 位和 $n=52$ 位，得到一个 64 位的表示。

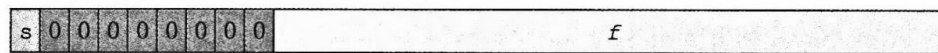
(**float:1+8+23, double:1+11+52**)

根据 exp 的值，被编码的值可以分成三种不同的情况（以单精度格式为例）。

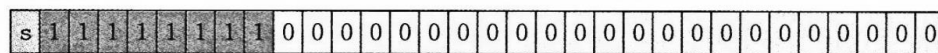
1. 规格化的



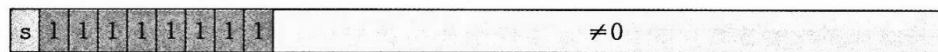
2. 非规格化的



3a. 无穷大



3b. NaN



1. 规格化的值：当 exp 的位模式既不全为 0，也不全为 1

阶码字段 exp 被解释为以偏置(biased)形式表示的有符号整数，即阶码的值是 $E = e - Bias$ ，其中 e 是无符号数，而 $Bias = 2^{k-1} - 1$ 。

小数字段 $frac$ 被解释为描述小数 $M = 1.xxxx = 1 + f$ ，其中 1 不显式表示

2. 非规格化的值：当阶码域为全 0

在这种情况下，阶码值是 $E = 1 - Bias$ ，而尾数的值是 $M = 0.xxxx = f$ 也就是小数字段的值，不包含隐含的开头的 1。

3. 特殊值：当阶码域为全 1

当小数域全为 0 时，得到的值表示无穷；当小数域为非零时，结果值被称为 NaN。

◆ 考点五：整数除法舍入和浮点数的舍入

整数除法舍入：向零舍入（舍入到零）

浮点数的舍入：向偶数舍入，即当数字是中间值时，将数字向上或者向下舍入，使得结果的最低有效数字是偶数，否则直接是向上或者向下舍入。

◆ 考点六：C 语言中的浮点数

1. 浮点数运算的性质（不考虑正负无穷和 NaN）

| | 交换律 | 结合律 | 单调性 |
|----------|-----|-----|--------|
| 无符号/补码加法 | 有 | 有 | 无（有溢出） |

| | | | |
|----------|---|----------|--------|
| 无符号/补码乘法 | 有 | 有 | 无（有溢出） |
| 浮点加法 | 有 | 无（大数吃小数） | 有 |
| 浮点乘法 | 有 | 无（大数吃小数） | 有 |

2.double、float 和 int 的转换

从 int 转换成 float，数字不会溢出，但是可能被舍入。

从 int 或 float 转换成 double，不会溢出，也不会舍入

从 double 转换成 float，可能溢出或舍入

从 float 或者 double 转换成 int，值将会向零舍入，也可能溢出

◆ 考点七：期末真题出现过的重要简答题：

1. 请在数轴上画出非负 float 数的各区间的密度分布，并标示各区间是规格化还是非规格化数、浮点数密度、最小值、最大值。

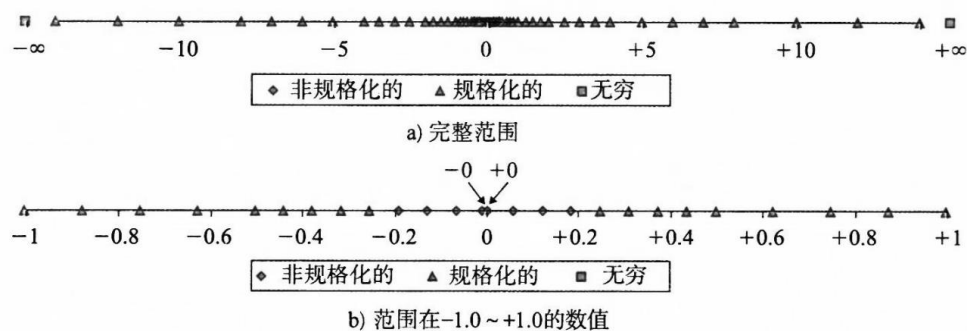


图 2-34 6 位浮点格式可表示的值($k=3$ 的阶码位和 $n=2$ 的尾数位。偏置量是 3)

2. 请结合 ieee754 编码，说明怎样判断两个浮点数是否相等？

(1) 说明浮点数表示原理：以 float 为例，1 符号、8 位的阶码、23 位的尾数三部分，可以表示浮点规格化数、非规格化数、无穷大、NaN 等浮点数据。

(2) 相等的判别描述：由于浮点数的 ieee754 编码表示存在着精度、舍入、溢出、类型不匹配等问题，两个浮点数不能够直接比较大小，应计算两个浮点数的差的绝对值，当绝对值小于某个可以接受的数值（精度）时认为相等。

◆ 考点八：期末真题经常考查的程序转化问题

这类题目选择编译器、编译阶段、汇编阶段!!!

程序中的 2 进制、10 进制、16 进制数，在 **汇编阶段** 变成 2 进制。

c 语言程序中的整数常量、整数常量表达式是在 **编译阶段** 变成 2 进制补码的

c 语句中的有符号常数，在 **编译阶段** 转换成了补码

c 语言程序中的常量表达式的计算是由 **编译器** 完成的

十进制的有符号常数由 **编译器** 转换为二进制补码[2020 考点]

c 语言的常量表达式的计算是由 **编译器** 完成的

二、 真题演练

(一) 选择题

1. 程序中的 2 进制、10 进制、16 进制数, 在 () 时变成 2 进制。

A. 汇编时 B. 连接时 C. 执行时 D. 调试时

2. C 语言程序中的整数常量、整数常量表达式是在 () 阶段变成 2 进制补码的。

A. 预处理 B. 编译 C. 链接 D. 执行

3. 变量 x 的值为 0x01234567, 地址 &x 为 0x100; 则该变量的值在 x86 和 Sun 机器内存中的存储排列顺序正确的是 ()

| 选项 | 机器类型 | 地址 | | | |
|----|------|-------|-------|-------|-------|
| | | 0x100 | 0x101 | 0x102 | 0x103 |
| A | x86 | 67 | 45 | 23 | 01 |
| | Sun | 01 | 23 | 45 | 67 |
| B | x86 | 76 | 54 | 32 | 10 |
| | Sun | 01 | 23 | 45 | 67 |
| C | x86 | 01 | 23 | 45 | 67 |
| | Sun | 67 | 45 | 23 | 01 |
| D | x86 | 01 | 23 | 45 | 67 |
| | Sun | 01 | 23 | 45 | 67 |

4. C 语句中的有符号常数, 在 () 阶段转换成了补码

A. 编译 B. 链接 C. 执行 D. 调试

5. C 语言中的 int 和 unsigned 类型的常数进行比较时, 下列表达式及描述正确的是: ()

(注: 位宽为 32 位, TMIN=-2147483648, TMAX=2147483647)

- A. $0 == 0U$, 按有符号数进行比较
 B. $2147483647U > -2147483647-1$, 按无符号数进行比较
 C. $(unsigned)-1 < -2$, 按无符号数进行比较
 D. $2147483647 > (int)2147483648U$, 按有符号数进行比较

6. 计算机信息常用编码中, 字符 0 的编码不可能是 16 进制数 ()

A. 30 B. 30 00 C. 00 D. 00 30

7. 在采用补码运算的 32 位机器上, 下列 C 表达式中正确的是 ()

- A. $-1 < 0U$ B. $2147483647U > -2147483647-1$
 C. $(unsigned)-1 > -2$ D. $-2147483647-1U < 2147483647$

注: $2147483648 = 2^{31}$

8. 下面关于 IEEE 浮点数标准说法正确的是 ()

- A. 在位数一定的情况下, 不论怎么分配 exponent bits 和 fraction bits, 所能表示的数的个数是不变的;
 B. 如果甲类浮点数有 10 位, 乙类浮点数有 11 位, 那么甲所能表示的最大数一定比乙小;
 C. 如果甲类浮点数有 10 位, 乙类浮点数有 11 位, 那么甲所能表示的最小正数一定比乙小;

D. "0111000"可能是 7 位浮点数的 NAN 表示;

9. 在整数除法运算中, 默认的舍入方式为 (), 而在浮点数运算中, 默认的舍入方式为 ()。

A. 向偶数舍入 B. 向零舍入 C. 向下舍入 D. 向上舍入

10. 在对有符号值使用补码运算的 32 位机器上运行代码, 对于有符号值使用的是算术右移, 而对于无符号值使用的是逻辑右移, 下列 C 表达式一定正确的是 ()

```
int x = random();      int y = random();
```

```
unsigned ux = x;      unsigned uy = y;
```

I. $(x * x) \geq 0$ II. $x+y == uy+ux$ III. $(x > 0) || (x-1 < 0)$

IV. $(x < y) == (-x > -y)$ V. $\sim x + \sim y + 1 == \sim (x+y)$ VI. $((x >> 2) << 2) \leq x$

A. I、II、V B. II、V、VI C. III、IV、VI D. IV、V、VI

11. 对 $x = 1/8$ 和 $y = 3/8$ 进行小数点后两位取整 (向偶数舍入), 结果正确的是 ()

A. 1/4, 1/4 B. 0, 1/4 C. 1/4, 1/2 D. 0, 1/2

12. 假设有下面 x 和 y 的程序定义

```
int x = a >> 2;
```

```
int y = (x + a) / 4;
```

那么有多少个位于闭区间 $[-8, 8]$ 的整数 a 能使得 x 和 y 相等? ()

A. 12 B. 13 C. 14 D. 15

13. 单精度浮点数 27.5 实际存储在内存中的十六进制数值为: ()

A. 0x41ee0000 B. 0x425c0000 C. 0x41dc0000 D. 0x025c0000

14. C 语言中 float 类型的数据 0.1 的机器数表示, 错误的是 ()

A. 规格化数 B. 不能精确表示 C. 与 0.2 有 1 个二进制位不同 D. 唯一的

15. 对于 IEEE 浮点数, 如果减少 1 位指数位, 将其用于小数部分, 将会有怎样的效果? ()

A. 能表示更多数量的实数值, 但实数值取值范围比原来小了。

B. 能表示的实数数量没有变化, 但数值的精度更高了。

C. 能表示的最大实数变小, 最小的实数变大, 但数值的精度更高。

D. 以上说法都不正确。

16. 关于 IEEE float 类型的数据 +0.0 的机器数表示, 说法错误的是 ()

A. 是非规格化数 B. 不能精确表示 C. +0.0 与 -0.0 不同 D. 唯一的

17. 下列说法错误的是 ()

A. 无符号加法和补码加法具有结合律, 而浮点加法不具有结合律。

B. 无符号乘法和补码乘法具有结合律, 而浮点乘法不具有结合律。

C. 无符号加法和补码加法具有单调性, 而浮点加法不具有单调性。

D. 浮点乘法在加法上不具有分配律 $[a * (b + c) = a * b + a * c.]$

18. 变量 x、f、和 d 的类型分别是 int、float 和 double, 其值是任意的, 不过 f 和 d 不能是 $+\infty$ 、 $-\infty$ 或者 NaN, 对于下面的 C 表达式, 一定正确的是 ()

I. $x == (int)(double) x$

II. $d == (double)(float) d$

III. $f == (\text{float})(\text{double}) f$

 IV. $(f+d)-f == d$

 V. $d*d \geq 0.0$

 VI. $1.0/2 == 1/2.0$

A. I、II、V、VI

B. I、II、V、VI

C. I、III、IV、VI

D. I、III、V、VI

19. C 语言中不同类型的数值进行强制类型转换时, 下列说法错误的是 ()

 A. 从 `int` 转换成 `float` 时, 数值可能会溢出

 B. 从 `int` 转换成 `double` 后, 数值不会溢出

 C. 从 `double` 转换成 `float` 时, 数值可能会溢出, 也可能舍入

 D. 从 `double` 转换成 `int` 时, 数值可能溢出, 可能舍入

20. 下列说法错误的是 ()

A. CPU 无法判断参与加法运算的数据是有符号或无符号数。

 B. 对 `unsigned int x`, $(x*x) \geq 0$ 总成立。

C. CPU 无法判断加法运算的和是否溢出。

 D. C 语言中 `int` 的个数比 `float` 个数少。

 21. 下列 16 进制数值中, 可能是 Linux64 系统中 `char*` 类型的指针值是 ()

 A. `e4f9`

 B. `b4cc2200`

 C. `b811e5ffff7f0000`

 D. `30`

22. C 语言中整数 -1 与无符号 0 比较, 其结果是 ()

A. 大于

B. 小于

C. 可能大于可能小于

D. 无法比较

23. 在 IEEE 浮点数标准中, 单精度浮点数采用 () 位的小数字段对尾数进行编码。

A. 23

B. 24

C. 52

D. 63

 24. 给定字长的整数 `x` 和 `y` 按补码相加, 和为 `s`, 则发生正溢出的情况是 ()

 A. $x > 0, y > 0, s \leq 0$

 B. $x > 0, y < 0, s \leq 0$

 C. $x > 0, y < 0, s \geq 0$

 D. $x < 0, y < 0, s \geq 0$

25. C 语言程序中的常量表达式的计算是由 () 完成的

A. 编辑器

B. 编译器

C. 链接器

D. 加载器

(二) 填空题

1. 在机器上的字节存储中, 最低/最高有效字节在最前面的方式, 分别称为小端法/大端法, 大多数 Intel 兼容机使用_____模式。

 2. 64 位系统中 `int -2` 的机器数二进制表示_____。

3. C 语言的常量表达式的计算是由_____完成的

 4. 判断整型变量 `x` 的位 7 为 1 的 C 语言表达式是_____。

 5. 在 C 语言中进行右移运算时, 大多数情况下都对有符号数使用_____, 因此对参数 `x=[10010101]` 而言, 对其进行该移位时 `x>>4` 的值为_____。

6. C 语言程序中, 有符号数强制转换成无符号数时, 二进制表示_____ (会/不会) 做相应调整。

 7. C 程序中定义 `int x=-3`, 则 `&x` 处依次存放 (小端模式) 的十六进制数据为_____。

 8. 编译器常常会优化常数因子乘法, 针对表达式 `x * 55`, 编译器可以采用 2 个移位、2 个加法/减法产生表达式_____。

9. 在整数除法中,其默认舍入的方式是_____,因此在使用算术右移的补码机器中用一条 C 表达式_____即可计算 $x / 2^k$ 。

10. 若字节变量 x 和 y 分别为 $0x10$ 和 $0x01$, 则 C 表达式 $x \& \sim y$ 的字节值是_____。

11. 在数学上, $1/3$ 是一个无限循环小数, 其二进制表达式是_____。

12. 根据 IEEE 浮点表示标准, _____的数称为规格化数, _____的数被称为非规格化数。

13. 按照“向偶数舍入”的规则, 二进制小数 101.110_2 舍入到最接近的 $1/2$ (小数点右边 1 位) 后的二进制为_____。

14. 若按 IEEE 浮点标准的单精度浮点数 (符号位 1 位, 阶码字段 exp 占据 8 位, 小数字段 frac 占据 23 位) 表示 -8.25 , 结果是_____ (用 16 进制)。

15. 若我们采用基于 IEEE 浮点格式的浮点数表示方法, 阶码字段 exp 占据 k 位, 小数字段 frac 占据 n 位, 则最大的非规格的正数是_____ (结果用含有 n, k 的表达式表示)

(三) 简答题

1. 假设 $a=0x66$ 、 $b=0x39$, 填写下列 C 表达式的字节值 (用十六进制表示)

| C 表达式 | 表达式值 | C 表达式 | 表达式值 |
|--------------|------|----------------------|------|
| $a \& b$ | | $\sim a \mid \sim b$ | |
| $a \mid b$ | | $a \& \sim b$ | |
| $!a \mid !b$ | | $a \& b$ | |
| $a \& !b$ | | $a \mid b$ | |

2. 根据舍入到偶数规则, 说明如何将下列二进制小数值舍入到最接近的二分之一 (二进制小数点右边 1 位)。对每种情况, 给出舍入前后的数字值。

A. 10.010_2

B. 10.011_2

C. 10.110_2

D. 11.001_2

3. 请按 IEEE 浮点标准的单精度浮点数表示下表中的数值, 首先写出形如 $(-1)^s \times M \times 2^E$ 的表达式, 然后给出十六进制的表示。

注: 单精度浮点数的字段划分如下: 符号位 (s): 1bit; 阶码字段 (exp): 8bit; 小数字段 (frac): 23bit; 偏置值 (bias): 127。

| 值 | $(-1)^s \times M \times 2^E, 1 \leq M < 2$ | 十六进制 |
|------------|--|------|
| -1.5 | | |
| 0.375 | | |
| -12.5 | | |
| 2^{-149} | | |

4. 考虑一种新的遵从 IEEE 规范的浮点的格式, 包含 3 位指数位, 4 位小数位和 1 位符号位。请填写下面的表格:

| 描述 | 十进制数 (或分数) | 二进制表示 |
|--------|------------|-------|
| Bias | | ----- |
| 最小的正数 | | |
| 最小的有限数 | | |
| 最小规格正数 | 1/4 | |
| --- | -11/32 | |
| --- | 7/2 | |
| --- | 63/128 | |
| --- | 13.25 | |

5. 写出 `float f = -1` 的 IEEE754 编码。(请按步骤写出转换过程)

6. 考虑一种 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数的格式原理，其字段划分如下：符号位 (s): 1bit; 阶码字段 (exp): 4bit; 小数字段 (frac): 7bit。

1) 请写出在下列区间中包含多少个用上面规则精确表示的浮点数

A: $[1, 2)$ _____

B: $[2, 3)$ _____

2) 请写出下面浮点数的二进制表示

| 数值 | 二进制表示 |
|-------------------|-------|
| 最小的正规格化数 | |
| 最大的非规格化数 | |
| $17\frac{1}{16}$ | |
| $-\frac{1}{8192}$ | |
| $20\frac{3}{8}$ | |
| $-\infty$ | |

第三章、程序的机器级表示

一、真题考点

◆ 考点一：常用的整数寄存器

| 63 | 31 | 15 | 7 | 0 |
|------|------|------|------|---------|
| %rax | %eax | %ax | %al | 返回值 |
| %rbx | %ebx | %bx | %bl | 基址寄存器 |
| %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 | 栈帧 |
| %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 |

当这些指令以寄存器作为目标时，对于生成小于 8 字节结果的指令，寄存器中剩下的字节会怎么样，对此有两条规则：

1. 生成 1 字节和 2 字节数字的指令会保持剩下的字节不变；
2. 生成 4 字节数字的指令会把高位 4 个字节置为 0。

◆ 考点二：寻址模式以及数据传送指令

1. 立即数寻址、寄存器寻址、内存寻址

立即数：用来表示常数值，立即数的书写方式是 '\$' 后面跟一个标准 C 表示法表示的整数

寄存器：表示某个寄存器的内容。用符号 r_a 来表示任意寄存器 a ，用引用 $R[r_a]$ 来表示它的值，这是寄存器集合看成一个数组 R ，用寄存器标识符作为索引。

存储器（内存引用）：根据计算出来的地址访问某个内存位置。用符号 $M_b[Addr]$ 表示对存储在内存中从地址 $Addr$ 开始的 b 个字节值的引用，为了简便，通常省去下标 b 。

| 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 |

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

2. movb、movw、movl、movq

| 源 | 目的 | 源操作数, 目的操作数 | C 语言模拟 |
|------|-----|-------------------------|----------------|
| movq | Imm | Reg movq \$0x4, %rax | temp = 0x4; |
| | | Mem movq \$-147, (%rax) | *p = -147; |
| | Reg | Reg movq %rax, %rdx | temp2 = temp1; |
| | | Mem movq %rax, (%rdx) | *p = temp; |
| | Mem | Reg movq (%rax), %rdx | temp = *p; |

单条指令不能进行从内存到内存的数据传送

◆ 考点三：算术逻辑运算

这些指令需要认识记忆，在代码填空中应该会写：

| 指令 | 效果 | 描述 |
|-----------|---------------------------|------------|
| leaq S, D | $D \leftarrow \&S$ | 加载有效地址 |
| INC D | $D \leftarrow D + 1$ | 加1 |
| DEC D | $D \leftarrow D - 1$ | 减1 |
| NEG D | $D \leftarrow -D$ | 取负 |
| NOT D | $D \leftarrow \sim D$ | 取补 |
| ADD S, D | $D \leftarrow D + S$ | 加 |
| SUB S, D | $D \leftarrow D - S$ | 减 |
| IMUL S, D | $D \leftarrow D * S$ | 乘 |
| XOR S, D | $D \leftarrow D \wedge S$ | 异或 |
| OR S, D | $D \leftarrow D \vee S$ | 或 |
| AND S, D | $D \leftarrow D \& S$ | 与 |
| SAL k, D | $D \leftarrow D \ll k$ | 左移 |
| SHL k, D | $D \leftarrow D \ll k$ | 左移（等同于SAL） |
| SAR k, D | $D \leftarrow D \gg_A k$ | 算术右移 |
| SHR k, D | $D \leftarrow D \gg_L k$ | 逻辑右移 |

注意：移位量可以是一个立即数，或者放在单字节寄存器%cl 中，而且移位有逻辑和算术移位

◆ 考点四：控制

包括条件语句 if、循环语句 for、分支语句 switch

1. 条件码及相关指令

| | |
|----------------------------|---|
| CF: 进位标志——检查无符号操作的溢出 | (1) leaq 操作不会改变条件码，因为 leaq 是操作地址 (2) 只设置条件码而不改变任何其他寄存器： CMP (cmpb、w、l、q) : 类似 SUB 指令 TEST (testb、w、l、q) : 类似于 AND 指令 |
| ZF: 零标志——最近操作结果为 0 | |
| SF: 符号标志——最近操作结果为负数 | |
| OF: 溢出标志——导致一个补码溢出，正溢出或负溢出 | |

2. 条件分支

用条件控制实现：if-else，用条件传送实现：cmovxx

3. 循环

- (1) do-while 循环
- (2) while 循环

jump to middle (跳转到中间): 它执行一个无条件跳转跳到循环结尾处的测试, 以此来执行初始的测试。

guarded-do: 首先用条件分支, 如果初始条件不成立就跳过循环, 把代码变换为 do-while 循环。

(3) for 循环

4.switch 语句

掌握跳转表

◆ 考点五: 过程

1. 运行时栈

需要栈存储信息的情况:

(1) 寄存器不足够存放所有的本地数据: x86-64 中, 可以通过寄存器最多传递 6 个整型 (例如整数和指针) 参数。如果一个函数有大于 6 个整型参数, 超出 6 个的部分就要通过栈来传递。

(2) 对一个局部变量使用地址运算符 “&” 因此必须能够为它产生一个地址。

(3) 某些局部变量是数组或结构, 因此必须能够通过数组或结构引用被访问到。

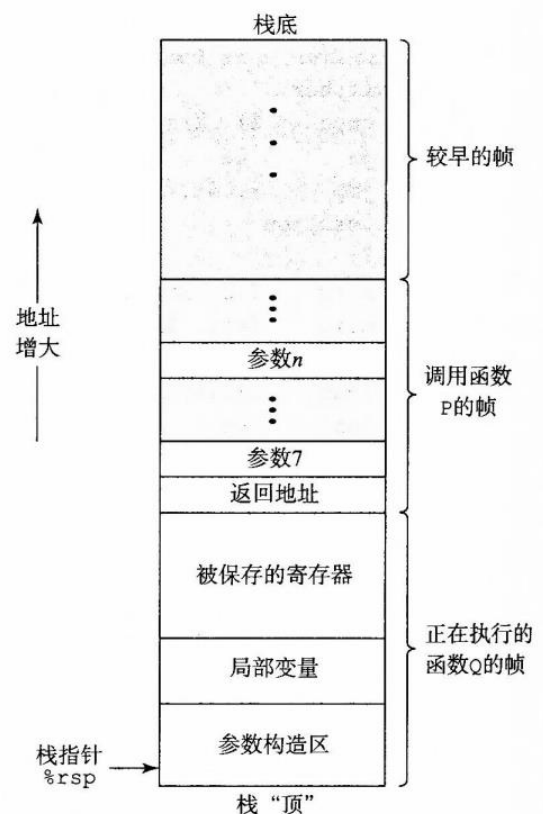


图 3-25 通用的栈帧结构 (栈用来传递参数、存储返回信息、保存寄存器, 以及局部存储。省略了不必要的部分)

◆ 考点六: 数组、结构体、联合

结构体: 结构的所有组成部分都存放在内存中一段连续的区域内, 而指向结构的指针就是结构第一个字节的地址。编译器维护关于每个结构类型的信息, 指示每个字段 (field) 的字节偏移, 它以这些偏移作为内存引用指令中的位移, 从而产生对结构元素的引用。

联合: 一个联合的总的大小等于它最大字段的大小。

在这里主要考查数据的对齐原则: 在 Intel 中建议数据结构遵循数据对齐, 其对齐原则为: 任何 κ 字节的基本对象的地址必须是 κ 的倍数, 具体说来有以下几点:

1. 每种数据类型有各自的对齐要求

每种类型的对象都满足它的对齐限制, 就可保证实施对齐, 比如 float 要求四字节对齐, 其起始地址必须为 4 的整数倍。

2. 结构体的对齐

首先满足各数据的对齐要求(体现在偏移量上),因此编译器可能需要在字段的分配中插入间隙。

| | | | | | | | | |
|----|---|---|---|---|--|---|---|----|
| 偏移 | 0 | | 4 | 5 | | 8 | | 12 |
| 内容 | | i | | c | | | j | |

在此基础上编译器结构的末尾可能需要一些填充,这样结构数组中的每个元素都会满足它的对齐要求,如结构中含有 char、int、double、char*,则必须满足结构体长度为 8 的整数倍。

| | | | | | | | | |
|----|---|---|---|---|---|---|--|----|
| 偏移 | 0 | | 4 | | 8 | 9 | | 12 |
| 内容 | | i | | j | | c | | |

3. 结构体数组

将结构体当成整体看待,以满足整体的对齐要求(即满足结构体单元中最大对齐限制),如结构体数组中含有最大字节限制为 8*,因为其含有 double,故其对齐限制为 8*。

◆ 考点七: 内存越界引用和缓冲区溢出

1. 缓冲区溢出

可能引起缓冲区溢出的函数有: strcpy()、sprintf()、gets()、scanf()、strcat(), 这些函数不需要告诉它们目标缓冲区的大小,就产生一个字节序列: **gets()** 函数无法确定是否为保存整个字符串分配了足够的空间, **strcpy, strcat**: 任意长度字符串的拷贝, **scanf, fscanf, sscanf** 使用 %s 转换符时

2. 对抗缓冲区溢出攻击

(1) 避免溢出漏洞(函数、代码细节)

如用 fgets 代替 gets、strncpy 代替 strcpy, scanf 函数中别用 %s (或用 %ns 代替 %s, 其中 n 是一个合适的整数)

(2) 使用系统级的防护

栈随机化(随机的栈偏移), 程序启动后,在栈中分配随机数量的空间;

限制可执行代码区域(非可执行代码段), 消除攻击者向系统中插入可执行代码的能力

栈破坏检测(栈金丝雀)

二、 真题演练

(一) 选择题

- 在 X86-64 指令集体系结构中, 程序员可见的状态不包括()
A. 程序计数器(PC) B. 高速缓存 C. 整数寄存器 D. 条件码寄存器
- 下列说法不正确的是()
A. 机器级程序使用的内存地址是虚拟地址
B. 机器级程序不区分有符号数、无符号数
C. 汇编代码可以区分数据类型, 辨别整数和指针
D. X86-64 的通用寄存器可以用来存储整数数据和指针
- 下列的指令组中, 那一组指令只改变条件码, 而不改变寄存器的值? ()
A. CMP, SUB B. TEST, AND C. CMP, TEST D. LEAQ, CMP
- 下列叙述正确的是()
A. 一条 mov 指令不可以使用两个内存操作数
B. 在一条指令执行期间, CPU 不会两次访问内存
C. CPU 不总是执行 CS::RIP 所指向的指令, 例如遇到 call、ret 指令时
D. X86-64 指令"movl \$1,%eax"不会改变%rax 的高 32 位
- x86 体系结构中, 下面说法正确的是()
A. leal 指令只能够用来计算内存地址
B. 使用栈随机化的方法不能完全避免针对缓冲区溢出的攻击。
C. 在一个函数内, 改变任一寄存器的值之前必须先将其原始数据保存在栈内
D. c 函数调用过程中, 调用函数的栈帧一旦被修改, 被调用函数则无法正确返回。
- 将%rax 清零, 下列指令错误的是()
A. subq %rax, %rax B. xorq %rax, %rax
C. testq %rax,%rax D. andq \$0, %rax
- 在 X86-64 中, CPU 维护了一组单个位的条件码(CF、ZF、SF、OF 等), 下面说法不正确的是()
A. leaq 指令不会改变条件码的值, 但会改变相关寄存器的值
B. CMP、TEST 指令分别类似于 SUB、AND 指令, 会改变寄存器值并设置条件码
C. ADD、XOR 指令会改变条件码的值, 也会改变相关寄存器的值
D. 跳转指令 jne 会根据条件码 ZF 来判断是否选择分支
- 请问横线上的数字应该是()
已知函数

```
int x( int n ) {
    return n*_____;
```

 对应的汇编代码如下:

```
leaq (%rdi, %rdi, 4), %rdi
leaq (%rdi, %rdi, 1), %rax
retq
```


A. 4 B. 5 C. 2 D. 10
- 对于如下的 C 语言中的条件转移指令, 它所对应的汇编代码中至少包含几条条件转移指令:

if (a > 0 && a != 1 || a < 0 && a != -1) b=a;()

- A. 2 条 B. 3 条 C. 4 条 D. 5 条

10. 左边的 C 函数中, 在 x86_64 服务器上采用 GCC 编译产生的汇编语言如右边所示。那么 (1) 和 (2) 的内容分别是: ()

```
int arith(int x, int y) {                                <arith>:
    return (x < y) ? ( 1 ) : ( 2 );                      lea    (%rsi,%rdi,1),%eax
}                                                         mov    %esi,%edx
                                                         sub    %edi,%edx
                                                         cmp    %esi,%edi
                                                         cmovge %edx,%eax
                                                         retq
```

- A. x-y, x+y B. x+y, x-y C. x+y, y-x D. y-x, x+y

11. 下列寻址模式中, 正确的是:

- A. (%rax, , 4) B. (%rax, %rsp, 3) C. 123 D. \$1(%rbx, %rbp, 1)

12. 已知短整型数组 S 的起始地址和下标 i 分别存放在寄存器 %rdx 和 %rcx, 将 &S[i] 存放在寄存器 %rax 中所对应的汇编代码是 ()

- A. leaq (%rdx, %rcx, 1) , %rax B. movw (%rdx, %rcx, 2), %rax
C. leaq (%rdx, %rcx, 2), %rax D. movw (%rdx, %rcx, 1), %rax

13. 关于条件跳转 JXX 和条件传送 COMV 指令说法错误的是 ()

- A. 条件跳转 JXX 需要测试判断语句来选择分支, 条件传送 CMOV 不需要
B. 条件传送 CMOV 没有分支预测处罚, 所以在条件分支语句中始终优于条件跳转 JXX
C. 条件跳转 JXX 和条件传送 CMOV 都需要检查条件码的值
D. 条件跳转 JXX 和条件传送 CMOV 都有针对有符号数和无符号数的指令

14. 在如下代码段的跳转指令中, 目的地址是 ()

```
400020: 74 F0          je _____
400022: 5d             pop %rbp
```

- A. 400010 B. 400012 C. 400110 D. 400112

15. 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

16. 下面哪条指令不会引起 %rsp 的变化? ()

- A. movq %rsp, %rbp B. pushq %rbp
C. callq printf D. subq \$20, %rsp

17. 下列关于比较指令 CMP 说法中, 正确的是 ()

- A. 专用于有符号数比较 B. 专用于无符号数比较
C. 专用于串比较 D. 不区分比较的对象是有符号数还是无符号数

18. 在 x86-64 中, 有初始值 %rax = 0x1122334455667788, 执行下述指令后 %rax 寄存器的值是 ()

movl \$0xaa11, %rax

A. 0xaa11

B. 0x112233445566aa11

C. 0x112233440000aa11

D. 0x11223344ffffaa11

19. 下列指令中，寻址方式不正确的是 ()

A. MOVb %ah, 0x20(, %ecx, 8)

B. LEAl (0xA, %eax), %ebx

C. SUBb 0x1B, %bl

D. INCl (%ebx, %eax)

20. x86-64 体系结构的内存寻址方式有多种格式，请问下列指令不正确的： ()

A. movq \$34, (%rax)

B. movq (%rax), %rax

C. movq \$23, 10(%rdx, %rax)

D. movq (%rax), 8(%rbx)

21. 已知变量 x 的值已经存放在寄存器 %rax 中，现在想把 $5x+7$ 的值计算出来并存放到寄存器 %rbx 中，如果不允许用乘法和除法指令，则至少需要多少条 x86-64 指令完成该任务？ ()

A. 1 条

B. 3 条

C. 2 条

D. 4 条

22. 对简单的 switch 语句常采用跳转表的方式实现，在 x86-64 系统中，下述最有可能正确的 switch 分支跳转汇编指令为哪个 ()

A. jmp .L3(, %eax, 4)

B. jmp .L3(, %eax, 8)

C. jmp *.L3(, %eax, 4)

D. jmp *.L3(, %eax, 8)

23. 假定 struct P {int i; char c; int j; char d;}; 在 x86_64 服务器的 Linux 操作系统上，下面哪个结构体的大小与其它三个不同 ()

A. struct P1 {struct P a[3]};

B. struct P2 {int i[3]; char c[3]; int j[3]; char d[3]};

C. struct P3 {struct P *a[3]; char *c[3]};

D. struct P4 {struct P *a[3]; int *f[3]};

24. x86-64 中，某 C 程序定义了结构体

```
struct SS {
    double v;
    int i;
    short s;
} aa[10];
```

则执行 sizeof(aa) 的值是 ()

A. 14

B. 80

C. 140

D. 160

25. 考虑下面的 union 的声明，这个 union 总共大小 ()。

```
union ELE {
    struct {
        char c;
        char *p;
    } el;
    struct {
        int *q;
    }
};
```

```

        int x;
    }e2;
};
    
```

A.9 B.12 C.21 D. 9 或者 12

26. 关于如何避免缓冲区溢出带来的程序风险, 下述错误的做法为? ()

- A. 编程时定义大的缓冲区数组
- B. 编程时避免使用 gets, 而采用 fgets
- C. 程序运行时随机化栈的偏移地址
- D. 在硬件级别引入不可执行代码段的机制

27. 有时程序需要将局部数据保留在内存中, 常见的情况不包括 ()

- A. 寄存器不足够存放所有的本地数据
- B. 对一个局部变量使用地址运算符
- C. 某些局部变量是指针
- D. 某些局部变量是数组或结构

28. x86-64 系统中, 函数 `int sum (int x, int y)` 经编译后其返回值保存在 ()

A. %rdi B. %rsi C. %rax D. %rdx

29. 在 x86-64 系统中, 调用函数 `int gt (long x, long y)` 时, 保存参数 y 的寄存器是 ()

A. %rdi B. %rsi C. %rax D. %rdx

30. Linux 中对抗缓冲区溢出攻击的系统级保护机制不包括 ()

- A. 栈随机化
- B. 栈破坏检测
- C. 限制可执行代码区域
- D. 限制使用无边界检查函数

31. 递归函数程序执行时, 正确的是 ()

- A. 使用了堆
- B. 可能发生栈溢出
- C. 容易有漏洞
- D. 必须用循环计数器

(二) 填空题

1. 在 Linux 中, 利用 gcc 编译器, 使用 -Og 优化选项, 将 p1.c 和 p2.c 生成 p.o 文件的命令行为_____。

2. 在 x86-64 的 CPU 内包含了一组 16 个存储 64 位值的通用寄存器, 通常用来传递返回值的寄存器是_____, 指明运行时栈顶的寄存器是_____, 在函数调用时依次传递第 1、2、3、4 个整型参数的寄存器是_____, _____。(用四字寄存器表示)。

3. C 语言 64 位系统中参数可以采用_____、_____来传递, 它们存放着传递控制和数据、分配内存的相关信息。

4. 在 x86-64 中, 若过程调用有多个整型参数, 超过_____个的部分需要通过栈来传递, 如果通过寄存器传递, 那么可用的寄存器为_____。

5. 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_____。

(三) 简答题

1. 已知内存和寄存器中的数值情况如下:

| 内存地址 | 值 |
|-------|------|
| 0x100 | 0xff |

| 寄存器 | 值 |
|------|-------|
| %rax | 0x100 |

| | |
|-------|------|
| 0x104 | 0xAB |
| 0x108 | 0x13 |
| 0x10c | 0x11 |

| | |
|------|-----|
| %rcx | 0x1 |
| %rdx | 0x3 |
| | |

请填写下表，给出对应操作数的值：

| 操作数 | 值 | 操作数 | 值 |
|---------------|---|----------------|---|
| %rax | | 0x104 | |
| (%rax) | | \$0x108 | |
| 9(%rax,%rdx) | | 4(%rax) | |
| 0xfc(,%rcx,4) | | 260(%rcx,%rdx) | |
| (%rax,%rdx,4) | | | |

2. 在 X86-64 中有许多算术逻辑操作可供使用，根据所学知识回答以下问题

(1) 根据提示写出 CSAPP 中所列出的算术和逻辑指令

| 指令 | 效果 | 描述 | 指令 | 效果 | 描述 |
|----|-----------------------|--------|----|--------------------------|------|
| | $D \leftarrow \&S$ | 加载有效地址 | | $D \leftarrow D \ll k$ | 左移 |
| | $D \leftarrow D+1$ | 加 1 | | $D \leftarrow D \ll k$ | 左移 |
| | $D \leftarrow D-1$ | 减 1 | | $D \leftarrow D \gg_A k$ | 算术右移 |
| | $D \leftarrow -D$ | 取负 | | $D \leftarrow D \gg_L k$ | 逻辑右移 |
| | $D \leftarrow \sim D$ | 取补 | | | |
| | | 加法 | | | 异或 |
| | | 减法 | | | 或 |
| | | 乘法 | | | 与 |

(2) 填写下表关于 leaq 指令的内容，指明每条指令存储在寄存器 %rax 中的值

| | |
|-----------------------------|--|
| <u>x in %rdx, y in %rbx</u> | |
| leaq 9(%rdx), %rax | |
| leaq (%rdx,%rbx), %rax | |
| leaq (%rdx,%rbx,4), %rax | |
| leaq 2(%rbx,%rbx,8), %rax | |
| leaq 0xE(,%rdx,4), %rax | |
| leaq 6(%rbx,%rdx,2), %rax | |

(3) 关于一元和二元操作，给出下面指令的效果，说明将被更新的寄存器或内存位置，以及得到的值。

| 内存地址 | 值 |
|-------|------|
| 0x100 | 0xFF |
| 0x108 | 0xAB |
| 0x110 | 0x13 |

| 寄存器 | 值 |
|------|-------|
| %rax | 0x100 |
| %rcx | 0x1 |
| %rdx | 0x3 |

| | |
|-------|------|
| 0x118 | 0x11 |
|-------|------|

| | |
|--|--|
| | |
|--|--|

| 指令 | 目的 | 值 |
|-----------------------------|----|---|
| addq %rcx, (%rax) | | |
| subq %rdx, 8(%rax) | | |
| imulq \$16, (%rax, %rdx, 8) | | |
| incq 16(%rax) | | |
| decq %rcx | | |
| subq %rdx, %rax | | |

(4) 关于移位运算, 假设我们想生成以下 c 函数的汇编代码

```
long shift_left4_rightn(long x, long n) {
    x <<= 4;
    x >>= n;
    return x;
}
```

下面这段汇编代码执行实际的移位, 并将最后的结果放在寄存器%rax 中。此处省略了两条关键的指令。参数 x 和 n 分别存放在寄存器%rdi 和 %rsi 中。

long shift_left4_rightn(long x, long n)

x in %rdi, n in %rsi

shift_left4_rightn:

movq %rdi, %rax

movl %esi, %ecx

填写上面缺失的指令。请使用算术右移操作。

(5) 综合上面的讨论, 请根据汇编代码完成 c 语言代码

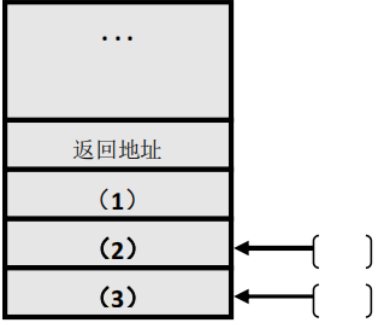
| | |
|--|---|
| <p><u>long arith(long x, long y, long z)</u></p> <p><u>x in %rdi, y in %rsi, z in %rdx</u></p> <p>arith:</p> <p> xorq %rsi, %rdi</p> <p> leaq (%rdx,%rdx,2), %rax</p> <p> salq \$4, %rax</p> <p> andl 0x0F0F0F0F, %edi</p> <p> subq %rdi, %rax</p> <p> ret</p> | <p>long arith(long x, long y, long z)</p> <p>{</p> <p> long t1 = _____;</p> <p> long t2 = _____;</p> <p> long t3 = _____;</p> <p> long t4 = _____;</p> <p> _____;</p> <p>}</p> |
| <p>arith:</p> <p> orq %rsi,%rdi</p> | <p>long arith(long x, long y, long z)</p> <p>{</p> |

| | |
|---|--|
| <pre> sarq \$3,%rdi notq %rdi movq %rdx,%rax subq %rdi,%rax retq </pre> | <pre> long t1 = _____; long t2 = _____; long t3 = _____; long t4 = _____; _____ </pre> |
| <pre> arith: xorq %rsi,%rdi leaq (%rdi,%rdi,4),%rax leaq (%rax,%rsi,2),%rax subq %rdx,%rax retq </pre> | <pre> long arith(long x, long y, long z) { long t1 = _____; long t2 = _____; long t3 = _____; long t4 = _____; _____; } </pre> |
| <p>[2020]分析如下子程序，写出其对应的 C 语言程序（函数名，参数名可自定义）。</p> <pre> ox40114a mov (%rsi), %eax ox40114c neg %eax ox40114e mov (%rdi), %edx ox401150 neg %edx ox401152 mov %edx, (%rsi) ox401154 mov %eax, (%rdi) ox401156 retq </pre> | |

3. 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？并填写 `sum` 函数的栈帧（x86-64 形式）。

| |
|------|
| x8 |
| x7 |
| 返回地址 |
| |
| |
| |

4. **[2018B]**函数 `call_incr2` 的汇编代码如下所示，画出函数 `call_incr2` 相应的栈帧内结构与内容。

| | |
|--|--|
| <pre> call_incr2: pushq %rbx subq \$16, %rsp movq %rdi, %rbx movq \$15213, 8(%rsp) movl \$3000, %esi leaq 8(%rsp), %rdi call incr addq %rbx, %rax addq \$16, %rsp popq %rbx ret </pre> |  <p>请填写出上述栈帧缺失的内容</p> <p>(1) _____</p> <p>(2) _____</p> <p>(3) _____</p> <p>(4) _____</p> <p>(5) _____</p> |
|--|--|

5. 阅读下列代码，回答后面的问题

| | |
|---|--|
| <pre> typedef struct { short x[A][B]; int y; }str1; typedef struct { char array[B]; int t; short s[B]; int u; }str2; (short 以 2 字节计算) </pre> | <pre> void setVal(str1 *p, str2 *q) { int v1=q->t; int v2=q->u; p->y=v1+v2; } </pre> <p>GCC 为 setVal 的主体产生下面的代码</p> <pre> movl 12(%ebp),%eax movl 28(%eax),%edx addl 8(%eax),%edx movl 8(%ebp),%eax movl %edx,44(%eax) </pre> |
|---|--|

请直接写出 A 和 B 的值各是多少？

A=_____; B=_____.

(四) 综合分析题

1. C 语言提供了许多机制来实现程序的控制转移，如条件分支、条件传送、循环等，下面根据相关知识回答下列问题

(1) C 语言程序中，常常利用 if-else 语句来实现条件分支，下面根据汇编代码逆向工程条件分支

| | |
|---|---|
| <p>1) 请根据汇编代码填写下面的 C 代码：</p> <pre> long test(long x, long y, long z){ long val = _____; if (_____) { if (_____) val = _____; } } </pre> | <p>左边的 C 代码编译的汇编代码如下：</p> <pre> long test(long x, long y, long z) x in %rdi, y in %rsi, z in %rdx test: leaq (%rdi,%rsi), %rax addq %rdx, %rax cmpq \$-3, %rdi </pre> |
|---|---|

| | |
|---|---|
| <pre> else val = _____; } else if (_____) val = _____; return val; } </pre> | <pre> jge .L2 cmpq %rdx, %rsi jle .L3 movq %rdi, %rax imulq %rsi, %rax ret .L3: movq %rsi, %rax imulq %rdx, %rax ret .L2: cmpq \$2, %rdi jge .L4 movq %rdi, %rax imulq %rdx, %rax .L4: rep; ret </pre> |
| <p>2) 请根据汇编代码填写下面的 C 代码:</p> <pre> long test(long x, long y) { long val = _____; if (_____) { if (_____) val = _____; else val = _____; } else if (_____) val = _____; return val; } </pre> | <p>左边的 C 代码编译的汇编代码如下:</p> <pre> <u>long test(long x, long y)</u> <u>x in %rdi, y in %rsi</u> test: leaq 0(,%rdi,8), %rax testq %rsi, %rsi jle .L2 movq %rsi, %rax subq %rdi, %rax movq %rdi, %rdx andq %rsi, %rdx cmpq %rsi, %rdi cmovge %rdx, %rax ret .L2: addq %rsi, %rdi cmpq \$-2, %rsi cmovle %rdi, %rax ret </pre> |

(2) C 语言提供多种循环及分支结构, 如 while 循环、do-while 循环、for 循环和 switch 语句(多重分支), 请完成下面有关循环的练习

1) 这是一个 while 循环, 根据右边的汇编代码可知编译器采用了_____的翻译方法。

| | |
|---|--|
| <p>请根据汇编代码填写下面的 C 代码:</p> <pre> long loop_while(long a, long b){ long result = _____; while (_____) { result *= _____; } } </pre> | <p>左边的 C 代码编译的汇编代码如下:</p> <pre> <u>long loop_while(long a, long b)</u> <u>a in %rdi, b in %rsi</u> loop_while: movl \$1, %eax jmp .L2 </pre> |
|---|--|

| | |
|--|--|
| <pre> a = _____; } return result; } </pre> | <pre> .L3: leaq (%rdi,%rsi), %rdx imul %rdx, %rax addq \$1, %rdi .L2: cmpq %rsi, %rdi jl .L3 rep; ret </pre> |
|--|--|

2) 这是一个 while 循环，根据右边的汇编代码可知编译器采用了_____的翻译方法代码

| | |
|---|---|
| <p>填写下面的 C 代码：</p> <pre> long loop_while2(long a, long b){ long result = _____; while (_____) { result *= _____; b = _____; } return result; } </pre> | <p>左边的 C 代码编译的汇编代码如下：</p> <p><u>a in %rdi, b in %rsi</u></p> <pre> loop_while2: testq %rsi, %rsi jle .L8 movq %rsi, %rax .L7: imulq %rdi, %rax subq %rdi, %rsi testq %rsi, %rsi jg .L7 rep; ret .L8: movq %rsi, %rax ret </pre> |
|---|---|

3) 下面两个同样是 while 循环，请根据汇编代码填写下面的 C 代码：

| | |
|--|--|
| <p>(1) 请填写缺失的 C 语言代码：</p> <pre> long fun_a(unsigned long x) { long val = 0; while (_____) { _____; _____; } return _____; } </pre> | <p>左边的 C 代码编译的汇编代码如下：</p> <p><u>long fun a(unsigned long x)</u></p> <p><u>x in %rdi</u></p> <pre> fun_a: movl \$0, %eax jmp .L5 .L6: xorq %rdi, %rax shrq %rdi .L5: testq %rdi, %rdi jne .L6 andl \$1, %eax ret </pre> |
| <p>(2) 请填写缺失的 C 语言代码：</p> <pre> long sum(long *start, long count){ long sum = 0; while (_____) { sum += _____; } } </pre> | <p>左边的 C 代码编译的汇编代码如下：</p> <p><u>long sum(long *start, long count)</u></p> <p><u>start in %rdi, count in %rsi</u></p> <pre> sum: movl \$0, %eax jmp .L2 </pre> |

| | |
|---|--|
| <pre> _____; _____; } return sum; } </pre> | <pre> .L3: addq (%rdi), %rax addq \$8, %rdi subq \$1, %rsi .L2: testq %rsi, %rsi jne .L3 rep; ret </pre> |
|---|--|

4) 这是一个 for 循环，在一定情况下其等价于 while 循环，请回答下面的问题

| | |
|---|---|
| <p>请根据汇编代码填写下面的 c 代码：</p> <pre> long fun_b(unsigned long x) { long val = 0; short i; for (_____; _____ ; _____) { _____; _____; _____; } return val; } </pre> | <p>左边的 c 代码编译的汇编代码如下：</p> <pre> long fun b(unsigned test x) x in %rdi fun_b: movl \$64, %edx movl \$0, %eax .L10: movq %rdi, %rcx andl \$1, %ecx addq %rax, %rax orq %rcx, %rax shrq %rdi addq \$1, %rdx jne .L10 rep; ret </pre> |
|---|---|

5) 下面是一个 switch 语句的 C 代码和汇编语言，由于由多重分支，看起来比循环更为复杂，请回答下列问题并根据汇编代码填写下面的 c 语句，除了情况标号 C 和 D 的顺序之外，将不同情况填入这个模板的方式是唯一的

| | |
|--|---|
| <p>(1) Switch 语句中情况标号的值分别是：</p> <p>case: _____;</p> <p>(2) 哪些情况有多个标号？ case: _____;</p> <p>(3) 请填写缺失的 C 语言代码：</p> <pre> void switcher(long a, long b, long c, long *dest){ long val; switch(a) { case_____: /* Case A */ c = _____; /* Fall through */ case_____: /* Case B */ val = _____; break; } } </pre> | <p>左边的 c 代码编译的汇编代码如下：</p> <pre> void switcher(long a, long b, long c, long *dest) a in %rsi, b in %rdi, c in %rdx, d in %rcx switcher: cmpq \$7, %rdi ja .L2 jmp *.L4(,%rdi,8) .section .rodata .L7: xorq \$15, %rsi movq %rsi, %rdx .L3: leaq 112(%rdx), %rdi jmp .L6 .L5: leaq (%rdx,%rsi), %rdi </pre> |
|--|---|

| | |
|---|--|
| <pre> case____: /* Case C */ case____: /* Case D */ val = _____; break; case____: /* Case E */ val = _____; break; default: val = _____; } *dest = val; } </pre> | <pre> salq \$2, %rdi jmp .L6 .L2: movq %rsi, %rdi .L6: movq %rdi, (%rcx) ret .L4: .quad .L3 .quad .L2 .quad .L5 .quad .L2 .quad .L6 .quad .L7 .quad .L2 .quad .L5 </pre> |
|---|--|

2. 递归过程，利用栈实现递归地调用他们自身，每个过程在栈中都有相应的私有空间，使得各自局部变量不会相互干扰，根据递归的相关知识完成下面的练习

| | |
|---|---|
| <p>请根据汇编代码填写下面的 c 代码：</p> <pre> long rfun(unsigned long x) { if (_____) return_____; unsigned long nx = _____; long rv = rfun(nx); return _____; } </pre> | <p>左边的 c 代码编译的汇编代码如下：</p> <pre> <u>long rfun(unsigned long x)</u> <u>x in %rdi</u> rfun: pushq %rbx movq %rdi, %rbx movl \$0, %eax testq %rdi, %rdi je .L2 shrq \$2, %rdi call rfun addq %rbx, %rax .L2: popq %rbx ret </pre> |
|---|---|

3. 结构体是 C 语言中常见的一种数据结构，其能高效地组织数据，下面是关于结构体的一些练习。

1) 考虑下面的结构声明

| | |
|---|--|
| <pre> struct prob { int *p; struct { int x; int y; } } </pre> | <p>下面的过程（省略了某些表达式）对这个结构进行操作：</p> <pre> void sp_init(struct prob *sp) { st->s.x = _____; st->p = _____; st->next = _____; } </pre> |
|---|--|

要求:

A. `struct P1 { int i; char c; char d; long d; };`

B. `struct P2 { short w[3]; char c[3] };`

C. `struct P5 { struct P3 a[2]; struct P2 t };`

(2) 对于下列结构声明回答后续问题:

| | |
|--|---|
| <pre> struct { char *a; short b; double c; char d; float e; char f; long g; int h; } rec; </pre> | <p>(1) 这个结构中所有的字段的字节偏移量是多少?</p> <p>(2) 这个结构总的大小是多少?</p> <p>(3) 重新排列这个结构中的字段, 以最小化浪费的空间, 然后再给出重排过的结构的字节偏移量和总的大小。</p> |
|--|---|

4. 请完成下面有关缓冲区溢出的相关练习。

1) [2018A] 简述缓冲区溢出攻击的原理 (可结合 `gets()` 说明) 以及防范方法。

2) [2019A] 下列 C 程序存在安全漏洞, 请给出攻击方法。如何修复或防范 (给出具体的方法)?

```

int getbuf(char *s) {
    char buf[32];
    strcpy( buf, s );
}
    
```

3) [2020] 下列子程序存在缓冲器溢出漏洞, 请分析漏洞产生的原因, 说明如何攻击?

```

0x401152 push %rbp
0x401153 mov %rsp, %rbp
0x401156 sub $0x28, %rsp
0x40115a mov %rdi, %rsi
0x40115d lea -0x20(%rbp), %rdi
0x401161 callq 0x401030<strcpy@plt>
0x401166 lea -0x20(%rbp), %rdi
0x40116a callq 0x401040<puts@plt>
0x40116f leaveq
0x401170 retq
    
```

第五章、 优化程序性能

一. 真题考点

◆ 考点一：基于编译器的优化

1. 编译器的代码优化局限性

内存别名使用：两个指针可能指向同一个内存位置的情况

函数调用：如果函数有副作用，那么编译器不会对其进行优化

2. 基于编译器的优化

| 优化 | 作用 | 例子 |
|------------|--|-------------------------------------|
| 代码移动 | 识别要执行多次（例如在循环里）但是计算结果不会改变的计算。因而可以将计算移动到代码前面不会被多次求值的部分。 | 用 ni 代替 $n*i$ |
| 复杂运算简化 | 用更简单的方法替换昂贵的操作 | 移位、加，替代乘法/除法 |
| 共享公用子表达式 | 重用表达式的一部分 | |
| 减少过程调用 | 尽可能地减少函数调用 | <code>strlen()</code> 移除循环 |
| 消除不必要的内存引用 | 引入局部累计变量 | <code>acc</code> 、 <code>sum</code> |

3. 基于处理器的优化

（1）功能单元的指标

延迟：表示完成运算所需要的总时间

发射时间：表示两个连续的同类型的运算之间需要的最小的时钟周期数

容量：表示能够执行该运算的功能单元的数量

（2）优化方法

| 优化 | 作用 | 例子 |
|---------------------------------|----------------------|---|
| 循环展开 ($k*1$) | 增加每次迭代计算的元素的数量 | <pre>for (i = 0; i < n-1; i+=2) { acc=(acc OP data[i]) OP data[i+1]; }</pre> |
| 提高并行性—— 多个累积变量 ($k*k$) | 使用多个累计变量 | <pre>for (i = 0; i < n-1; i+=2) { acc0 = acc0 OP data[i]; acc1 = acc1 OP data[i+1];}</pre> |
| 提高并行性—— 重新结合变换 ($k*1a$) | 打破顺序相关从而使性能提高到延迟界限之外 | <pre>for (i = 0; i < limit; i+=2) { acc = acc OP (data[i] OP data[i+1]);}</pre> |

二. 真题演练

(一) 选择题

1. 下列说法正确的是 ()

- A. 同一个任务采用时间复杂度为 $O(\log N)$ 算法一定比采用复杂度为 $O(N)$ 算法的执行时间短
- B. 编译器进行程序优化时, 总是可以使用算数结合律来减少计算量
- C. 增大循环展开 (loop unrolling) 的级数, 有可能降低程序的执行性能 (即增加执行时间)
- D. 分支预测时, “总是预测不跳转” (branch not taken) 一定比 “总是预测跳转” (branch taken) 预测准确率高

2. 【双选】以下哪些程序优化编译器总是可以自动进行? (假设 `int i, int j, int A[N], int B[N], int m` 都是局部变量, `N` 是一个整数型常量, `int foo(int)` 是一个函数) ()

| | 优化前 | 优化后 |
|----|---|--|
| A. | <pre>for (j = 0 ; j < N ; j ++) m += i*N*j;</pre> | <pre>int temp = i*N; for (j= 0 ; j < N ; j ++) m + = temp * j;</pre> |
| B. | <pre>for (j = 0 ; j < N ; j ++) B[i] *= A[j];</pre> | <pre>int temp= B[i]; for (j= 0 ; j < N ; j ++) temp *= A[j]; B[i] = temp;</pre> |
| C. | <pre>for (j = 0 ; j < N ; j ++) m = (m + A[j]) + B[j];</pre> | <pre>for (j = 0 ; j < N ; j ++) m = m + (A[j] + B[j]);</pre> |
| D. | <pre>for (j = 0 ; j < foo(N) ; j ++) m ++;</pre> | <pre>int temp = foo(N); for (j= 0 ; j < temp ; j ++) m ++;</pre> |

3. 根据编译器安全优化的策略, 如下手工程序代码的优化, 哪个达不到优化效果? ()

- A. 循环展开, 以减少循环的迭代次数
- B. 将函数调用移到循环内, 以提高程序的模块性
- C. 消除不必要的存储器引用, 减少访存开销
- D. 分离多个累计变量, 以提高并行性

4. 下面关于程序性能的说法中, 哪种是正确的? ()

- A. 处理器内部只要有多个功能部件空闲, 就能实现指令并行, 从而提高程序性能。
- B. 同一个任务采用时间复杂度为 $O(\log N)$ 算法一定比采用复杂度为 $O(N)$ 算法的执行时间短
- C. 转移预测总是能带来好处, 不会产生额外代价, 对提高程序性能有帮助。
- D. 增大循环展开 (loop unrolling) 的级数, 有可能降低程序的性能 (即增加执行时间)

5. 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

6. 利用 GCC 生成代码过程中, 不属于编译器优化的结果是 ()

A. 用移位操作代替乘法指令

B. 消除循环中的函数调用

C. 循环展开

D. 使用分块提高时间局部性

7. 对于下面这段程序, 哪个描述是正确的 ()

```

void upper(char *s){
    for(int i=0; i<strlen(s); i++){
        if(s[i]>='a' && s[i]<='z')
            s[i] += ('A' - 'a');
    }
}

```

A. 假设 s 的长度为 N, 该程序的时间复杂度为 $O(N\log N)$

B. 将 “for(int i=0; i<strlen(s); i++){ }” 修改为: “int len = strlen(s); for(int i=0; i<len; i++){ }” 可以使程序运行时间与 s 的长度线性相关

C. B 选项中的修改策略不影响程序的空间局部性与时间局部性

D. B 选项中的修改策略仅影响程序的空间局部性, 不影响时间局部性

(二) 简答题

1. [2017B、2018B] 列举至少种程序优化的方法 (基于编译器和处理器), 并简述其原理。

2. 下面有一个函数:

```

double poly( double a[] ,double x, int degree){
    long int i;
    double result = a[0];
    double xpwr =x;
    for(i=1 ; i<=degree; i++){
        result += a[i] *xpwr;
    }
}

```

```

        xpwr = x*xpwr;
    }
    return result;
}

```

1) 当 degree=n, 这段代码共执行多少次加法和多少次乘法?

2) 在 CSAPP 中参考机中, 算术运算的延迟如下图所示, 这个函数的 CPE = 5.00。解释为什么会得到这样的 CPE? (结合关键路径说明)

| Operation | Integer | | | Floating point | | |
|----------------|---------|-------|----------|----------------|-------|----------|
| | Latency | Issue | Capacity | Latency | Issue | Capacity |
| Addition | 1 | 1 | 4 | 3 | 1 | 1 |
| Multiplication | 3 | 1 | 1 | 5 | 1 | 2 |
| Division | 3-30 | 3-30 | 1 | 3-15 | 3-15 | 1 |

(三) 综合分析题

1. [2017A]程序优化: 矩阵 $c[n][n] = a[n][n] * b[n,n]$, 采用题首 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];
    }
}

```

请给出基于编译、CPU、存储器的三种优化方法, 并编写程序

2. [2018]现代超标量 CPU X86-64 的 Cache 的参数 $s=5$, $E=1$, $b=5$, 若 $M=N=64$, 请优化如下程序, 并说明优化的方法 (至少 CPU 与 Cache 各一种)。

```
void trans(int M, int N, int A[M][N], int B[N][M]){
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            B[j][i] = A[i][j];
}
```

3. [2019]优化如下程序, 给出优化结果并说明理由。

```
int sum_array(int a[M][N][N]) {           //M、N 足够大
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

4. 如下是使用 C 语言描述的链表结构的声明，链表的结尾使用空指针来表示。同时使用函数 `int length (List *p)` 来计算链表的长度。为简化起见，假设该链表是非循环的。

```
typedef struct LIST {
    struct LIST *next;
    int data;
} List;
```

1) 函数 `count_pos1` 用来计算链表中 `data` 为正数的元素个数，并将结果存放在地址 `k`。以下的程序可能存在问题导致效率很低或程序出错，请指出并修改。

```
void count_pos1 (List *p, int *k) {
    int i;
    for (i = 0; i < length(p); i++) {
        if (p->data > 0)
            *k++;
        p = p->next;
    }
}
```

2) 为提高程序性能，可以考虑删除变量 `i` 以消除函数调用。请修改上述程序达到该目的。

3) 上述程序内层循环的汇编片段如下所示。假设该链表不为空且大部分数据都为正数，转移预测全部正确，设计中有足够多的部件来实现指令并行。其中访存操作全部 `cache` 命中，时延为 3 cycle，其他指令时延为 1cycle。请计算以下程序的 CPE 下限，并给出文字说明。

| | |
|---|---|
| <pre>.L1: movl 4(%eax), %ecx testl %ecx, %ecx jle .L2 incl %edx</pre> | <pre>.L2: movl (%eax), %eax testl %eax, %eax jne .L1</pre> |
|---|---|

5. [2017B] 向量元素和计算的相关程序如下，请改写或重写计算函数 `vector_sum`，进行速度优化，并简要说明优化的依据。

```
/*向量的数据结构定义 */
typedef struct{
    int len;           //向量长度，即元素的个数
    float *data;       //向量元素的存储地址
} vec;
/*获取向量长度*/
int vec_length(vec *v){return v->len;}
/* 获取向量中指定下标的元素值，保存在指针参数 val 中*/
int get_vec_element(*vec v, size_t idx, float *val){
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
/*计算向量元素的和*/
void vector_sum(vec *v, float *sum){
    long int i;
    *sum = 0;           //初始化为 0
    for (i = 0; i < vec_length(v); i++) {
        float val;
        get_vec_element(v, i, &val);    //获取向量 v 中第 i 个元素的值，存入 val
中
        *sum = *sum + val;               //将 val 累加到 sum 中
    }
}
```

第六章、 存储器层次结构

一、 真题考点

◆ 考点一：局部性

1. 局部性原理

程序倾向于引用邻近于其他最近引用过的数据项的数据项，或者最近引用过的数据项本身。

局部性通常有两种不同的形式：**时间局部性**和**空间局部性**，在一个具有良好时间局部性的程序中，被引用过一次的内存位置很可能在不远的将来再被多次引用。在一个具有良好空间局部性的程序中，如果一个内存位置被引用了一次，那么程序很可能在不远的将来引用附近的一个内存位置。

2. 基于局部性的优化

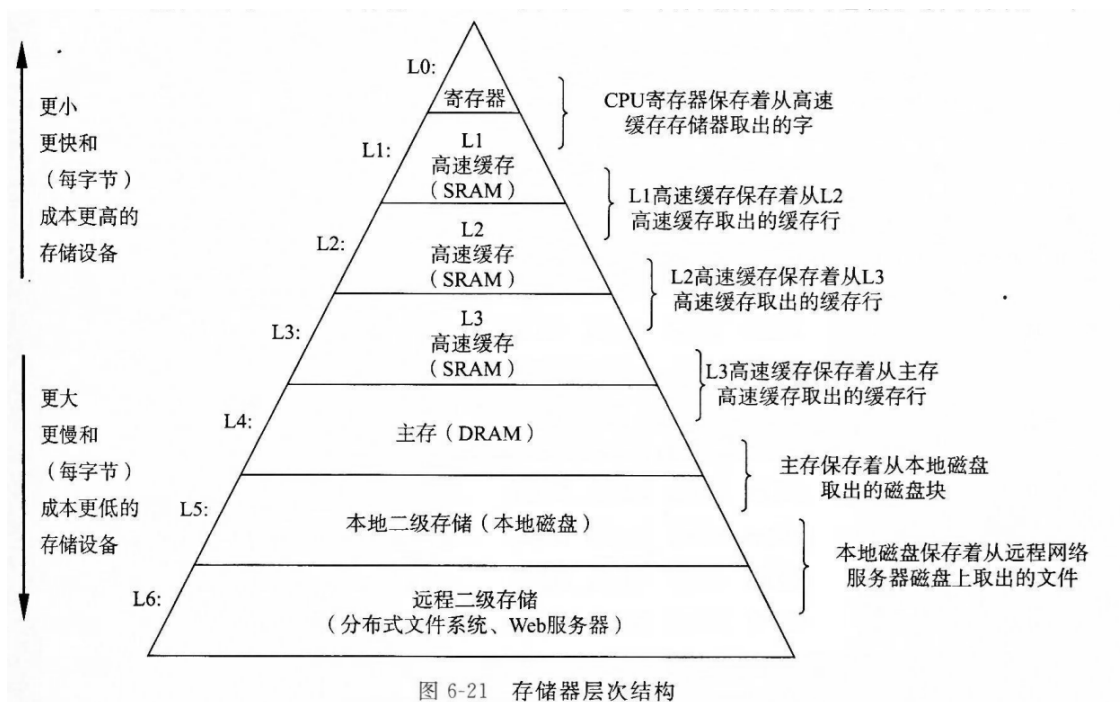
重复引用相同变量的程序有良好的时间局部性。

对于具有步长为 l 的引用模式的程序，步长越小，空间局部性越好。

对于取指令来说，循环有好的时间和空间局部性。循环体越小，循环迭代次数越多，局部性越好。

◆ 考点二：存储器层次结构

1. 存储器层次结构



2. 命中和不命中

缓存不命中的种类：

冷不命中：如果第 k 层的缓存是空的（冷缓存），那么对任何数据对象的访问都会不命中。

冲突不命中：限制性的放置策略引起的不命中

容量不命中：工作集的大小超过缓存的大小

3. 缓存管理

编译器管理**寄存器文件**，缓存层次结构的最高层：**L1、L2 和 L3** 层的缓存完全是由内置在缓存中的**硬件逻辑**来管理的；**DRAM** 主存作为存储在磁盘上的数据块的缓存，是由**操作系统软件**和

CPU 上的地址翻译硬件共同管理的

| 类型 | 缓存什么 | 被缓存在何处 | 延迟（周期数） | 由谁管理 |
|--------|----------|------------|---------|---------|
| CPU寄存器 | 4字节或8字节字 | 芯片上的CPU寄存器 | 0 | 编译器 |
| TLB | 地址翻译 | 芯片上的TLB | 0 | 硬件MMU |
| L1高速缓存 | 64字节块 | 芯片上的L1高速缓存 | 4 | 硬件 |
| L2高速缓存 | 64字节块 | 芯片上的L2高速缓存 | 10 | 硬件 |
| L3高速缓存 | 64字节块 | 芯片上的L3高速缓存 | 50 | 硬件 |
| 虚拟内存 | 4KB页 | 主存 | 200 | 硬件 + OS |

◆ 考点三：高速缓存存储器（cache）

1. Cache 的种类

| | 每组行数（E） | 组数（S） | 块大小（B） | 特点 |
|----------|---------------|-------|--------|------------|
| 直接映射高速缓存 | 1 | 2^s | 2^b | 每组只有 1 行 |
| 组相联高速缓存 | $C / (S * B)$ | 2^s | 2^b | 有多个组，每组有多行 |
| 全相联高速缓存 | C / B | 1 | 2^b | 只有一组 |

2. 高速缓存参数的性能影响

| 影响因素 | 影响结果 |
|-----------|--|
| 高速缓存大小 | 较大的高速缓存可能会提高命中率，但可能会增加命中时间 |
| 块大小 | 较大的块能利用程序中可能存在的空间局部性，帮助提高命中率。不过，对于给定的高速缓存大小，块越大就意味着高速缓存行数越少，这会损害时间局部性比空间局部性更好的程序中的命中率。较大的块对不命中处罚也有负面影响 |
| 相联度（参数 E） | 较高的相联度的优点是降低了高速缓存由于冲突不命中出现抖动的可能性 较高的相联度会增加命中时间，还会增加不命中处罚 |

◆ 考点四：基于 Cache 的程序优化方法

1. 提高时间局部性

2. 提高空间局部性

重新排列循环

| | |
|---|---|
| <pre> for (i = 0; i < n; i++){ for (j = 0; j < n; j++) { sum = 0.0; for (k = 0; k < n; k++) sum += A[i][k]*B[k][j]; C[i][j] += sum; } } </pre> | <pre> for (i = 0; i < n; i++){ for (k = 0; k < n; k++) { r = A[i][k]; for (j = 0; j < n; j++) C[i][j] += r*B[k][j]; } } </pre> |
|---|---|

二、 真题演练

(一) 选择题

1. 如果直接映射高速缓存 (Cache) 的大小是 4KB, 并且块大小 (block) 大小为 32 字节, 而且数据访问的地址序列为 0->4->16->132->232->4096->60 (以字节为单位), 请问一共发生多少次替换? ()

- A. 0 B. 1 C. 2 D. 3

2. 如果直接映射高速缓存大小是 4KB, 并且块大小为 32 字节, 请问它每组有多少行? ()

- A. 128 B. 64 C. 32 D. 1

3. 关于局部性 (locality) 的描述, 不正确的是 ()

- A. 数组通常具有很好的时间局部性 B. 数组通常具有很好的空间局部性
C. 循环通常具有很好的时间局部性 D. 循环通常具有很好的空间局部性

4. 通常情况下, 下面的哪些表述是正确的? ()

- A. 在一次读操作中, 返回的内容由高速缓存中的信息块决定
B. 高速缓存利用了时间局部性
C. 大部分情况下, 缓存需要用户程序采取显式的管理行为
D. 一级高速缓存更看重命中率, 二级高速缓存更看重命中时

5. 在代码中, 变量 sum 具有的特性是: ()

```
int sumvec(int v[N]){
    int i, sum = 0;
    for (i = 0; i < N; i ++){
        sum += v[i];
    }
    return sum;
}
```

- A. 良好的时间局部性 B. 良好的空间局部性
C. 同时具有良好的时间局部性和空间局部性 D. 不具有局部性

6. 以下关于存储结构的讨论, 那个是正确的 ()

- A. 增加额外一级存储, 数据存取的延时一定不会下降
B. 增加存储的容量, 数据存取的延时一定不会下降
C. 增加额外一级存储, 数据存取的延时一定不会增加
D. 以上选项都不正确

7. 关于 cache 的 miss rate, 下面那种说法是错误的 ()

- A. 保持 E 和 B 不变, 增大 S, miss rate 一定不会增加
B. 保持总容量和 B 不变, 提高 E, miss rate 可能不会增加
C. 保持总容量和 E 不变, 提高 B, miss rate 一定不会增加
D. 如果不采用“LRU”, 使用“随机替换策略”, miss rate 可能会降低

8. 下面关于存储器的说法, 错误的是 ()

- A. SDRAM 的速度比 FPM DRAM 快

- B. SDRAM 的 RAS 和 CAS 请求共享相同的地址引脚
- C. 磁盘的寻道时间和旋转延迟大致在一个数量级
- D. 固态硬盘的随机读写性能基本相当
9. 某磁盘的旋转速率为 7200 RPM, 每条磁道平均有 400 扇区, 则一个扇区的平均传送时间为 ()
- A. 0.02 ms B. 0.01 ms C. 0.03 ms D. 0.04 ms
10. 某高速缓存 ($E=2, B=4, S=16$), 地址宽度 14, 当引用地址 0x9D28 处的 1 个字节时, tag 位应为 ()
- A. 01110100 B. 001110000 C. 1110000 D. 10011100
11. 关于高速缓存的说法正确的是 ()
- A. 直写 (write through) 比写回 (write back) 在电路实现上更复杂
- B. 固定的高速缓存大小, 较大的块可提高时间局部性好的程序的命中率
- C. 随着高速缓存组相联度的不断增大, 失效率不断下降
- D. 以上说法全不正确
12. 某机器内存为 8 位地址, cache 设计参数为 $E=2, t=2, s=2, b=4$, cache 块大小为 16 字节。cache 为空的初始状态下, 数据访问的地址序列为 0->4->34->162->128->192->2 (以字节为单位), 请问一共发生多少次 cache 命中? ()
- A. 0 B. 1 C. 2 D. 3
13. 位于存储器层次结构中的最顶部的是 ()。
- A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存
14. CPU 一次访存时, 访问了 L1、L2、L3 Cache 所用地址 A1、A2、A3 的关系 ()
- A. $A1 > A2 > A3$ B. $A1 = A2 = A3$ C. $A1 < A2 < A3$ D. $A1 = A2 < A3$
15. 下列各种存储器中存储速度最快的是 ()。
- A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存
16. 采用缓存系统的原因是 ()
- A. 高速存储部件造价高 B. 程序往往有比较好的空间局部性
- C. 程序往往有比较好的时间局部性 D. 以上都对
17. 下列说法正确的是 ()
- A. 全相联 Cache 不会发生冲突不命中的情况
- B. 直接映射 Cache 一定会发生冲突不命中的情况
- C. 冲突不命中、容量不命中、冷不命中均可通过调整 cache 来避免
- D. Cache 的大小对程序运行非常重要, 必要的时候可以通过操作系统提高 Cache 的大小
18. CPU 寄存器作为计算机缓存层次结构的最高层, 决定哪个寄存器存放某个数据的是 ()
- A. MMU B. 操作系统内核 C. 编译器 D. CPU
19. 如果缓存命中时间为 1 个周期, 不命中处罚为 100 个周期, 则 99% 的命中率和 97% 的命中率的平均访问时间大约为 ()
- A. 2 周期、2 周期 B. 2 周期、4 周期 C. 4 周期、4 周期 D. 4 周期、8 周期

20. 以下各类存储器属于非易失性存储器的是 ()

- A. DRAM B. SRAM C. ROM D. RAM

21. 关于存储器层次结构说法错误的是 ()

- A. 寄存器文件是缓存层次结构的最高层, 由编译器管理。
B. L1、L2 和 L3 层的缓存完全是由内置在缓存中的硬件逻辑来管理的
C. DRAM 主存作为存储在磁盘上的数据块的缓存, 是由操作系统软件和 CPU 上的地址翻译硬件共同管理的
D. TLB 是页表 (PTE) 的缓存, 其由 CPU 进行管理

22. 下列各种存储器中存储速度最慢的是 ()

- A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存

(二) 填空题

1. 某 CPU 主存地址 32 位, 高速缓存总大小为 4K 行, 块大小 16 字节, 采用 4 路组相连, 则标记位的总位数 (每行标记位数*总行数) 是_____。
2. Cache 命中率分别是 97% 和 99% 时, 访存速度差别_____ (很大/很小)。
3. 当工作集的大小超过高速缓存的大小时, 会发生_____不命中。
4. I7 的 CPU, L2 Cache 为 8 路的 2M 容量, B=64, 则其 Cache 组的位数 s=_____
5. 在计算机的存储体系中, 速度最快的是_____。
6. 存储器层次结构中, 高速缓存 (Cache) 是_____的缓存。
7. 一个磁盘有 2 个盘片, 10000 个柱面, 每条磁道平均有 400 个扇区, 而每个扇区有 512B, 其容量为_____GB。
8. 程序所具有的_____特点使得高速缓存能够有效。
9. 若主存地址 32 位, 高速缓存总大小为 2K 行, 块大小 16 字节, 采用 2 路组相连, 则标记位的总位数是_____。
10. 若高速缓存的块大小为 B (B>8) 字节, 向量 v 的元素为 int, 则对 v 的步长为 1 的应用的不命中率为_____。

(三) 简答题

1. [2017B] 结合下面的程序段, 解释局部性。

```
int cal_array_sum(int *a,int n){
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

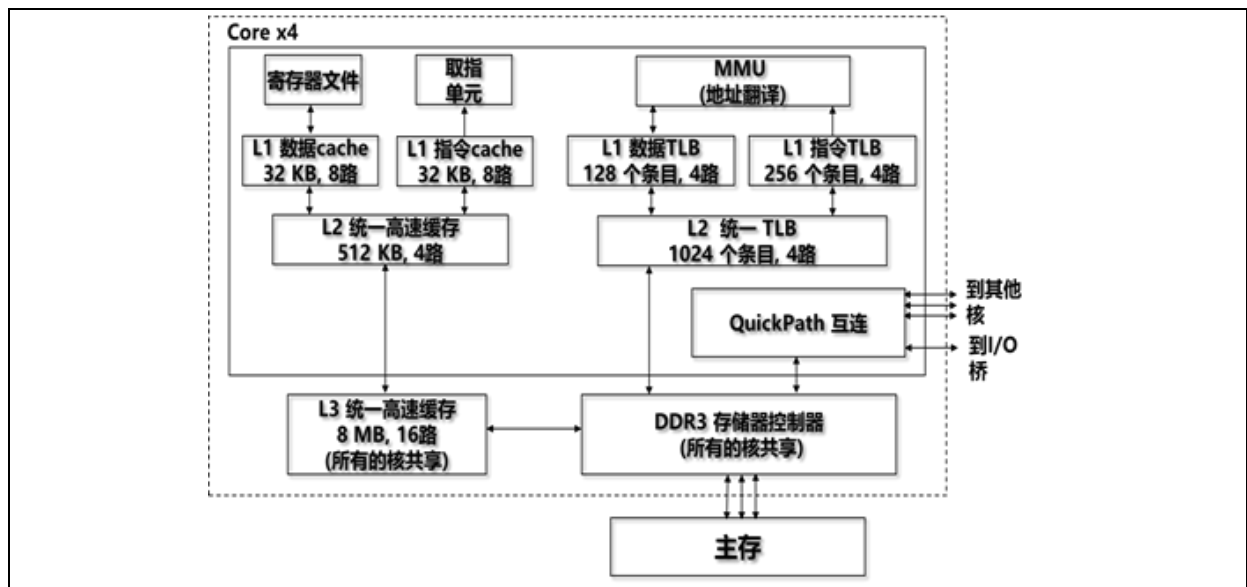

2. 某 CPU 的 L1 cache 容量 32kb, 64B/块, 采用 8 路组相连, 物理地址 47 位。试分析其结构参数 B、S、E 分别是多少? 地址 0x00007f6635201010 访问该 L1 时, 其块偏移 CO、组索引 CI、标记 CT 分别多少?

3. [2018B]某高速缓存大小 256 字节, 直接映射, 块大小为 16 字节。定义 L 为数据装载命令, S 为存储, M 为数据修改。若每一数据装载(L)或存储(S)操作可引发最多 1 次缓存缺失(miss); 数据修改操作(M)可认为是同一地址上 1 次装载后跟 1 次存储, 因此可引发 2 次缓存命中(hit)或 1 次缺失加 1 次命中外加可能的 1 次淘汰/驱逐(evict)。根据下列的访存命令序列, 分析每一命令下的上述高速缓存(高速缓存最初是空的)的命中及淘汰情况。

L 10,1 → M 20,1 → L 22,1 → S 18,1 → L 110,1 → L 210,1 → M 12,1

说明: L 10,1 表示从地址 0x10 处加载 1 个字节数据, 其它同理。

4. [2018B]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_____ (___位); 访问 L1 TLB 的 TLBI 为_____ (___位); 若指令 A 的物理地址为 0x86049b, 则该地址对应的 PPN 为_____ (___位); 访问 L1 cache 的 CT 为_____ (___位), CO 为_____ (___位)。



(四) 综合分析题

1. 假设存在一个能够存储四个数据的 Cache，每一个 line 的长度 (B) 为 2 字节。假设内存空间的地址一共是 32 字节，既内存空间地址长度一共是 5 个比特：从 0 (00000) 到 31 (11111)，一共有 8 个数据读取操作，每个操作的地址按顺序如下所示 (单位是字节)，数据替换采用 LRU (least recently used) 策略。

数据访问地址：1 -> 4 -> 17 -> 2 -> 8 -> 16 -> 9 -> 0

1) 如果 Cache 的结构是直接映射的 ($S=4, E=1$)，如下图所示，请在下图空白处填入，访问上述数据序列访问后 Cache 的状态。(用 [A-B] 表示地址 A 到 B 之间对应的数据)

| 有效位 | 标记位 (二进制) | 数据 |
|-----|-----------|----|
| | | |
| | | |
| | | |
| | | |

2) 如果 cache 的结构如下图所示既 ($S = 2, E = 2$)，请填入访问后的状态

| 有效位 | 标记位 (二进制) | 数据 | 有效位 | 标记位 (二进制) | 数据 |
|-----|-----------|----|-----|-----------|----|
| | | | | | |
| | | | | | |

在这种情况下，数据访问一共产生了多少次不命中 _____

3) 如果 cache 的结构变成 ($S=1, E=4$)，最终存储在 Cache 里面的数据有那些 (注：只需要填写数据部分，顺序不限)？

| | | | |
|--|--|--|--|
| | | | |
|--|--|--|--|

2. Cache

2-1 仔细阅读下面的程序，根据条件回答下列各题

- (1) 地址宽度为 7，数组的起始地址为 0x1000000
- (2) Block Size = 4 Byte, Set = 4, 两路组相连.
- (3) 替换算法位 LRU (最近最少使用)

```
#define LENGTH 8
void clear4x4 ( char array[LENGTH][LENGTH] ) {
    int row, col ;
    for ( col = 0 ; col < 4; col++ ) {
        for ( row = 0; row < 4 ; row++ ) {
            array[ row] [ col] = 0;
        }
    }
}
```

- 1) 以上程序执行会引起多少次失效? _____
- 2) 如果 LENGTH 改为 16, 会引起多少次失效? _____
- 3) 如果 LENGTH 变为 17, 与 2) 相比, 下面描述正确的是: ()。

此时会引起_____次失效。

- A) 16×16 比 17×17 产生更多的失效次数
- B) 16×16 和 17×17 产生的失效次数相同
- C) 16×16 比 17×17 产生更少的失效次数

2-2 改变假设条件, 回答下列各题

- (1) 地址宽度为 8, 数组的起始地址为 0x10000000
- (2) Cache 容量为 16 Byte, Block Size = 4 Byte, 全相联 Cache
- (3) 替换算法位 LRU (最近最少使用)

- 1) Tag 的位数为_____
- 2) 如果执行上述程序, 当 LENGTH=8 时, 会引起多少次失效? _____
- 3) 如果 LENGTH 改为 16, 会引起多少次失效? _____
- 4) 如果 LENGTH 变为 17, 与 3) 相比, 下面描述正确的是: ()
- A. 16×16 比 17×17 产生更多的失效次数
- B. 16×16 和 17×17 产生的失效次数相同
- C. 16×16 比 17×17 产生更少的失效次数

3. 假设我们有一个具有如下属性的系统:

- (a) 内存是字节寻址的
- (b) 内存访问是对 1 字节字的
- (c) 地址宽 13 位。
- (d) 高速缓存是四路组相联的 (E=4)

考虑下面的高速缓存状态: 所有的地址、标记和值都以十六进制表示。每组有 4 行, 索引列包含组索引。标记列包含每一行的标记值。v 列包含每一行的有效位。字节 0~3 列包含每一行的数据, 标号从左向右, 字节 0 在左边。

4 路组相联 cache

| Index | Tag | V | Bytes 0-3 | Tag | V | Bytes 0-3 | Tag | V | Bytes 0-3 | Tag | V | Bytes 0-3 |
|-------|-----|---|-------------|-----|---|-------------|-----|---|-------------|-----|---|-------------|
| 0 | F0 | 1 | ED 32 0A A2 | 8A | 1 | BF 80 1D FC | 14 | 1 | EF 09 86 2A | BC | 0 | 25 44 6F 1A |
| 1 | BC | 0 | 03 3E CD 38 | A0 | 0 | 16 7B ED 5A | BC | 1 | 8E 4C DF 18 | E4 | 1 | FB B7 12 02 |
| 2 | BC | 1 | 54 9E 1E FA | B6 | 1 | DC 81 B2 14 | 00 | 0 | B6 1F 7B 44 | 74 | 0 | 10 F5 B8 2E |
| 3 | BE | 0 | 2F 7E 3D A8 | C0 | 1 | 27 95 A4 74 | C4 | 0 | 07 11 6B D8 | BC | 0 | C7 B7 AF C2 |
| 4 | 7E | 1 | 32 21 1C 2C | 8A | 1 | 22 C2 DC 34 | BC | 1 | BA DD 37 D8 | DC | 0 | E7 A2 39 BA |
| 5 | 98 | 0 | A9 76 2B EE | 54 | 0 | BC 91 D5 92 | 98 | 1 | 80 BA 9B F6 | BC | 1 | 48 16 81 0A |
| 6 | 38 | 0 | 5D 4D F7 DA | BC | 1 | 69 C2 8C 74 | 8A | 1 | A8 CE 7F DA | 38 | 1 | FA 93 EB 48 |
| 7 | 8A | 1 | 04 2A 32 6A | 9E | 0 | B1 86 56 0E | CC | 1 | 96 30 47 F2 | BC | 1 | F8 1D 42 30 |

1) 这个高速缓存的 S, E, B, C 是多少?

2) 引用位于地址 0x071A 处的 1 字节字。用十六进制表示出它所访问的高速缓存条目, 以及返回的高速缓存字节值。指明是否发生了高速缓存不命中。如果有高速缓存不命中, 对于“返回的高速缓存字节”输入“—”。提示: 注意那些有效位!

A. 地址格式

| | | | | | | | | | | | | |
|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | | | | | | | | |

B. 内存引用

| 参数 | 值 |
|---------------|---|
| 高速缓存块偏移 (CO) | |
| 高速缓存组索引 (CI) | |
| 高速缓存标记 (CT) | |
| 高速缓存命中? (是/否) | |
| 返回的高速缓存字节 | |

3) 引用位于地址 0x16E8 处的 1 字节字。

A. 地址格式

| | | | | | | | | | | | | |
|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | | | | | | | | |

B. 内存引用

| 参数 | 值 |
|---------------|---|
| 高速缓存块偏移 (CO) | |
| 高速缓存组索引 (CI) | |
| 高速缓存标记 (CT) | |
| 高速缓存命中? (是/否) | |
| 返回的高速缓存字节 | |

4) 对于上面高速缓存, 列出会在组 2 中命中的 8 个内存地址 (以十六进制表示)。

4. 假设存在一个能够存储四个 Block 的 Cache, 每一个 Block 的长度为 2Byte。假设内存空间大小共是 16Byte, 即内存空间地址长度一共是 4bit, 可访问地址为 (0~15), 数据访问地址序列

如下所示，访问数据单位是 Byte，默认替换策略是 LRU。

2 3 10 9 6 8

1) 如果 Cache 的结构如下图所示($S=2, E=2$), 请在下图空白处填入访问上述六次数据访问后 Cache 的状态。注: 用[0-1]表示地址 0 至 1 上对应的数据

| | 有效位 | 标识 (二进制) | 块 | 有效位 | 标识 (二进制) | 块 |
|------|-----|----------|---|-----|----------|---|
| set0 | | | | | | |
| set1 | | | | | | |

2) 这六次数据访问一共产生了多少次不命中 4。

3) 如果 Cache 的替换策略改成 MRU (即, 最近使用的数据被替换出去), 请在下图空白处填入访问上述六次数据访问后 Cache 的状态。

| | 有效位 | 标识 (二进制) | 块 | 有效位 | 标识 (二进制) | 块 |
|------|-----|----------|---|-----|----------|---|
| set0 | | | | | | |
| set1 | | | | | | |

4) 这六次数据访问一共产生了多少次不命中 _____

5. 考虑下面的矩阵转置函数:

假设右边的代码运行在一台具有如下属性的机器上:

- (a) `sizeof(int) = 4`
- (b) `src` 从地址 0 开始, 而数组 `dst` 从地址 64 开始 (十进制)。
- (c) 数组只有一个 L1 数据高速缓存, 它是直接映射、直写、写分配的, 块大小为 16 B。
- (d) 这个高速缓存总共有 32 个数据字节, 初始为空。
- (e) 对 `src` 和 `dst` 数组的访问分别是读和写不命中中的唯一来源。

```
typedef int array[4][4];
void transpose2(array dst, array src){
    int i, j;
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            dst[j][i] = src[i][j];
        }
    }
}
```

对于每个 row 和 col 指明对 `src[row][col]` 和 `dst[row][col]` 的访问是命中(h)还是不命中(m)。

1) 例如, 读 `src[0][0]` 会不命中, 而写 `dst[0][0]` 也会不命中

| | dst | | | |
|-----|-----|---|---|---|
| 行/列 | 0 | 1 | 2 | 3 |
| 0 | m | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

| | src | | | |
|-----|-----|---|---|---|
| 行/列 | 0 | 1 | 2 | 3 |
| 0 | m | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

2) 如果对于总大小为 128 数据字节的高速缓存, 情况是怎么样的?

| dst | | | | |
|-----|---|---|---|---|
| 行/列 | 0 | 1 | 2 | 3 |
| 0 | m | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

| src | | | | |
|-----|---|---|---|---|
| 行/列 | 0 | 1 | 2 | 3 |
| 0 | m | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

第七章、 链接

一、 真题考点

◆ 考点一：编译器驱动程序概述

(1) 静态链接（编译时链接）

符号解析和重定位

(2) 加载时链接

(3) 运行时链接

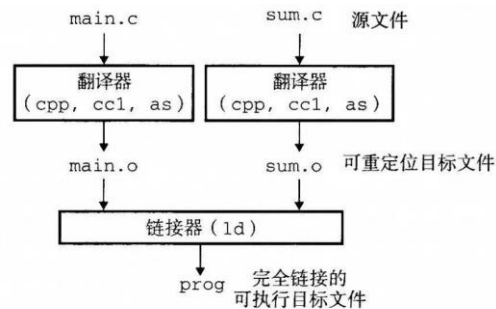


图 7-2 静态链接。链接器将可重定位目标文件组合起来，形成一个可执行目标文件 prog

◆ 考点二：目标文件

可重定位目标文件（.o 文件）、可执行目标文件（a.out）、共享目标文件（.so）

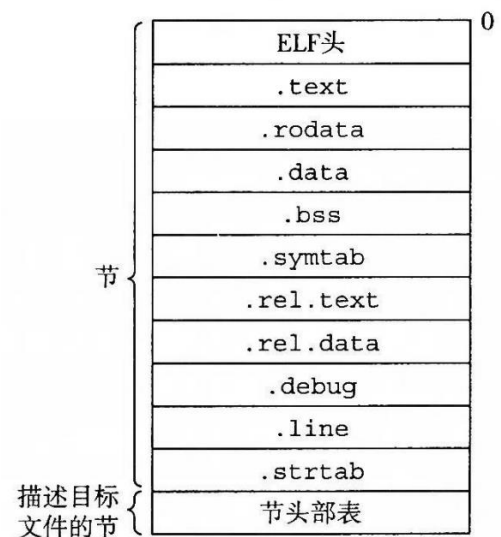
可重定位目标文件

.rodata 节（只读数据）：如跳转表和 printf 中的格式串

.data 节（数据/可读写）：已初始化全局变量，局部 C 变量在运行时被保存在栈中，既不出现在 .data 节中，也不出现在 .bss 节中。

.bss 节（未初始化全局变量）：未初始化的全局变量

.symtab 节（符号表）：.symtab 符号表不包含局部变量的条目。



可执行目标文件

可重定位目标文件没有段头部表和 .init 节

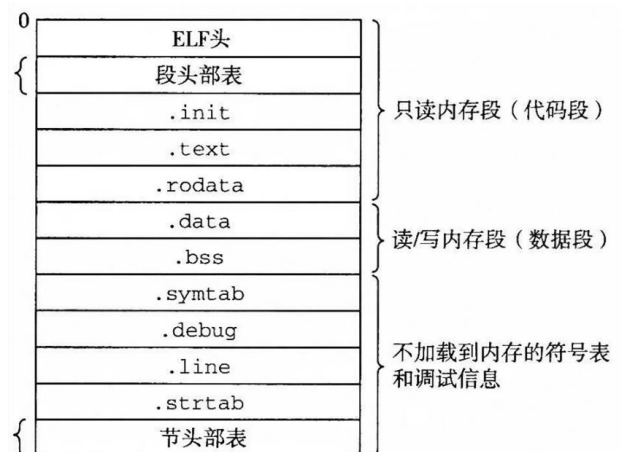


图 7-13 典型的 ELF 可执行目标文件

◆ 考点三：符号解析

1. 符号表

(1) 全局符号

由模块 *m* 定义的，可以被其他模块引用的符号，例如：非静态 `non-static C` 函数与非静态全局变量。

(2) 外部符号

由模块 *m* 引用的全局符号，但由其他模块定义。

(3) 本地/局部符号

由模块 *m* 定义和仅由 *m* 唯一引用的符号，如：使用静态属性定义的 `C` 函数和全局变量

注意：本地链接符号不是本地程序变量-局部变量。本地静态 `C` 变量存储在 `.bss` 或 `.data`，本地非静态 `C` 变量存储在栈上

2. 符号解析：将目标文件中的每个全局符号都绑定到一个唯一的定义

(1) 全局符号

强符号：函数和初始化全局变量；**弱符号：**未初始化的全局变量

规则 1：不允许多个同名的强符号，每个强符号只能定义一次；否则，链接器错误

规则 2：若有一个强符号和多个弱符号同名，则选择强符号；对弱符号的引用解析为强符号

规则 3：如果有多个弱符号，选择任意一个

(2) 静态库及其优点

编译时将所有相关的目标模块打包成为一个单独的文件用作链接器的输入，这个文件称为静态库。

使用时，通过把相关函数编译为独立模块，然后封装成一个单独的静态库文件，应用程序可以通过在命令行上指定单独的文件名来使用这些在库中定义的函数，链接器只需复制被程序引用的目标模块，减少了可执行文件在磁盘和内存中的大小；同时，应用程序员只需包含较少的库文件的名字。

(3) 链接器使用静态库来解析引用

关于库的一般准则是将它们**放在命令行的结尾**。如果各个库的成员是相互独立的，那么这些库就可以以任何顺序放置在命令行的结尾处。另一方面，如果库不是相互独立的，那么必须对它们排序。

◆ 考点四：重定位

确定每个符号的最终内存地址，并修改对那些目标的引用

重定位符号引用：1.重定位 `PC` 相对引用；2.重定位绝对引用（具体细节见课本）

◆ 考点五：动态链接共享库

1. 共享库（动态链接库）

共享库（动态链接库）是一个 `.so` 的目标模块（`elf` 文件），在运行或加载时，由动态链接器程序加载到任意的内存地址，并和一个和内存中的程序（如当前可执行目标文件）动态完全连接为一个可执行程序。使用它可节省内存与硬盘空间，方便软件的更新升级。如标准 `C` 库 `libc.so`。

2. 动态链接的实现方法

加载时动态链接：应用程序第一次加载和运行时，通过 `ld-linux.so` 动态链接器重定位动态库的代码和数据到某个内存段，再重定位当前应用程序中对共享库定义的符号的引用，然后将控制传递给应用程序（此后共享库位置固定了并不变）。

运行时动态链接：在程序执行过程中，通过 `dlopen/dlsym` 函数加载和连接共享库，实现符号重定位，通过 `dlclose` 卸载动态库。

◆ **考点六：高级主题——位置无关代码、库打桩**

1. 可以加载而无需重定位的代码称为**位置无关代码（PIC）**

2. **库打桩：**允许截获对共享库函数的调用，取而代之执行自己的代码，

编译时：源代码被编译时

链接时：当可重定位目标文件被静态链接来形成一个可执行目标文件时

加载/运行时：当一个可执行目标文件被加载到内存中，动态链接，然后执行时

二、 真题演练

(一) 选择题

1. 在链接时,对于什么样的符号一定不需要进行重定位? ()

- A. 不同 C 语言源文件中定义的函数
B. 同一 C 语言源文件中定义的全局变量
C. 同一函数中定义时不带 static 的变量
D. 同一函数中定义时带有 static 的变量

2. 下列程序运行的结果是什么? ()

```
/* main.c */
int i=0;
int main(){
    foo();
    return 0;
}

/* foo.c */
int i=1;
void foo(){
    printf("%d", i);
}
```

- A. 编译错误
B. 链接错误
C. 段错误
D. 有时打印输出 1, 有时打印输出 0

3. 下列关于静态库链接的描述中,错误的是 ()

- A. 链接时,链接器只拷贝静态库中被程序引用的目标模块
B. 使用库的一般准则是将它们放在命令行的结尾
C. 如果库不是相互独立的,那么它们必须排序
D. 每个库在命令行只须出现一次即可

4. 以下关于静态库链接的描述中,正确的是 ()

- A. 链接时,链接器会拷贝静态库中的所有目标模块。
B. 使用库的时候必须把它们放在命令行的结尾处。
C. 如果库不是相互独立的,那么它们必须排序。
D. 每个库在命令行只须出现一次即可。

5. 在 foo.c 文件中包含如下代码:

```
int foo(void) {
    int error = printf("You ran into a problem!\n");
    return error;
}
```

经过编译和链接之后,字符串 "You ran into a problem!\n" 会出现在哪个段中? ()

- A. .bss
B. .data
C. .rodata
D. .text

6. 链接时两个文件同名的弱符号,以 () 为基准

- A. 链接时先出现的
B. 链接时后出现的
C. 任一个
D. 链接报错

7. 链接过程中,赋初值的局部变量名,正确的是 ()

- A. 强符号
B. 弱符号
C. 若是静态的则为强符号
D. 以上都错

8. C 语句中的全局变量,在 () 阶段被定位到一个确定的内存地址

- A. 编译
B. 链接
C. 执行
D. 调试

9. 链接时两个同名的强符号, 以哪种方式处理? ()
- A. 链接时先出现的符号为准 B. 链接时后出现的符号为准
- C. 任一个符号为准 D. 链接报错
10. 链接过程中, 带 `static` 属性的全局变量属于 ()
- A. 全局符号 B. 局部符号 C. 外部符号 D. 以上都错
11. 以下关于程序中链接“符号”的陈述, 错误的是 ()
- A. 赋初值的非静态全局变量是全局强符号 B. 赋初值的静态全局变量是全局强符号
- C. 未赋初值的非静态全局变量是全局弱符号 D. 未赋初值的静态全局变量是本地符号
12. 关于动态库的描述错误的是 ()
- A. 可在加载时链接, 即当可执行文件首次加载和运行时进行动态链接。
- B. 更新动态库, 即便接口不变, 也需要将使用该库的程序重新编译。
- C. 可在运行时链接, 即在程序开始运行后通过程序指令进行动态链接。
- D. 即便有多个正在运行的程序使用同一动态库, 系统也仅在内存中载入一份动态库。
13. 关于局部变量, 正确的叙述是 ()
- A. 普通 (`auto`) 局部变量也是一种编程操作的数据, 存放在数据段
- B. 非静态局部变量在链接时是本地符号
- C. 静态局部变量是全局符号
- D. 编译器可将 `%rsp` 减取一个数为局部变量分配空间
14. Linux 系统中将可执行目标文件 (`.out` 文件) 装入到存储空间时, 没有装入到 `.text` 段-只读代码段的是 ()
- A. ELF 头 B. `.init` 节 C. `.rodata` 节 D. `.symtab` 节
15. 链接过程中, 赋初值的静态全局变量属于 ()
- A. 强符号 B. 弱符号 C. 可能是强符号也可能是弱符号 D. 以上都不是
16. 下面说法错误的是 ()
- A. 一个 C 程序中的跳转表数据经链接后被映射到代码段
- B. 共享目标文件 (共享库) 是在运行时由动态链接器链接和加载的, 或者隐含地在调用程序被加载和开始执行时, 或者根据需要在程序调用 `dlopen` 库的函数时
- C. 链接器用库 (假设库相互独立) 来解析其他目标模块中的符号引用, 并以任意的顺序解析符号引用
- D. 链接时, 若一个强符号和多个弱符号同名, 则对弱符号的引用均将被解析成强符号。
17. 链接过程中, 赋初值的非静态全局变量名, 属于 ()
- A. 强符号 B. 弱符号 C. 强符号或弱符号 D. 以上都错
18. 共享库 (动态链接库) 在程序的 () 阶段由动态链接器加载到任意的内存地址。
- A. 编译 B. 链接 C. 运行 D. 运行或链接

(二) 填空题

1. 链接器经过_____和重定位两个阶段, 将可重定位目标文件生成可执行目标文件。
2. 填写下面空缺的部分: `400534: e8 _____ callq 4004e7 <sum>`

(R_X86_64_PC32 sum-0x4)

3. 若 `p.o->libx.a->liby.a` 且 `liby.a->libx.a->p.o` 则最小链接命令行_____。

4. 可重定位目标文件中代码地址从_____开始。

5. 链接有_____、_____、_____三种方式。

6. 可以加载而无需重定位的代码称为_____。

7. C 语言程序运行时，局部变量存在放_____段。

8. C 语句中的全局变量，在_____阶段被定位到一个确定的内存地址。

(三) 简答题

1. 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。

| | 存储区域 | 赋初值 | 生命周期 | 指令中寻址方式 | 是否需要重定位 |
|---------------------------|------|-----|------|---------|---------|
| <code>int</code> 全局变量 | | | | | |
| 非静态 <code>int</code> 局部变量 | | | | | |

2. 什么是共享库（动态连接库）？简述动态链接的实现方法。

3. 什么是静态库？使用静态库的优点是什么？

4. `a` 和 `b` 表示当前路径中的目标模块或静态库，而 `a->b` 表示 `a` 依赖于 `b`，也就是说 `a` 引用了一个 `b` 定义的符号。对于下面的每个场景，给出使得静态链接器能够解析所有符号引用的最小的命令行（即含有最少数量的目标文件和库参数的命令）。

A. `p.o → libx.a → p.o`

B. `p.o → libx.a → liby.a` and `liby.a → libx.a`

C. `p.o → libx.a → liby.a → libz.a` and `liby.a → libx.a → libz.a`

5. 考虑目标文件中对 m.o 中对 swap 函数的调用

```
9: e8 00 00 00 00          callq e <main+0xe>      //swap()
```

它的重定位条目如下：

```
r.offset = 0xa
r.symbol = swap
r.type = R_X86_64_PC32
r.addend = -4
```

现在假设链接器将 m.o 中的 .text 重定位到地址 0x4004d0, 将 swap 重定位到地址 0x4004e8。那么 callq 指令中对 swap 的重定位引用的值是什么？

(四) 综合分析题

1. 考虑下面的程序，它由两个模块组成：

| | |
|--|--|
| <pre>/*main*/ #include <stdio.h> int x = 100; int y; void p1(void); int main() { int z=0; p1(); y = 2000; printf("x=%d,y=%d\n", x, y); return 0; }</pre> | <pre>/*p1.c*/ int x; int y; void p1() { x=1000; y=200; }</pre> |
|--|--|

请指出 main.o 中属于强符号的是？_____

程序最后的输出是什么？_____

2. 观察下面的 m.o 模块和 swap.c 函数

| | |
|--|--|
| <pre>void swap(); int buf[2] = {1, 2}; int main(){ swap(); return 0; }</pre> | <pre>extern int buf[]; int *bufp0 = &buf[0]; static int *bufp1; static void incr(){ static int count=0; count++; } void swap(){ int temp; incr(); bufp1 = &buf[1]; temp = *bufp0; *bufp0 = *bufp1;</pre> |
|--|--|

| | |
|--|-----------------------------|
| | <pre>*bufp1 = temp; }</pre> |
|--|-----------------------------|

对于每个 `swap.o` 中定义和引用的符号，请指出它是否在模块 `swap.o` 的 `.symtab` 节中有符号表条目。如果是这样，请指出定义该符号的模块（`swap.o` 或 `m.o`）、符号类型（局部、全局或外部）以及它在模块中所处的节（`.text`、`.data` 或 `.bss`）。

| 符号 | <code>swap.o.symtab</code> 条目? | 符号类型 | 定义符号的模块 | 节 |
|--------------------|--------------------------------|------|---------|---|
| <code>buf</code> | | | | |
| <code>bufp0</code> | | | | |
| <code>bufp1</code> | | | | |
| <code>swap</code> | | | | |
| <code>temp</code> | | | | |
| <code>incr</code> | | | | |
| <code>count</code> | | | | |

3-4 两个C语言程序 `main.c`、`test.c` 如下所示:

```
/* main.c */
#include <stdio.h>
int a[4]={-1,-2,2, 3};
extern int val;
int sum();
int main(int argc, char * argv[ ] ){
    val=sum();
    printf("sum=%d\n",val);
}

/* test.c */
extern int a[ ];
int val=0;
int sum( ){
    int i;
    for (i=0; i<4; i++)
        val += a[i];
    return val;
}
```

用如下两条指令编译、链接，生成可执行程序 `test`:

```
gcc -m64 -no-pie -fno-PIC -c test.c main.c
gcc -m64 -no-pie -fno-PIC -o test test.o main.o
```

运行指令 `objdump -dxs main.o` 输出的部分内容如下:

```
Contents of section .data:
0000 ffffffff feffffff 02000000 03000000 .....
Contents of section .rodata:
0000 73756d3d 25640a00          sum=%d..
...
```

Disassembly of section `.text`:

```
0000000000000000 <main>:
  0: 55                push    %rbp
  1: 48 89 e5          mov     %rsp,%rbp
  4: 48 83 ec 10       sub     $0x10,%rsp
  8: 89 7d fc          mov     %edi,-0x4(%rbp)
 b: 48 89 75 f0       mov     %rsi,-0x10(%rbp)
 f: b8 00 00 00 00    mov     $0x0,%eax
14: e8 00 00 00 00    callq   19 <main+0x19>
    15: R X86 64 PC32    sum-0x4
19: 89 05 00 00 00 00    mov     %eax,0x0(%rip) # 1f <main+0x1f>
```

```

1b: R X86 64 PC32 val-0x4
1f: 8b 05 00 00 00 00 mov 0x0(%rip),%eax # 25 <main+0x25>
21: R X86 64 PC32 val-0x4
25: 89 c6 mov %eax,%esi
27: bf 00 00 00 00 mov $0x0,%edi
28: R X86 64 32 .rodata
2c: b8 00 00 00 00 mov $0x0,%eax
31: e8 00 00 00 00 callq 36 <main+0x36>
32: R X86 64 PC32 printf-0x4
36: b8 00 00 00 00 mov $0x0,%eax
3b: c9 leaveq
3c: c3 retq
    
```

objdump -dxs test 输出的部分内容如下 (■是没有显示的隐藏内容):

SYMBOL TABLE:

```

0000000000400400 l d .text 0000000000000000 .text
00000000004005e0 l d .rodata 0000000000000000 .rodata
0000000000601020 l d .data 0000000000000000 .data
0000000000601040 l d .bss 0000000000000000 .bss
0000000000000000 F *UND* 0000000000000000 printf@@GLIBC_2.2.5
0000000000601044 g O .bss 0000000000000004 val
0000000000601030 g O .data 0000000000000010 a
00000000004004e7 g F .text 0000000000000039 sum
0000000000400400 g F .text 000000000000002b _start
0000000000400520 g F .text 000000000000003d main
    
```

Contents of section .rodata:

```
4005e0 01000200 73756d3d 25640a00 ....sum=%d..
```

...

Contents of section .data:

```

601020 00000000 00000000 00000000 00000000 .....
601030 ffffffff feffffff 02000000 03000000 .....
    
```

...

00000000004003f0 <printf@plt>:

```
4003f0: ff 25 22 0c 20 00 jmpq *0x200c22(%rip) # 601018
```

<printf@GLIBC_2.2.5>

```
4003f6: 68 00 00 00 00 pushq $0x0
```

```
4003fb: e9 e0 ff ff jmpq 4003e0 <.plt>
```

Disassembly of section .text:

0000000000400400 <_start>:

```
400400: 31 ed xor %ebp,%ebp
```

....

00000000004004e7 <sum>:

```
4004e7: 55 push %rbp #①
```

```
4004e8: 48 89 e5 mov %rsp,%rbp #②
```

```
4004eb: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp) #③
```

```
4004f2: eb 1e jmp 400512 <sum+0x2b>
```

```
4004f4: 8b 45 fc mov -0x4(%rbp),%eax
```

```
4004f7: 48 98 cltq
```

```
4004f9: 8b 14 85 30 10 60 00 mov 0x601030(,%rax,4),%edx
```

```
400500: 8b 05 3e 0b 20 00 mov 0x200b3e(%rip),%eax #601044 <val>
```

```
400506: 01 d0 add %edx,%eax
```

```
400508: 89 05 36 0b 20 00 mov %eax,0x200b36(%rip) #601044 <val>
```

```
40050e: 83 45 fc 01 addl $0x1,-0x4(%rbp)
```

```
400512: 83 7d fc 03 cmpl $0x3,-0x4(%rbp) #④
```

```
400516: 7e dc jle 4004f4 <sum+0xd> #⑤
```

```
400518: 8b 05 26 0b 20 00 mov 0x200b26(%rip),%eax # 601044 <val>
```

```
40051e: 5d pop %rbp
```

```
40051f: c3 retq
```

0000000000400520 <main>:

```

400520: 55                push  %rbp
400521: 48 89 e5          mov   %rsp,%rbp
400524: 48 83 ec 10       sub   $0x10,%rsp
400528: 89 7d fc          mov   %edi,-0x4(%rbp)
40052b: 48 89 75 f0       mov   %rsi,-0x10(%rbp)
40052f: b8 00 00 00 00    mov   $0x0,%eax
400534: e8( ① )          callq 4004e7 <sum>
400539: 89 05( ② )       mov   %eax,██████(%rip) #601044<val>
40053f: 8b 05( ③ )       mov   ██████(%rip),%eax #601044<val>
400545: 89 c6            mov   %eax,%esi
400547: bf ( ④ )         mov   ██████,%edi
40054c: b8 00 00 00 00    mov   $0x0,%eax
400551: e8 ( ⑤ )         callq 4003f0 <printf@plt>
400556: b8 00 00 00 00    mov   $0x0,%eax
40055b: c9              leaveq
40055c: c3              retq
40055d: 0f 1f 00         nopl  (%rax)
    
```

3. 阅读的sum函数反汇编结果中带下划线的汇编代码(编号①-⑤),解释每行指令的功能和作用

```

4004e7: 55                push  %rbp                #①
4004e8: 48 89 e5          mov   %rsp,%rbp          #②
4004eb: c7 45 fc 00 00 00 00 movl  $0x0,-0x4(%rbp)      #③
400512: 83 7d fc 03       cmpl  $0x3,-0x4(%rbp)     #④
400516: 7e dc            jle   4004f4 <sum+0xd>    #⑤
    
```

4. 根据上述信息,链接程序从目标文文件test.o和main.o生成可执行程序test,对main函数中空格①-⑤所在语句所引用符号的重定位结果是什么?以16进制4字节数值填写这些空格,将机器指令补充完整(写出任意2个即可)。

0

```

400534: e8( ① )          callq 4004e7 <sum>
400539: 89 05( ② )       mov   %eax,██████(%rip) #601044<val>
40053f: 8b 05( ③ )       mov   ██████(%rip),%eax #601044<val>
400545: 89 c6            mov   %eax,%esi
400547: bf ( ④ )         mov   ██████,%edi
40054c: b8 00 00 00 00    mov   $0x0,%eax
400551: e8 ( ⑤ )         callq 4003f0 <printf@plt>
400556: b8 00 00 00 00    mov   $0x0,%eax
    
```


5. 在sum函数地址4004f9处的语句"mov 0x601030(,%rax,4),%edx"中,源操作数是什么类型、有效地址如何计算、对应C语言源程序中的什么量(或表达式)?其中,rax数值对应C语言源程序中的哪个量(或表达式)?如何解释数字4?

6-7 两个C语言程序main2.c、addvec.c如下所示:

| | |
|---|---|
| <pre> /* 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("z = [%d %d]\n", z[0], z[1]); return 0; } /* \$end main2 */ </pre> | <pre> /* 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 */ </pre> |
|---|---|

用如下两条指令编译、链接,生成可执行程序prog2:

```
gcc -m64 -no-pie -fno-PIC -c addvec.c main2.c
```

```
gcc -m64 -no-pie -fno-PIC -o prog2 addvec.o main2.o
```

运行指令objdump -dxs main2.o输出的部分内容如下:

Disassembly of section .text:

0000000000000000 <main>:

```

0: 48 83 ec 08          sub    $0x8,%rsp
4: b9 02 00 00 00      mov    $0x2,%ecx
9: ba 00 00 00 00      mov    $0x0,%edx
   a: R_X86_64_32    z
e: be 00 00 00 00      mov    $0x0,%esi
   f: R_X86_64_32    y
13: bf 00 00 00 00      mov    $0x0,%edi
   14: R_X86_64_32    x
18: e8 00 00 00 00      callq 1d <main+0x1d>
   19: R_X86_64_PC32    addvec-0x4
1d: 8b 0d 00 00 00 00      mov    0x0(%rip),%ecx    # 23 <main+0x23>
   1f: R_X86_64_PC32    z
23: 8b 15 00 00 00 00      mov    0x0(%rip),%edx    # 29 <main+0x29>
   25: R_X86_64_PC32    z-0x4
29: be 00 00 00 00      mov    $0x0,%esi
   2a: R_X86_64_32    .rodata.str1.1
                
```

```

2e: bf 01 00 00 00      mov     $0x1,%edi
33: b8 00 00 00 00      mov     $0x0,%eax
38: e8 00 00 00 00      callq  3d <main+0x3d>
    39: R_X86_64_PC32      __printf_chk-0x4
3d: b8 00 00 00 00      mov     $0x0,%eax
42: 48 83 c4 08         add     $0x8,%rsp
46: c3                  retq
    
```

objdump -dxs prog2 输出的部分内容如下 (■是没有显示的隐藏内容):

SYMBOL TABLE:

```

0000000000400238 l    d  .interp 0000000000000000          .interp
0000000000400254 l    d  .note.ABI-tag
0000000000000000 l    df *ABS* 0000000000000000          main2.c
0000000000601038 g        *ABS* 0000000000000000          _edata
000000000060103c g    O  .bss 0000000000000000          z
0000000000601030 g    O  .data 0000000000000000          x
0000000000000000 F  *UND* 0000000000000000          addvec
0000000000601018 g        .data 0000000000000000          __data_start
00000000004007e0 g    O  .rodata 0000000000000004
_IO_stdin_used
0000000000601028 g    O  .data 0000000000000000          y
00000000004006f0 g    F  .text 0000000000000047          main
    
```

00000000004005c0 <addvec@plt>:

```

4005c0: ff 25 42 0a 20 00      jmpq    *0x200a42(%rip)    # 601008
    
```

<_GLOBAL_OFFSET_TABLE_+0x20>

```

4005c6: 68 01 00 00 00      pushq   $0x1
    
```

```

4005cb: e9 d0 ff ff ff      jmpq    4005a0 <_init+0x18>
    
```

00000000004005d0 <__printf_chk@plt>:

```

4005d0: ff 25 3a 0a 20 00      jmpq    *0x200a3a(%rip)    # 601010
    
```

<_GLOBAL_OFFSET_TABLE_+0x28>

....

00000000004006f0 <main>:

```

4006f0: 48 83 ec 08      sub     $0x8,%rsp
4006f4: b9 02 00 00 00      mov     $0x2,%ecx
4006f9: ba ① _ _ _ _      mov     ■■■■,%edx
4006fe: be ② _ _ _ _      mov     ■■■■,%esi
400703: bf ③ _ _ _ _      mov     ■■■■,%edi
400708: e8 ④ _ _ _ _      callq   4005c0 <addvec@plt>
40070d: 8b 0d ⑤ _ _ _ _      mov     ■■■■(%rip),%ecx # 601040 <z+0x4>
400713: 8b 15 ⑥ _ _ _ _      mov     ■■■■(%rip),%edx # 60103c <z>
400719: be e4 07 40 00      mov     $0x4007e4,%esi
40071e: bf 01 00 00 00      mov     $0x1,%edi
400723: b8 00 00 00 00      mov     $0x0,%eax
400728: e8 ⑦ _ _ _ _      callq   4005d0 <__printf_chk@plt>
40072d: b8 00 00 00 00      mov     $0x0,%eax
400732: 48 83 c4 08      add     $0x8,%rsp
400736: c3              retq
    
```

6. 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?

7. 根据上述信息, main函数中空格①--⑦所在语句所引用符号的重定位结果是什么? 以16进制4字节数值填写这些空格, 将机器指令补充完整(写出任意3个即可)。(5分)

① _____

② _____

③ _____

④ _____

⑤ _____

⑥ _____

⑦ _____

第八章、 异常控制流

一. 真题考点

◆ 考点一：异常（硬件层 EOF）

硬件检测到的事件会触发控制转移到异常处理程序，操作系统和硬件共同实现

1. 异常的种类

异常可以分为四类：中断（interrupt）、陷阱（trap）、故障（fault）和终止（abort）。

中断： 例如时钟中断、I/O 中断（ctrl-c）

陷阱： 系统调用

故障： 缺页、保护故障、浮点异常

终止： 非法指令、奇偶校验错误、机器检查

| 类别 | 原因 | 异步 / 同步 | 返回行为 |
|----|--------------|---------|------------|
| 中断 | 来自 I/O 设备的信号 | 异步 | 总是返回到下一条指令 |
| 陷阱 | 有意的异常 | 同步 | 总是返回到下一条指令 |
| 故障 | 潜在可恢复的错误 | 同步 | 可能返回到当前指令 |
| 终止 | 不可恢复的错误 | 同步 | 不会返回 |

图 8-4 异常的种类。异步异常是由处理器外部的 I/O 设备中的事件产生的。同步异常是执行一条指令的直接产物

2. Linux/x86-64 故障和终止

除法错误。当应用试图除以零时，或者当一个除法指令的结果对于目标操作数来说太大了的时候，就会发生除法错误（异常 0）。

一般保护故障。通常是因为一个程序引用了一个未定义的虚拟内存区域，或者因为程序试图写一个只读的文本段。Linux shell 通常会把这种一般保护故障报告为**段故障**

缺页。是会重新执行产生故障的指令的一个异常示例。

机器检查。机器检查是在导致故障的指令执行中检测到致命的硬件错误时发生的

| 异常号 | 描述 | 异常类别 |
|--------|-----------|-------|
| 0 | 除法错误 | 故障 |
| 13 | 一般保护故障 | 故障 |
| 14 | 缺页 | 故障 |
| 18 | 机器检查 | 终止 |
| 32~255 | 操作系统定义的异常 | 中断或陷阱 |

注：异常表（中断向量表）是一张跳转表，表目 k 包含异常 k 的处理程序代码的地址

◆ 考点二：进程（操作系统级 EOF）

1. 并发：一个逻辑流的执行在时间上与另一个流重叠，称为并发流

2. 上下文切换

（1）概念

内核为每个进程维持一个上下文（context）。上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值组成，这些对象包括**通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构**，比如描述地址空间的**页表**、包含有关当前进程信息的**进程表**，以及包含进程已打开文件的信息的**文件表**。

(2) 上下文切换过程

保存当前进程的上下文，恢复某个先前被抢占的进程被保存的上下文，将控制传递给这个新恢复的进程。

(3) 发生上下文切换的情况

(a) 内核代表用户执行系统调用

(b) 中断也可能引发上下文切换。比如所有的系统都有某种产生周期性定时器中断的机制

(c) 当前进程时间片用尽

3. 进程控制

(1) 进程状态：运行，停止或者终止

(2) **fork()** 函数：调用一次，却会返回两次：一次是在调用进程（父进程）中，一次是在新建的子进程中。在父进程中，**fork** 返回子进程的 **PID**，在子进程中，**fork** 返回 **0**。**fork()** 函数可以创建子进程，父进程和子进程的关系为：

并发执行：父进程和子进程是并发运行的独立进程。内核能够以任意方式交替执行它们的逻辑控制流中的指令。

相同但是独立的地址空间：每个进程有相同的用户栈、相同的本地变量值、相同的堆、相同的全局变量值，以及相同的代码。

共享文件：子进程继承了父进程所有的打开文件

(3) 回收子进程：一个进程可以通过调用 **waitpid** 函数来等待它的子进程终止或者停止。

(4) 加载并运行程序：**execve** 函数在当前进程的上下文中加载并运行一个新程序，**execve** 调用一次并从不返回。

◆ 考点三：信号（应用层级 EOF）

Linux 信号是软件形式的异常，它允许进程和内核中断其他进程。

1. 发送信号

从键盘发送信号：在键盘上输入 **Ctrl+C** 会导致内核发送一个 **SIGINT** 信号到前台进程组中的每个进程。默认情况下，结果是终止前台作业。类似地，输入 **Ctrl+Z** 会发送一个 **SIGTSTP** 信号到前台进程组中的每个进程。默认情况下，结果是停止（挂起）前台作业。

用 kill 函数发送信号：进程通过调用 **kill** 函数发送信号给其他进程（包括它们自己）。

（注：信号中还需要记忆的是 **SIGKILL** 和 **SIGSTOP** 既不能被捕获，也不能被忽略；**SIGKILL** 是杀死子进程；子进程运行结束后会向父进程发生 **SIGCHLD** 信号）

2. 接收信号

每个信号类型都有一个预定义的默认行为，是下面中的一种：

(1) **进程终止**。导致进程终止的原因有：1) 收到一个信号，2) 从主程序返回，3) 执行 **exit** 函数

(2) 进程终止并转储内存。

(3) 进程停止（挂起）直到被 **SIGCONT** 信号重启。

(4) 进程忽略该信号。

3. Linux 处理信号以及编写信号处理程序

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。

信号处理程序的编写原则：

- 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**：如 write，而 printf、sprintf、malloc 和 exit) 都不在此列
- 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

◆ 考点四：非本地跳转

C 语言提供了一种用户级异常控制流形式，称为非本地跳转（nonlocal jump），它将控制直接从一个函数转移到另一个当前正在执行的函数，而不需要经过正常的调用-返回序列。非本地跳转是通过 setjmp 和 longjmp 函数来提供的。

setjmp 函数只被调用一次，但返回多次；

longjmp 函数被调用一次，但从不返回

◆ 考点五：shell

1. shell 的概念

Linux 系统中，Shell 是一个交互型应用级程序，代表用户运行其他程序（是命令行解释器，以用户态方式运行的终端进程）。其基本功能是解释并运行用户的指令，重复如下处理过程：

- (1) 终端进程读取用户由键盘输入的命令行。
- (2) 分析命令行字符串，获取命令行参数，并构造传递给 execve 的 argv 向量
- (3) 检查第一个（首个、第 0 个）命令行参数是否是一个内置的 shell 命令
- (3) 如果不是内部命令，调用 fork() 创建新进程/子进程
- (4) 在子进程中，用步骤 2 获取的参数，调用 execve() 执行指定程序。
- (5) 如果用户没要求后台运行（命令末尾没有 & 号）否则 shell 使用 waitpid（或 wait...）等待作业终止后返回。
- (6) 如果用户要求后台运行（如果命令末尾有 & 号），则 shell

2. 结合 fork() 和 execve()，shell 中加载和运行 hello 程序的过程

①在 shell 命令行中输入命令：\$./hello

②shell 命令行解释器构造 argv 和 envp；

③调用 fork() 函数创建子进程，其地址空间与 shell 父进程完全相同，包括只读代码段、读写数据段、堆及用户栈等

④调用 execve() 函数在当前进程（新创建的子进程）的上下文中加载并运行 hello 程序。将 hello 中的 .text 节、.data 节、.bss 节等内容加载到当前进程的虚拟地址空间

⑤调用 hello 程序的 main() 函数，hello 程序开始在一个进程的上下文中运行。

二. 真题演练

(一) 选择题

- 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
- 不属于进程上下文的是 ()
 A. 页全局目录 pgd B. 内核栈 C. 内核代码 D. 打开的文件表
- 下列说法中哪一个是错误的? ()
 A. 中断一定是异步发生的 B. 异常处理程序一定运行在内核模式下
 C. 故障处理一定返回到当前指令 D. 陷阱一定是同步发生的
- 关于信号的描述, 以下不正确的是哪一个? ()
 A. 在任何时刻, 一种类型至多只会有一个待处理信号
 B. 信号既可以发送给一个进程, 也可以发送给一个进程组
 C. SIGTERM 和 SIGKILL 信号既不能被捕获, 也不能被忽略
 D. 当进程在前台运行时, 键入 Ctrl-C, 内核就会发送一个 SIGINT 信号给这个前台进程
- 学完本课程后, 几位同学聚在一起讨论有关异常的话题, 请问你认为他们中谁学习的结果有错误? ()
 A. 发生异常和异常处理意味着控制流的突变。
 B. 与异常相关的处理是由硬件和操作系统共同完成的。
 C. 异常是由于计算机系统发生了不可恢复的错误导致的。
 D. 异常的发生可能是异步的, 也可能是同步的。
- 下面关于非局部跳转的描述, 正确的是 ()
 A. setjmp 可以和 siglongjmp 使用同一个 jmp_buf 变量
 B. setjmp 必须放在 main() 函数中调用
 C. 虽然 longjmp 通常不会出错, 但仍然需要对其返回值进行出错判断
 D. 在同一个函数中既可以出现 setjmp, 也可以出现 longjmp
- 进程 P1 通过 fork() 函数产生一个子进程 P2。假设执行 fork() 函数之前, 进程 P1 占用了 53 个(用户态的)物理页, 则 fork 函数之后, 进程 P1 和进程 P2 共占用_____个(用户态的)物理页; 假设执行 fork() 函数之前进程 P1 中有一个可读写的物理页, 则执行 fork() 函数之后, 进程 P1 对该物理页的页表项权限为_____。上述两个空格对应内容应该是 ()
 A. 53, 读写 B. 53, 只读 C. 106, 读写 D. 106, 只读
- 在系统调用成功的情况下, 下列代码会输出几个 hello? ()

```

void doit(){
    if ( fork() == 0 ) {
        printf("hello\n");
        fork();
    }
    return ;
}

int main(){
    doit();
    printf("hello\n");
    exit(0) ;
}
      
```

}

A. 3 B. 4 C. 5 D. 6

9. 下列这段代码的输出不可能是 ()

```
int main() {
    signal(SIGCHLD, handler) ;
    if ( fork() == 0 ) {
        printf("a") ;
    } else {
        printf("b") ;
    }
    printf("c") ;
    exit(0) ;
}
```

A. abcc B. abch C. bcach D. bchac

10. 一段程序中阻塞了 SIGCHLD 和 SIGUSR1 信号。然后，向它按顺序发送 SIGCHLD, SIGUSR1, SIGCHLD 信号，当程序取消阻塞继续执行时，将处理这三个信号中的哪几个？ ()

A. 都不处理 B. 处理一次 SIGCHLD
C. 处理一次 SIGCHLD, 一次 SIGUSR1 D. 处理所有三个信号

11. 下列说法正确的是 ()

A. SIGTSTP 信号既不能被捕获，也不能被忽略
B. 存在信号的默认处理行为是进程停止直到被 SIGCONT 信号重启
C. 系统调用不能被中断，因为那是操作系统的工作
D. 子进程能给父进程发送信号，但不能发送给兄弟进程

12. 在系统调用成功的情况下，下面哪个输出是可能的？

```
int main() {
    int pid = fork();
    if (pid == 0) {
        printf("A");
    }
    else {
        pid = fork();
        if (pid == 0) {
            printf("A");
        }
        else {
            printf("B");
        }
    }
}
```



```

    }
    exit(0);
}

```

- A. AAB B. AAA C. AABBB D. AA

13. 对于以下一段代码，可能的输出为：（ ）

```

int count = 0;
int pid = fork();
if (pid == 0){
    printf("count = %d\n",--count);
} else{
    printf("count = %d\n",++count);
}
printf("count = %d\n",++count);

```

- A. 1 2 -1 0 B. 0 0 -1 1 C. 1 -1 0 0 D. 0 -1 1 2

14. 下列哪一事件不会导致信号被发送到进程？（ ）

- A. 新连接到达监听端口 B. 进程访问非法地址
C. 除零 D. 上述情况都不对

15. 每个信号类型都有一个预定义的默认行为，可能是（ ）

- 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

16. C 程序执行到整数或浮点变量除以 0 可能发生（ ）

- A. 显示除法溢出错误直接退出 B. 程序不提示任何错误
C. 可由用户程序确定处理办法 D. 以上都可能

17. 同步异常不包括（ ）

- A. 终止 B. 陷阱 C. 停止 D. 故障

18. 进程上下文切换不会发生在如下（ ）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

19. 一个子进程终止或者停止时，操作系统内核会发送（ ）信号给父进程。

- A. SIGKILL B. SIGQUIT C. SIGSTOP D. SIGCHLD

20. 内核为每个进程维持一个上下文，不属于进程上下文的是（ ）

- A. 寄存器 B. 进程表 C. 文件表 D. 调度程序

21. Linux 进程终止的原因可能是（ ）

- 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

22. 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. 全局变量值 B. 寄存器 C. 虚拟内存一级页表指针 D. 文件表

23. 不属于同步异常的是（ ）

A. 中断 B. 陷阱 C. 故障 D. 终止

24. 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`

25. 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. 复位 D. 故障

26. 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

27. 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. 键盘中断 B. 陷阱 C. 故障 D. 终止

28. 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

29. 下列不属于进程上下文的是（ ）

A. 页全局目录 `pgd` B. 通用寄存器 C. 内核代码 D. 用户栈

30. 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`

31. 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

32. 下列属于操作系统层面的异常控制流（ECF）的是（ ）

A. 中断 B. 上下文切换 C. 信号机制 D. 非本地跳转

33. 进程的状态不包括（ ）

A. 运行 B. 等待 C. 停止 D. 终止

34. 下列关于进程和信号的说法，错误的是（ ）

- A. 前台进程和后台进程的区别在于是否使用 `waitpid()` 函数
- B. 一种类型至多只会有一个待处理信号，而且待处理信号最多只能被接收一次
- C. 在任何时刻，至多只有一个前台作业和 0 个或多个后台作业
- D. 默认情况下，父进程和子进程的 `pid` 和 `pgid` 均不同

35. [多选] 下列说法正确的是（ ）

A. `fork` 的子进程中与其父进程同名的全局变量始终对应同一物理地址。

- B. `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。
- C. 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。
- D. 进程是并发执行的，所以能够并发执行的都是进程

36. [多选] 下列说法错误的是 ()

- A. Linux 系统调用中的功能号 `n` 就是异常号 `n` 。
- B. 进程一旦终止就不再占用内存资源。
- C. 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。
- D. 进程在进行上下文切换时一定会运行内核函数。
- E. 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。

(二) 填空题

- 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，需要通过_____进行实现。
- 异常的有四类：____、____、____、____，其中异步异常是指_____。
- 进程创建函数 `fork` 执行后返回_____次。
- 非本地跳转中的 `setjmp` 函数调用一次，返回_____次。
- 进程加载函数 `execve`，如调用成功则返回_____次。
- 子程序运行结束会向父进程发送_____信号。
- 向指定进程发送信号的 linux 命令是_____。
- CPU 在执行异常处理程序时其模式为_____。
- 当进程在执行时，此时在键盘中键入 Ctrl+C，会导致内核发送_____信号给相关进程

(三) 简答题

- 这个程序会输出_____个 hello 输出行？

```
void doit(){
    Fork();
    Fork();
    printf("hello\n");
    return;
}
```

```
int main(){
    doit();
    printf("hello\n");
    exit(0);
}
```

- 下面程序的一种可能的输出是什么？

```
#include "csapp.h"
int main(){
    int x = 3;
    if (Fork() != 0)
        printf("x=%d\n", ++x);
    printf("x=%d\n", --x);
    exit(0);
}
```

- 下面这个程序的输出是什么？

```
#include "csapp.h."
int counter = 1;
int main(){
    if (fork() == 0){
        counter--;
        exit(0);
    }else {
        Wait(NULL);
        printf("counter = %d\n", ++counter);
    }
    exit(0);
}
```

4. 下面的程序可能的输出序列是什么？

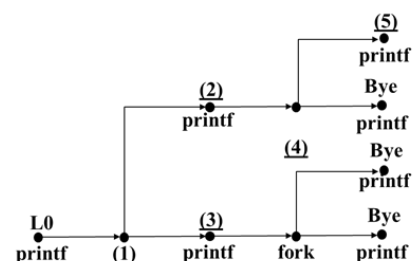
```
int main()
{
    if (fork() == 0) {
        printf("a"); fflush(stdout);
        exit(0);
    }else {
        printf("b"); fflush(stdout);
        waitpid(-1, NULL, 0);
    }
    printf("c"); fflush(stdout);
    exit(0);
}
```

5. 在 shell 命令行输入命令: Ubuntu>./test-trans-M 32 -N 32[回车]。shell 命令行解释器将构造参数 argv 和 envp, 请写出参数 argv 的内容。

| | |
|---------|---------|
| ... | |
| argv[5] | → _____ |
| argv[4] | → _____ |
| argv[3] | → _____ |
| argv[2] | → _____ |
| argv[1] | → _____ |
| argv[0] | → _____ |

6. 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");
}
```

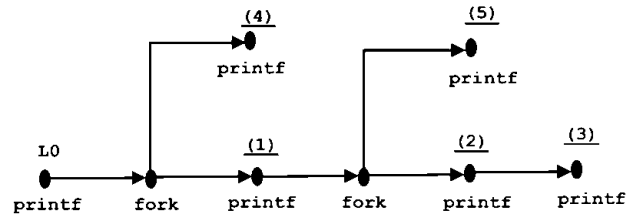


请写出上述进程图中空白处的内容

(1) _____ (2) _____ (3) _____ (4) _____ (5) _____

7. 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) _____ (2) _____ (3) _____ (4) _____ (5) _____

8. 一个 C 程序的 main() 函数如下:

```
int main( ){
    if(fork()==0){
        printf("a"); fflush(stdout);
        exit(0);
    }
    else{
        printf("b"); fflush(stdout);
        waitpid(-1,NULL,0);
    }
    printf("c"); fflush(stdout);
    exit(0);
}
```

(1) 请画出该程序的进程图

(2) 该程序运行后, 可能的输出数列是什么?

9. 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);
```

(四) 综合分析题

1. 请阅读以下程序，然后回答问题（假设程序中的函数调用都可以正确执行）：

```
int main() {
    printf("A\n");
    if (fork() == 0) {
        printf("B\n");
    }
    else {
        printf("C\n");
        A
    }
    printf("D\n");
    exit(0);
}
```

1) 如果程序中的 A 位置的代码为空，列出所有可能的输出结果

2) 如果程序中的 A 位置的代码为：waitpid(-1, NULL, 0)，列出所有可能的输出结果

(3) 如果程序中的 A 位置的代码为：printf("E\n")，列出所有可能的输出结果

2. 以下程序运行时系统调用全部正确执行，且每个信号都被处理到。请给出代码运行后所有可能的输出结果。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
int c = 1;
void handler1(int sig) {
    c++;
    printf("%d", c);
}
答:
int main() {
    signal(SIGUSR1, handler1);
    sigset_t s;
    sigemptyset(&s);
    sigaddset(&s, SIGUSR1);
    sigprocmask(SIG_BLOCK, &s, 0);
    int pid = fork()?fork():fork();
    if (pid == 0) {
        kill(getppid(), SIGUSR1);
        printf("S");
        sigprocmask(SIG_UNBLOCK, &s, 0);
        exit(0);
    } else {
        while (waitpid(-1, NULL, 0) != -1);
        sigprocmask(SIG_UNBLOCK, &s, 0);
        printf("P");
    }
    return 0;
}
```

3. 请阅读以下程序，然后回答问题。假设程序中的函数调用都可以正确执行，并默认 `printf` 执行完会调用 `fflush`。

```
int main() {
    int cnt=1;
    int pid_1,pid_2;
    pid_1=fork();
    if(pid_1==0) {
        pid_2=fork();
        if(pid_2!=0) {
            wait(pid_2,NULL,0);
            printf("B");
        }
        printf("F");
        exit(0);
    }
    else {
        A
        B
        wait(pid_1,NULL,0);
        pid_2=fork();
        if(pid_2==0) {
            printf("D");
            cnt-=1;
        }
        if(cnt==0)
            printf("E");
        else
            printf("G");
        exit(0);
    }
}
```

1) 如果程序中的 A、B 位置的代码为空，列出所有可能的输出结果：

2) 如果程序中的 A、B 位置的代码为：

A: `printf("C");`

B: `exit(0);`

列出所有可能的输出结果：

4. 设一个 C 语言源程序 `p.c` 编译链接后生成执行程序 `p`，反汇编如下：

| C 程序 | 反汇编程序的 main 部分（还有系统代码）如下 |
|---------------------------------------|---|
| <code>#include <stdio.h></code> | 1 <code>movw \$0x3ff, 0x80497d0</code> |
| <code>unsigned short b[2500];</code> | 2 <code>movw 0x804a324, %cx //k->cx</code> |
| <code>unsigned short k;</code> | 3 <code>mov \$0x801, %eax</code> |
| <code>void main(){</code> | 4 <code>xorw %dx, %dx</code> |
| <code>b[1000] = 1023;</code> | 5 <code>div %ecx //2049/d</code> |
| <code>b[2000] = 2049*k;</code> | 6 <code>movw %dx, 0x804a324</code> |
| <code>b[10000] = 20000;</code> | 7 <code>movw \$0x4e20, 0x804de20</code> |
| <code>}</code> | 8 <code>ret</code> |

现代 Intel 桌面系统，采用虚拟页式存储管理，每页 4KB，`p` 首次运行时系统中无其他进程。请结合进程与虚拟存储管理的知识，分析上述程序的执行过程中：

- (1) 在取指令时发生的缺页异常次数为_____。
- (2) 写出已恢复的故障指令序号与故障类型_____。
- (3) 写出没有恢复的故障指令序号与故障类型_____。

第九章、 虚拟内存

一、 真题考点

◆ 考点一：虚拟内存作为缓存的工具

(1) 在任意时刻，虚拟页面的集合都分为三个不相交的子集：

未分配的： VM 系统还未分配（或者创建）的页。未分配的关联，因此也就不占用任何磁盘空间。

缓存的： 当前已缓存在物理内存中的已分配页。

未缓存的： 未缓存在物理内存中的已分配页。

(2) 页表

页表 (page table) 是一个存放在物理内存中的数据结构，页表将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。

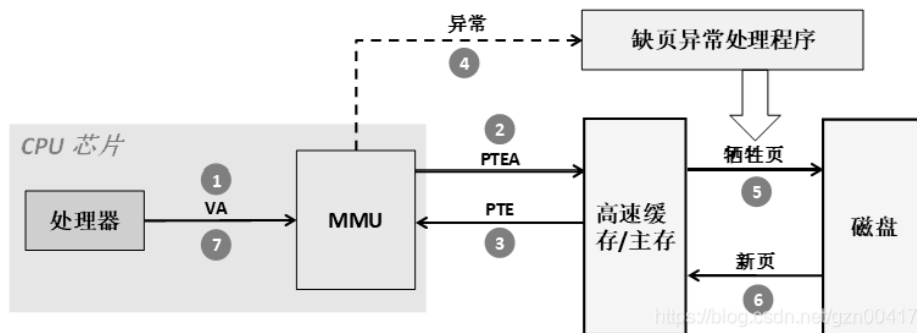
◆ 考点二：地址翻译

1. 虚拟地址和物理地址的组成

| 虚拟地址 (VA) 的组成部分 | | 物理地址 (PA) 的组成部分 | |
|-----------------|--------------|-----------------|--------------|
| 符 号 | 描 述 | 符 号 | 描 述 |
| VPO | 虚拟页面偏移量 (字节) | PPO | 物理页面偏移量 (字节) |
| VPN | 虚拟页号 | PPN | 物理页号 |
| TLBI | TLB 索引 | CO | 缓冲块内的字节偏移量 |
| TLBT | TLB 标记 | CI | 高速缓存索引 |
| | | CT | 高速缓存标记 |

2. 页命中和缺页

页面命中完全是由硬件来处理的，与之不同的是，处理缺页要求硬件和操作系统内核协作完成，当页面不命中时，CPU 硬件执行的步骤：



(1) 处理器生成一个虚拟地址，并把它传送给 MMU

(2) MMU 生成 PTE 地址，并从高速缓存/主存请求得到它。

(3) 高速缓存/主存向 MMU 返回 PTE

(4) **PTE 中的有效位是零**，所以 MMU 触发了一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序。

(5) 缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘。

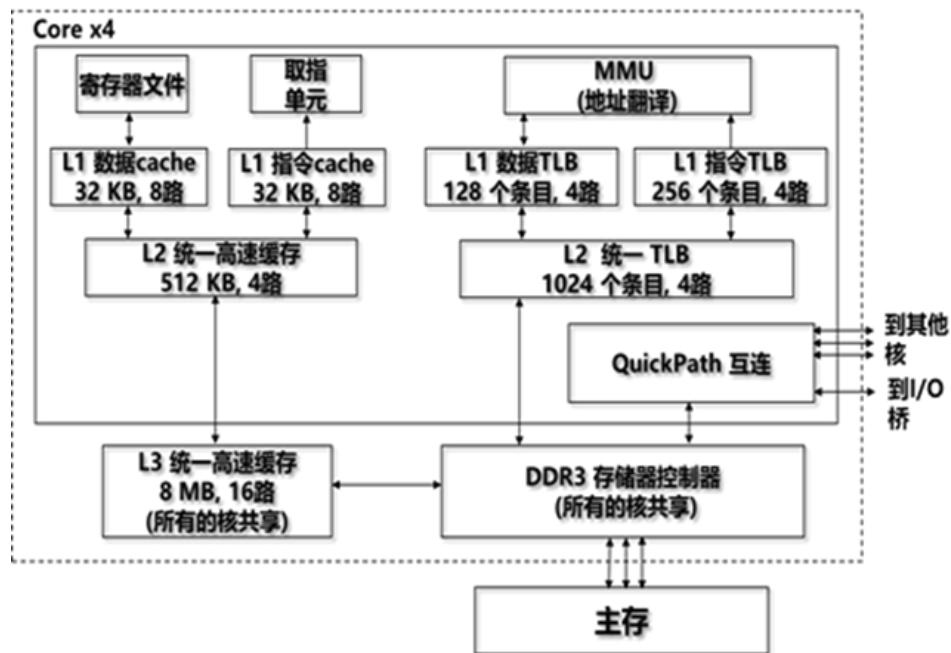
(6) 缺页处理程序页面调入新的页面，并更新内存中的 PTE

(7) 缺页处理程序返回到原来的进程，再次执行导致缺页的指令。**CPU 将引起缺页的虚拟地址重新发送给 MMU**

3. TLB (加速地址翻译)

在 MMU 中包括了一个关于 PTE 的小的缓存，称为翻译后备缓冲器（TLB）

4. Intel Core i7/Linux 内存系统的相关考点

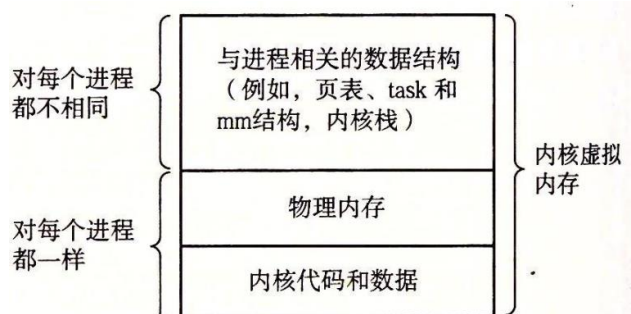


(1) L1、L2 和 L3 高速缓存是物理寻址的，块大小为 64 字节。L1 和 L2 是 8 路组相联的，而 L3 是 16 路组相联的。

(2) 页大小可以在启动时被配置为 4KB 或 4MB，Linux 使用的是 4KB 的页，并使用四级页表。

(3) Linux 虚拟内存空间

Linux 为每个进程维护了一个单独的虚拟地址空间。内核虚拟内存包含内核中的代码和数据结构。内核虚拟内存的某些区域被映射到所有进程共享的物理页面。例如，每个进程共享内核的代码和全局数据结构。



◆ 考点三：内存映射

1. 内存映射

Linux 通过将一个虚拟内存区域与一个磁盘上的对象 (object) 关联起来，以初始化这个虚拟内存区域的内容，这个过程称为**内存映射**。虚拟内存区域可以映射到两种类型的对象中的一种：

(1) Linux 文件系统中的普通文件

(2) 匿名文件：映射到匿名文件的区域中的页面有时也叫做**请求二进制零的页**

无论在哪种情况中 一旦一个虚拟页面被初始化了，它就在一个由内核维护的专门的交换文件之间换来换去。交换文件也叫做交换空间，在任何时刻，交换空间都限制着当前运行着的进程能够分配的虚拟页面的总数。

2. 结合内存映射再看 `fork()`、`execve()`

(1) `fork()`

当 `fork` 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的

PID。为了给这个新进程创建虚拟内存，它创建了当前进程的 `mm_struct`、区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

(2) `execve()`

假设运行在当前进程中的程序执行了如下的 `execve` 调用：

```
execve('a.out', NULL, NULL);
```

`execve` 函数在当前进程中加载并运行包含在可执行目标文件 `a.out` 中的程序，用 `a.out` 程序有效地替代了当前程序。加载并运行 `a.out` 需要以下几个步骤：

I) 删除已存在的用户区域。删除当前进程虚拟地址的用户部分中的已存在的区域结构。

II) 映射私有区域。为新程序的代码、数据、`bss` 和栈区域创建新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 `a.out` 文件中的 `.text` 和 `.data` 区。`bss` 区域是请求二进制零的，映射到匿名文件，其大小包含在 `a.out` 中。栈和堆区域也是请求二进制零的，初始长度为零。

III) 映射共享区域。

VI) 设置程序计数器 (PC)。

◆ 考点四：动态内存分配

1. 分配器

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块 (block) 的集合来维护。每个块就是一个连续的虚拟内存片 (chunk)，要么是已分配的，要么是空闲的。分配器有两种基本风格。

显式分配器：要求应用显式地释放任何已分配的块。

隐式分配器：要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器

2. 碎片

造成堆利用率很低的主要原因是一种称为碎片的现象，当虽然有未使用的内存但不能用来满足分配请求时，就发生这种现象：

内部碎片：是在一个已分配块比有效载荷大时发生的。在任意时刻，内部碎片的数量只取决于以前请求的模式和分配器的实现方式。产生的原因有：1) 维护数据结构产生的开销；2) 增加块大小以满足对齐的约束条件；3) 显式的策略决定 (比如，返回一个大块以满足一个小的请求)

外部碎片：当空闲内存合计起来足够满足一个分配请求，但是没有单独的空闲块足够大可以来处理这个请求时发生的。

3. 空闲块的组织

隐式空闲链表：通过头部中的大小字段—隐含地连接所有块

显式空闲链表：在空闲块中使用指针

分离的空闲列表：按照大小分类，构成不同大小的空闲链表

按块按大小排序—平衡树：在每个空闲块中使用一个带指针的平衡树，并使用长度作为权值

(1) 隐式空闲链表

(a) 概念

在这种情况下，一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小(包括头部和所有的填充)，以及这个块是已分配的还是空闲的。头部后面就是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。



(b) 放置已分配的块——放置策略

一些常见的策略是首次适配、下一次适配和最佳适配。

| 放置策略 | 描述 | 比较 |
|-------|----------------------------|--|
| 首次适配 | 从头开始搜索空闲链表, 选择第一个合适的空闲块 | 速度(吞吐率): 下一次适配 > 首次适配 > 最佳适配 内存利用率: 最佳适配 > 首次适配 > 下一次适配 |
| 下一次适配 | 是从链表的起始处开始每次搜索 | |
| 最佳适配 | 配检查每个空闲块, 选择适合所需请求大小的最小空闲块 | |

(2) 显式的空闲链表

一种更好的方法是将空闲块组织为某种形式的显式数据结构。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred(前驱) 和 siicc(后继) 指针。

(3) 分离的空闲链表

维护多个空闲链表，其中每个链表中的块有大致相等的大小。

a) 简单分离存储

使用简单分离存储，每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小。

b) 分离适配

分配器维护着一个空闲链表的数组。每个空闲链表是和一个小类相关联的，并且被组织成某种类型的显式或隐式链表

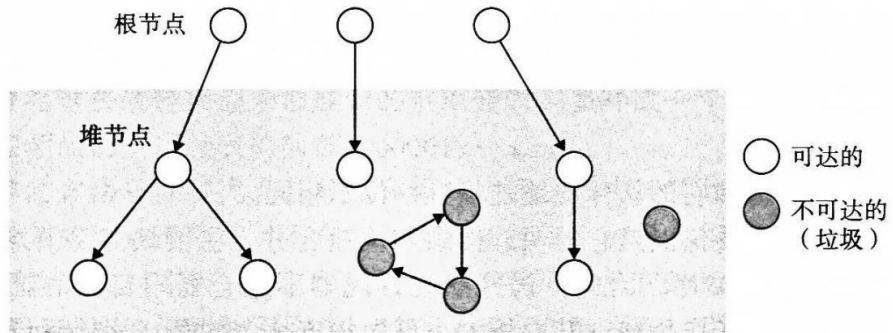
c) 伙伴系统

伙伴系统(buddy system)是分离适配的一种特例，其中每个大小类都是 2 的幂。

◆ 考点五：垃圾收集

垃圾收集器将内存视为一张有向可达图(reachability graph)，该图的节点被分成一组根节点(root node)和一组堆节点(heap node)，每个堆节点对应于堆中的一个已分配块。根节点对应于这样一种不在堆中的位置，它们中包含指向堆中的指针。这些位置可以是寄存器、栈里的

变量，或者是虚拟内存中读写数据区域内的全局变量。



二、 真题演练

(一) 选择题

- 对于虚拟存储系统，一次访存过程中，下列命中组合不可能发生的是()
 - TLB 未命中，Cache 未命中，Page 未命中
 - TLB 未命中，Cache 命中，Page 命中
 - TLB 命中，Cache 未命中，Page 命中
 - TLB 命中，Cache 命中，Page 未命中
- 假设有一台 64 位的计算机的物理页块大小是 8KB（页表条目为 8B），采用三级页表进行虚拟地址寻址，它的虚拟地址的 VPO（虚拟页偏移）有 13 位，问它的虚拟地址的 VPN（虚拟页号码）有多少位？()
 - 20
 - 27
 - 30
 - 33
- 下列哪个例子是外部碎片？()
 - 分配块时为了字节对齐而多分配的空间
 - 空闲块中互相指向的指针所占据的空间
 - 多次释放后形成的不连续空闲块
 - 用户分配后却从未释放的堆空间
- 用带有 header 和 footer 的隐式空闲链表实现分配器时，如果一个应用请求一个 3 字节的块，下列说法哪一项是错误的？()
 - 搜索空闲链表时，存储利用率为：最佳适配 > 首次适配 > 下一次适配
 - 搜索空闲链表时吞吞吐吐率：下一次适配 > 首次适配 > 最佳适配
 - 在 x86 机器上，malloc(3) 实际分配的空闲块大小可能为 8 字节
 - 在 x64 机器上，malloc(3) 返回的地址可能为 2147549777
- 有程序段如下：


```
int foo( ) {
    char str1[20], *str2;
    str2 = (char*)malloc(20*sizeof(char));
    free(str2);
}
```

 str1 和 str2 指向的内存分别是分配在()
 - 栈 栈
 - 堆、堆
 - 栈、堆
 - 堆、栈
- 动态管理器分配策略中，最适合“最佳适配算法”的空白区组织方式是()
 - 按大小递减顺序排列
 - 按大小递增顺序排列
 - 按地址由小到大排列
 - 按地址由大到小排列
- 虚拟内存管理方式可行性的基础是()
 - 程序执行的离散性
 - 程序执行的顺序性
 - 程序执行的局部性
 - 程序执行的并发性
- Intel 的 IA32 体系结构采用二级页表，称第一级页表为页目录（Page Directory），第二级页表为页表（Page Table）。页面的大小为 4KB，页表项 4 字节。以下给出了页目录与若干页表

中的部分内容，例如，页目录中的第 1 个项索引到的是页表 3，页表 1 中的第 3 个项索引到的是物理地址中的第 5 个页。则十六进制逻辑地址 8052CB 经过地址转换后形成的物理地址应为十进制的（ ）

| 页目录 | | 页表 1 | | 页表 2 | | 页表 3 | |
|-----|-----|------|----|------|----|------|----|
| VPN | 页表号 | VPN | 页号 | VPN | 页号 | VPN | 页号 |
| 1 | 3 | 3 | 5 | 2 | 1 | 2 | 9 |
| 2 | 1 | 4 | 2 | 4 | 4 | 3 | 8 |
| 3 | 2 | 5 | 7 | 8 | 6 | 5 | 3 |

- A. 21195 B. 29387 C. 21126 D. 47195

9. 已知某系统页面长 8KB，页表项 4 字节，采用多层分页策略映射 64 位虚拟地址空间。若限定最高层页表占 1 页，则它可以采用多少层的分页策略？（ ）

- A. 3 层 B. 4 层 C. 5 层 D. 6 层

10. 在 C 语言中实现 Mark-and-Sweep 算法时，可以基于以下哪个假设：（宿主机为 32 位机器）（ ）

- A. 所有指针指向一个块的起始地址
B. 所有指针数据都是 4 字节对齐
C. 只需要扫描数据类型为指针的堆中的数据空间
D. 只需要扫描所有长度为 4 字节的堆中的数据空间

11. 在 Core i7 中，以下哪个页表项属于 4 级页表项，不属于 1 级页表项（ ）

- A. G 位 (Global Bit)
B. D 位 (Dirty Bit)
C. XD 位 (Disable or enable instruction fetch)
D. U/S 位 (User or supervisor mode access permission)

12. 在 Core i7 中，关于虚拟地址和物理地址的说法，不正确的是：（ ）

- A. $VPO = CI + CO$
B. $PPN = TLBT + TLBI$
C. $VPN1 = VPN2 = VPN3 = VPN4$
D. $TLBT + TLBI = VPN$

13. Intel x86-64 的现代 CPU，采用（ ）级页表

- A. 2 B. 3 C. 4 D. 由 BIOS 设置确定

14. 存储器垃圾回收时，内存被视为一张有向图，不能作为根结点的是（ ）

- A. 寄存器 B. 栈里的局部变量 C. 全局变量 D. 堆里的变量

15. "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

16. 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（ ）

- A. 程序代码和数据区 B. 栈 C. 共享库 D. 内核虚拟内存区

17. 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
18. 某 CPU 使用 32 位虚拟地址和 4KB 大小的页时, 需要 PTE 的数量是 ()
- A. 16 B. 8 C. 1M D. 512K
19. 动态内存分配时的块结构中, 关于填充字段的作用不可能的是 ()
- A. 减少外部碎片 B. 满足对齐 C. 标识分配状态 D. 可选的
20. 虚拟内存系统中的虚拟地址与物理地址之间的关系是 ()
- A. 1 对 1 B. 多对 1 C. 1 对多 D. 多对多
21. 虚拟内存发生缺页时, 缺页中断是由 () 触发
- A. 内存 B. Cache L1 C. Cache L2 D. MMU
22. 当调用 malloc 这样的 C 标准库函数时, () 可以在运行时动态的扩展和收缩。
- A. 堆 B. 栈 C. 共享库 D. 内核虚拟存储器
23. 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、载入物理内存
24. 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;
25. 虚拟内存发生缺页时, 正确的叙述是 ()
- 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
26. 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, 错误的叙述是 ()
- 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;
27. 某进程在成功执行函数 malloc(24) 后, 下列说法正确的是 ()
- A. 进程一定获得一个大小 24 字节的块 B. 进程一定获得一个大于 24 字节的块
C. 进程一定获得一个不小于 24 字节的块 D. 进程可能获得一个小于 24 字节的块
28. 虚拟页面的状态不可能是 ()
- A. 未分配 B. 已分配未缓存 C. 已分配已缓存 D. 已缓存未分配
29. 动态内存分配时产生内部碎片的原因不包括 ()

A. 维护数据结构的开销

B. 满足对齐约束

C. 分配策略要求

D. 超出空闲块大小的分配请求

30. Linux 进程的虚拟地址空间是私有的，其中哪一项在内核虚拟内存中且对于每个进程都是一样的？（ ）

- A. 用户栈 B. 页表 C. 物理内存 D. 内核栈

31. Linux 缺页处理中，当触发缺页时，会发生下面哪种情况（ ）

- A. 段错误 B. 保护异常 C. 正常缺页 D. 以上都是

32. 以下关于虚拟内存的说法错误的是（ ）

- A. 一个虚拟内存页面每次可以被分配到不同的物理页面
B. 代码、数据和堆都使用相同的起始地址
C. 每个页面被初次引用时，虚拟内存系统会按照需要自动地调入数据页。
D. 不同的虚拟内存页面不能被映射到相同的物理页面

33. 下列关于地址翻译中说法错误的是（ ）

- A. 若发生缺页异常，则由 MMU 触发缺页异常，由 MMU 完成缺页异常处理
B. 缺页异常是一种可恢复的故障
C. 发生缺页时，所选择的牺牲页面若被修改，则需换出到磁盘
D. 可以利用 TLB 加速地址翻译的进行

34. 下列关于 `fork()` 和 `execve()` 的说法错误的是（ ）

- A. `fork` 函数为每个新进程提供私有的虚拟地址空间
B. 新进程拥有与调用 `fork` 进程相同的虚拟内存
C. `execve` 函数加载新程序会覆盖其代码和数据
D. `fork()` 调用一次返回两次，`execve()` 调用一次返回多次

35. 记录和组织空闲块的方法不包括（ ）

- A. 分离的空闲列表 B. 显式空闲链表 C. 垃圾收集 D. 隐式空闲链表

36. 下列关于动态内存分配的说法正确的是（ ）

- A. 在一个伙伴系统中，最高可达 50% 的空间可以因为内部碎片而被浪费了。
B. 首次适配内存分配算法比最佳适配算法要慢一些（平均而言）。
C. 只有当空闲链表按照内存地址递增排序时，使用边界标记来回收才会快速。
D. 伙伴系统只会有内部碎片，而不会有外部碎片。

37. 下面关于空闲链表与适配算法的说法正确的是（ ）

- A. 在按照块大小递减顺序排序的空闲链表上，使用首次适配算法会导致分配性能很低，但是可以避免外部碎片。
B. 对于最佳适配方法，空闲块链表应该按照内存地址的递增顺序排序。
C. 最佳适配方法选择与请求段匹配的最大的空闲块。
D. 在按照块大小递增的顺序排序的空闲链表上，使用首次适配算法与使用最佳适配算法等价。

38. Mark&Sweep 垃圾收集器在下列哪种情况下叫做保守的：

- A. 它们只有在内存请求不能被满足时才合并被释放的内存。

- B. 它们把一切看起来像指针的东西都当做指针。
- C. 它们只在内存用尽时,才执行垃圾收集。
- D. 它们不释放形成循环链表的内存块。

39. 下列说法错误的是()

- A. 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。
- B. 在动态内存分配中,内部碎片不会降低内存利用率。
- C. 系统中当前运行进程能够分配的虚拟页面的总数取决于虚拟地址空间的大小。
- D. 虚拟内存系统能有效工作的前提是软件系统具有局部性。

40. 下列说法正确的是()

- A. 动态存储器分配时显式空闲链表比隐式空闲链表的实现节省空间。
- B. 动态内存隐式分配是指应用隐式地分配块并隐式地释放已分配块。
- C. 显式空闲链表的优点是在对堆块进行搜索时,搜索时间只与堆中的空闲块数量成正比。
- D. 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。
- E. 隐式空闲链表的优点是在对堆块进行搜索时,搜索时间只与堆中的空闲块数量成正比。

(二) 填空题

1. C 语言函数中的整数常量都存放在程序虚拟地址空间的_____段。
2. TLB(翻译后备缓冲器)俗称快表,是_____的缓存。
3. 虚拟页面的状态有_____,_____,未缓存共 3 种
4. 虚拟内存系统借助_____这一数据结构将虚拟页映射到物理页。
5. Linux 虚拟内存区域可以映射到普通文件和_____,这两种类型的对象中的一种。
6. 程序运行时,指令中的立即操作数存放的内存段是_____段。
7. 虚拟内存存在内存映射时,映射到匿名文件的页面是_____的页。
8. 虚拟内存发生缺页时,MMU 将触发_____。
9. 对一个给定块,当有效荷载小于块的大小时会产生_____ (内部碎片/外部碎片)
10. 私有对象使用一种叫做_____技术被映射到虚拟内存中

(三) 简答题

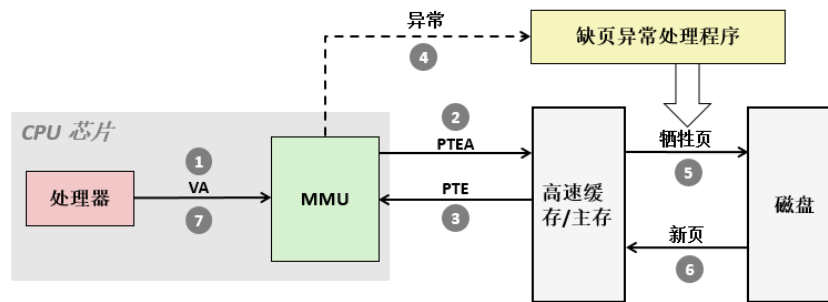
1. 假设:某 CPU 的虚拟地址 14 位;物理地址 12 位;页面大小为 64B;TLB 是四路组相联,共 16 个条目;L1 数据 Cache 是物理寻址、直接映射,行大小为 4 字节,总共 16 个组。分析如下项目:

(1) 虚拟地址中的 VPN 占___位;物理地址的 PPN 占___位。

(2) TLB 的组索引位数 TLBI 为___位。

(3) 用物理地址访问 L1 数据 Cache 时,Cache 的组索引 CI 占___位,Cache 标记 CT 占___位。

2. 结合下图,简述虚拟内存地址翻译的过程。

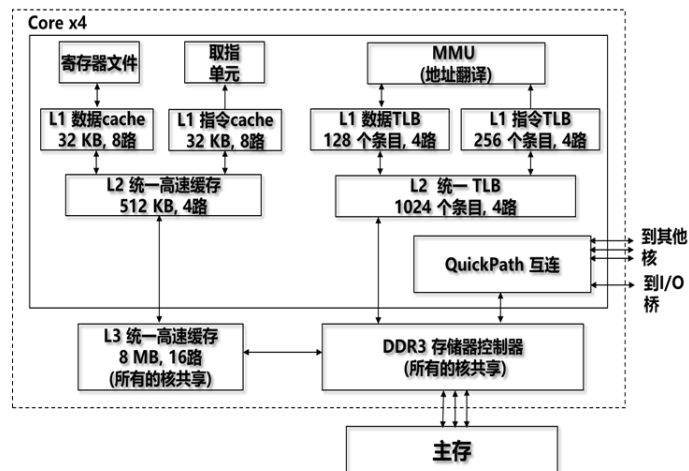


3. Intel I7 CPU 的虚拟地址 48 位，物理地址 52 位。其内部结构如下图所示，依据此结构，每一页面 4KB，分析如下项目：

虚拟地址中的 VPN 占_____位；其一级页表为_____项。

L1 数据 TLB 的组索引位数 TLBI 为_____位，L1 数据 Cache 共_____组(假设其块大小为 64B)。

用物理地址访问 L1 数据 Cache 时，Cache 标记 CT 占_____位



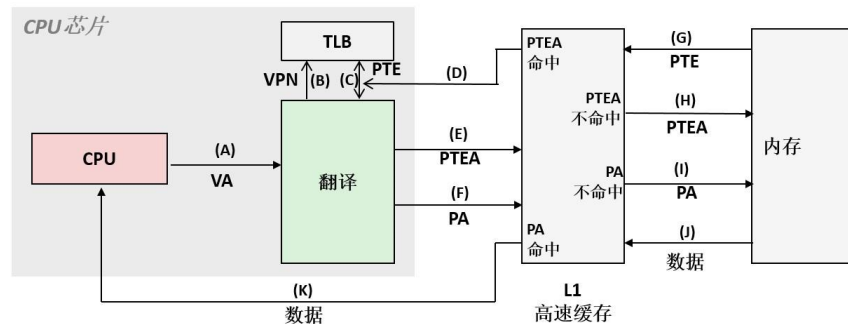
4. [2017A]在终端中的命令行运行显示“Hello World”的执行程序 hello，结合进程创建、加载、缺页中断、到存储访问（虚存）.....等等，论述 hello 是怎么一步步执行的。

5. 下图展示了一个虚拟地址的访存过程，每个步骤采用不同的字母表示。请分别针对下述情况，用字母序列写出每种情况下的执行流程：

(1) TLB 命中、缓存 物理地址命中；

(2) TLB 不命中, 缓存页表命中, 缓存物理地址命中;

(3) TLB 不命中, 缓存页表不命中, 缓存物理地址不命中。



(四) 综合分析题

1、假设有以下一个 TLB 和 L1 d-cache 的小系统, 我们做出如下假设:

- (1) 内存是按字节寻址的, 内存访问是针对 1 字节的字的 (不是 4 字节的字)。
- (2) 虚拟地址是 14 位长的 ($n=14$), 物理地址是 12 位长的 ($m=12$)。
- (3) 页面大小是 64 字节 ($P=64$)
- (4) TLB 是四路组相联的, 总共有 16 个条目。
- (5) L1 d-cache 是物理寻址、直接映射的, 行大小为 4 B, 而总共有 16 个组。

| Set | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid |
|-----|-----|-----|-------|-----|-----|-------|-----|-----|-------|-----|-----|-------|
| 0 | 03 | — | 0 | 09 | 0D | 1 | 00 | — | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | — | 0 | 04 | — | 0 | 0A | — | 0 |
| 2 | 02 | — | 0 | 08 | — | 0 | 06 | — | 0 | 03 | — | 0 |
| 3 | 07 | — | 0 | 03 | 0D | 1 | 0A | 34 | 1 | 02 | — | 0 |

(a) TLB: 4 sets, 16 entries, 4-way set associative

| VPN | PPN | Valid |
|-----|-----|-------|
| 00 | 28 | 1 |
| 01 | — | 0 |
| 02 | 33 | 1 |
| 03 | 02 | 1 |
| 04 | — | 0 |
| 05 | 16 | 1 |
| 06 | — | 0 |
| 07 | — | 0 |

| VPN | PPN | Valid |
|-----|-----|-------|
| 08 | 13 | 1 |
| 09 | 17 | 1 |
| 0A | 09 | 1 |
| 0B | — | 0 |
| 0C | — | 0 |
| 0D | 2D | 1 |
| 0E | 11 | 1 |
| 0F | 0D | 1 |

(b) Page table: Only the first 16 PTEs are shown

| Idx | Tag | Valid | Blk 0 | Blk 1 | Blk 2 | Blk 3 |
|-----|-----|-------|-------|-------|-------|-------|
| 0 | 19 | 1 | 99 | 11 | 23 | 11 |
| 1 | 15 | 0 | — | — | — | — |
| 2 | 1B | 1 | 00 | 02 | 04 | 08 |
| 3 | 36 | 0 | — | — | — | — |
| 4 | 32 | 1 | 43 | 6D | 8F | 09 |
| 5 | 0D | 1 | 36 | 72 | F0 | 1D |
| 6 | 31 | 0 | — | — | — | — |
| 7 | 16 | 1 | 11 | C2 | DF | 03 |
| 8 | 24 | 1 | 3A | 00 | 51 | 89 |
| 9 | 2D | 0 | — | — | — | — |
| A | 2D | 1 | 93 | 15 | DA | 3B |
| B | 0B | 0 | — | — | — | — |
| C | 12 | 0 | — | — | — | — |
| D | 16 | 1 | 04 | 96 | 34 | 15 |
| E | 13 | 1 | 83 | 77 | 1B | D3 |
| F | 14 | 0 | — | — | — | — |

(c) Cache: 16 sets, 4-byte blocks, direct mapped

对于给定的虚拟地址，指明访问的 TLB 条目、物理地址和返回的缓存字节值。指出是否发生了 TLB 不命中，是否发生了缺页，以及是否发生了缓存不命中。如果是缓存不命中，在“返回的缓存字节”栏中输入“—”。如果有缺页，则在“PPN”一栏中输入一栏中输入“—”，并且将 C 部分和 D 部分空着。

1) 虚拟地址：0x03d7

A. 虚拟地址格式

| | | | | | | | | | | | | | |
|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | | | | | | | | | |

B. 地址翻译

| 参数 | 值 |
|---------------|---|
| VPN | |
| TLB 索引 | |
| TLB 标记 | |
| TLB 命中? (Y/N) | |
| 缺页? (Y/N) | |
| PPN | |

C. 物理地址格式

| | | | | | | | | | | | |
|----|----|---|---|---|---|---|---|---|---|---|---|
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | | | | | | | |

D. 物理内存引用

| 参数 | 值 |
|-------------|---|
| 字节偏移 | |
| 缓存索引 | |
| 缓存标记 | |
| 缓存命中? (是/否) | |
| 返回的缓存字节 | |

第一章 计算机系统漫游

一、选择题

1. 计算机操作系统抽象表示时()是对处理器、主存和 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
2. 操作系统通过提供不同层次的抽象表示来隐藏系统实现的复杂性,其中()是对实际处理器硬件的抽象。
A. 进程 B. 虚拟存储器 C. 文件 D. 指令集架构 (ISA)
3. 操作系统提供的抽象表示中,()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
4. 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. ccl C. as D. ld
5. 当函数调用时,()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. 内核虚拟存储器
6. 当调用 malloc 这样的 C 标准库函数时,()可以在运行时动态的扩展和收缩。
A. 堆 B. 栈 C. 共享库 D. 内核虚拟存储器

第二章 信息的表示与处理

一、 选择题

1. C 语句中的有符号常数, 在 () 阶段转换成了补码
A. 编译 B. 链接 C. 执行 D. 调试
2. C 语言中整数-1 与无符号 0 比较, 其结果是 ()
A. 大于 B. 小于 C. 可能大于可能小于 D. 无法比较
3. 计算机常用信息编码标准中, 字符 0 的编码不可能是 16 进制数 ()
A. 30 B. 30 00 C. 00 D. 00 30
4. C 语言中 float 类型的数据 0.1 的机器数表示, 错误的是 ()
A. 规格化数 B. 不能精确表示 C. 与 0.2 有 1 个二进制位不同 D. 唯一的
5. 下列 16 进制数值中, 可能是 Linux64 系统中 char* 类型的指针值是 ()
A. e4f9 B. b4cc2200 C. b811e5ffff7f0000 D. 30
6. 关于 IEEE float 类型的数据+0.0 的机器数表示, 说法错误的是 ()
A. 是非规格化数 B. 不能精确表示 C. +0.0 与-0.0 不同 D. 唯一的
7. 程序中的 2 进制、10 进制、16 进制数, 在 () 时变成 2 进制
A. 汇编时 B. 连接时 C. 执行时 D. 调试时
8. C 语言程序中的整数常量、整数常量表达式是在 () 阶段变成 2 进制补码的。
A. 预处理 B. 编译 C. 链接 D. 执行
9. C 语言中不同类型的数值进行强制类型转换时, 下列说法错误的是 ()
A. 从 int 转换成 float 时, 数值可能会溢出
B. 从 int 转换成 double 后, 数值不会溢出
C. 从 double 转换成 float 时, 数值可能会溢出, 也可能舍入
D. 从 double 转换成 int 时, 数值可能溢出, 可能舍入
10. 在 IEEE 浮点数标准中, 单精度浮点数采用 () 位的小数字段对尾数进行编码。
A. 23 B. 24 C. 52 D. 63
11. 给定字长的整数 x 和 y 按补码相加, 和为 s, 则发生正溢出的情况是 ()
A. $x > 0, y > 0, s \leq 0$ B. $x > 0, y < 0, s \leq 0$
C. $x > 0, y < 0, s \geq 0$ D. $x < 0, y < 0, s \geq 0$
12. C 语言程序中的常量表达式的计算是由 () 完成的
A. 编辑器 B. 编译器 C. 链接器 D. 加载器

二、 填空题

1. 64 位系统中 int 数 -2 的机器数二进制表示_____。
2. Intel 桌面 X86-64 CPU 采用_____端模式。
3. C 语言中 short 类型-2 的机器数二进制表示为_____。
4. C 语言中的 double 类型浮点数用_____位表示。
5. 判断整型变量 n 的位 7 为 1 的 C 语言表达式是_____。
6. 整型变量 x=-2, 其在内存从低到高依次存放的数是 (16 进制表示)_____。

7. 若字节变量 x 和 y 分别为 $0x10$ 和 $0x01$, 则 C 表达式 $x\&\&\sim y$ 的字节值是_____。
8. 按照“向偶数舍入”的规则, 二进制小数 101.110_2 舍入到最接近的 $1/2$ (小数点右边 1 位) 后的二进制为_____。
9. C 程序中定义 `int x=-3`, 则 `&x` 处依次存放 (小端模式) 的十六进制数据为_____。
10. C 语言的常量表达式的计算是由_____完成的
11. 若 x 和 y 的字节值分别为 $0x12$ 和 $0x34$, 则 C 表达式 $x\&\&y$ 的值为_____。

三、 判断题

1. () C 语言中从 `int` 转换成 `float` 时, 数字不会溢出, 但可能舍入。
2. () C 语言程序中, 有符号数强制转换成无符号数时, 其二进制将会做相应调整。
3. () C 语言对整型指针 `p`, 当 `p=null` 时, 表达式 `p&&*p++` 会间接引用空指针。
4. () C 语言中, 关系表达式: `127 > (unsigned char)128U` 是成立的。
5. () `x` 和 `y` 是 C 中的整型变量, 若 `x` 大于 0 且 `y` 大于 0, 则 `x+y` 一定大于 0。
6. () CPU 无法判断参与加法运算的数据是有符号或无符号数。
7. () C 浮点常数 IEEE754 编码的缺省舍入规则是四舍五入。
8. () 对 `unsigned int x`, `(x*x) >= 0` 总成立。
9. () 一个整型机器数采用大端还是小端方式存储, 其值是不同的。
10. () CPU 无法判断加法运算的和是否溢出。
11. () 浮点数 IEEE 标准中, 规格化数比非规格化数多。
12. () C 浮点常数 IEEE754 编码的缺省舍入规则是向上舍入。
13. () C 语言中的有符号数强制转换成无符号数时位模式不会改变。
14. () C 语言中从 `double` 转换成 `float` 时, 值可能溢出, 但不可能被舍入。
15. () C 语言中 `int` 的个数比 `float` 个数多。
16. () 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。

四、简答题

1. 写出 `float f=-1` 的 IEEE754 编码。（请按步骤写出转换过程）
2. 请在数轴上画出非负 `float` 数的各区间的密度分布，并标示各区间是规格化还是非规格化数、浮点数密度、最小值、最大值。

3. 请结合 ieee754 编码, 说明怎样判断两个浮点数是否相等?

第三章 程序的机器级表示

一、选择题

- 当函数调用时, () 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. 内核虚拟存储器
- 递归函数程序执行时, 正确的是 ()
A. 使用了堆 B. 可能发生栈溢出 C. 容易有漏洞 D. 必须用循环计数器
- 下列叙述正确的是 ()
A. 一条 mov 指令不可以使用两个内存操作数
B. 在一条指令执行期间, CPU 不会两次访问内存
C. CPU 不总是执行 CS::RIP 所指向的指令, 例如遇到 call、ret 指令时
D. X86-64 指令 "movl \$1,%eax" 不会改变 %rax 的高 32 位
- 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
- 在 x86-64 中, 有初始值 %rax = 0x1122334455667788, 执行下述指令后 %rax 寄存器的值是 ()

```
movl $0xaa11, %rax
```

- 0xaa11 B. 0x112233445566aa11
C. 0x112233440000aa11 D. 0x11223344ffffaa11
- x86-64 系统中, 函数 int sum (int x, int y) 经编译后其返回值保存在 ()
A. %rdi B. %rsi C. %rax D. %rdx
 - 在 x86-64 系统中, 调用函数 int gt (long x, long y) 时, 保存参数 y 的寄存器是 ()
A. %rdi B. %rsi C. %rax D. %rdx
 - x86-64 中, 某 C 程序定义了结构体

```
struct SS {
    double v;
    int i;
    short s;
} aa[10];
```

则执行 sizeof(aa) 的值是 ()

- 14 B. 80 C. 140 D. 160
- 条件跳转指令 JZ 是依据 () 做是否跳转的判断。
A. ZF B. OF C. SF D. CF

二、填空题

- 64 位 C 语言程序中第一个参数采用_____传递。
- 64 位 C 语言程序在函数调用时第二个整型参数采用寄存器_____传递。
- C 语言 64 位系统中参数传递可采用_____、_____。
- 将 hello.c 编译生成汇编语言的命令行_____。

5. 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_____。

三、 判断题

1. () X86-64 CPU 中的寄存器一定都是 64 位的。
2. () 使用栈随机化的方法不能完全避免针对缓冲区溢出的攻击。
3. () C 函数调用过程中, 调用函数的栈帧一旦被修改, 被调用函数则无法正确返回。

四、 简答题

1. 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)。

2. 简述缓冲区溢出攻击的原理以及防范方法。

3. 下列 C 程序存在安全漏洞, 请给出攻击方法。如何修复或防范?

```
int getbuf(char *s) {  
    char buf[32];  
    strcpy( buf, s );  
}
```

五、分析题

1. 有下列 C 函数，右边是函数 arith 的汇编代码，请填写缺失的汇编代码

```
long arith(long x, long y, long z){    arith:
    long t1 = _____;           xorq   %rsi,%rdi
    long t2 = _____;           leaq   (%rdi,%rdi,4),%rax
    long t3 = _____;           leaq   (%rax,%rsi,2),%rax
    long t4 = _____;           subq   %rdx,%rax
    _____;                     retq
}
```

2. 已知内存和寄存器中的数值情况如下：

| 内存地址 | 值 |
|-------|------|
| 0x100 | 0xff |
| 0x104 | 0xAB |
| 0x108 | 0x13 |
| 0x10c | 0x11 |

| 寄存器 | 值 |
|------|-------|
| %rax | 0x100 |
| %rcx | 0x1 |
| %rdx | 0x3 |
| | |

请填写下表，给出对应操作数的值：

| 操作数 | 值 |
|---------------|---|
| %rax | |
| (%rax) | |
| 9(%rax,%rdx) | |
| 0xfc(,%rcx,4) | |
| (%rax,%rdx,4) | |

3. 请填写出上述 C 语言代码中缺失的部分

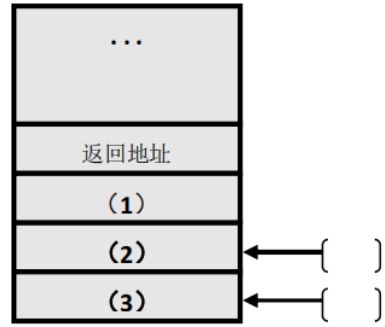
有下列 C 函数：

函数 arith 的汇编代码如下：

```
long arith(long x, long y, long z)    arith:
{                                     orq   %rsi,%rdi
    long t1 = _____;           sarq   $3,%rdi
    long t2 = _____;           notq   %rdi
    long t3 = _____;           movq   %rdx,%rax
    long t4 = _____;           subq   %rdi,%rax
    _____;                     retq
}
```

4. 函数 call_incr2 的汇编代码如下所示，画出函数 call_incr2 相应的栈帧内结构与内容。

```
call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



请填写出上述栈帧缺失的内容

- (1) _____
- (2) _____
- (3) _____
- (4) _____
- (5) _____

第五章 优化程序性能

一、选择题

1. 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
2. 利用 GCC 生成代码过程中，不属于编译器优化的结果是（ ）
- A. 用移位操作代替乘法指令
 - B. 消除循环中的函数调用
 - C. 循环展开
 - D. 使用分块提高时间局部性

二、简答题

1. 列举几种程序优化的方法，并简述其原理。

2. 列举至少 5 种程序优化的方法。

三、分析题

1. **[2017A]** 程序优化： 矩阵 $c[n][n] = a[n][n] * b[n][n]$ ，采用 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];
    }
}
```

```
}
```

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。

2. 程序优化：矩阵 $c[n][n] = a[n][n] * b[n][n]$ ，采用 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];  
        }  
    }  
}
```

请给出基于编译、CPU、存储器的三种优化方法，并编写程序。

3. 现代超标量 CPU X86-64 的 Cache 的参数 $s=5$, $E=1$, $b=5$, 若 $M=N=64$, 请优化如下程序, 并说明优化的方法 (至少 CPU 与 Cache 各一种)。

```
void trans(int M, int N, int A[M][N], int B[N][M])
{
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            B[j][i] = A[i][j];
}
```

4. 优化如下程序, 给出优化结果并说明理由。

```
int sum_array(int a[M][N][N]) {    //M、N 足够大
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```



```
}
```

5. 向量元素和计算的相关程序如下, 请改写或重写计算函数 `vector_sum`, 进行速度优化, 并简要说明优化的依据。

```
/*向量的数据结构定义 */
```

```
typedef struct{
```

```
    int len;           //向量长度, 即元素的个数
```

```
    float *data;       //向量元素的存储地址
```

```
} vec;
```

```
/*获取向量长度*/
```

```
int vec_length(vec *v){return v->len;}
```

```
/* 获取向量中指定下标的元素值, 保存在指针参数 val 中*/
```

```
int get_vec_element(*vec v, size_t idx, float *val){
```

```
    if (idx >= v->len)
```

```
        return 0;
```

```
        *val = v->data[idx];
```

```
        return 1;
```

```
}
/*计算向量元素的和*/
void vector_sum(vec *v, float *sum){
    long int i;
    *sum = 0;                                //初始化为 0
    for (i = 0; i < vec_length(v); i++) {
        float val;
        get_vec_element(v, i, &val); //获取向 v 中第 i 个元素的值, 存入 val
        *sum = *sum + val;           //将 val 累加到 sum 中
    }
}
```

第六章 存储器层次结构

一、选择题

1. 位于存储器层次结构中的最顶部的是()。
A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存
2. CPU 一次访存时,访问了 L1、L2、L3 Cache 所用地址 A1、A2、A3 的关系()
A. $A1 > A2 > A3$ B. $A1 = A2 = A3$ C. $A1 < A2 < A3$ D. $A1 = A2 < A3$
3. 下列各种存储器中存储速度最快的是()。
A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存
4. 采用缓存系统的原因是()
A. 高速存储部件造价高 B. 程序往往有比较好的空间局部性
C. 程序往往有比较好的时间局部性 D. 以上都对
5. 寄存器作为计算机缓存层次结构的最高层,决定哪个寄存器存放某个数据的是()
A. MMU B. 操作系统内核 C. 编译器 D. CPU
6. 下列各种存储器中存储速度最慢的是()
A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存

二、填空题

1. I7 CPU, L2 Cache 为 8 路的 2M 容量, $B=64$, 则其 Cache 组的位数 $s=$ _____。
2. 某 CPU 主存地址 32 位, 高速缓存总大小为 4K 行, 块大小 16 字节, 采用 4 路组相连, 则标记位的总位数(每行标记位数*总行数)是_____。
3. 存储器层次结构中, 高速缓存(Cache)是_____的缓存。
4. Cache 命中率分别是 97%和 99%时, 访存速度差别_____ (很大/很小)。
5. 在计算机的存储体系中, 速度最快的是_____。
6. 当工作集的大小超过高速缓存的大小时, 会发生_____不命中
7. 在计算机存储层次结构中, _____是磁盘的缓存。
8. 程序所具有的_____特点使得高速缓存能够有效。
9. 若主存地址 32 位, 高速缓存总大小为 2K 行, 块大小 16 字节, 采用 2 路组相连, 则标记位的总位数是_____。
10. 若高速缓存的块大小为 B ($B > 8$) 字节, 向量 v 的元素为 `int`, 则对 v 的步长为 1 的应用的不命中率为_____。

三、判断题

1. () 全相联高速缓存不会发生冲突不命中的情况。
2. () 直接映射高速缓存一定会发生冲突不命中的情况。
3. () Cache 的大小对程序运行非常重要, 必要的时候可以通过操作系统提高 Cache 的大小。
4. () CPU 在同一次访问 Cache L1、L2、L3 时使用的地址是一样的。

四、简答题

1. 简述程序的局部性原理, 如何编写局部性好的程序?

2. 结合下面的程序段，解释局部性。

```
int cal_array_sum(int *a,int n){
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

3. 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 CO、组索引 CI、标记 CT 分别多少？

4. 某高速缓存大小 256 字节，直接映射，块大小为 16 字节。定义 L 为数据装载命令，S 为存储，M 为数据修改。若每一数据装载(L)或存储(S)操作可引发最多 1 次缓存缺失(miss)；数据修改操作(M)可认为是同一地址上 1 次装载后跟 1 次存储，因此可引发 2 次缓存命中(hit)或 1 次缺失加 1 次命中外加可能的 1 次淘汰/驱逐(evict)。根据下列的访存命令序列，分析每一命令下的上述高速缓存（高速缓存最初是空的）的命中及淘汰情况。

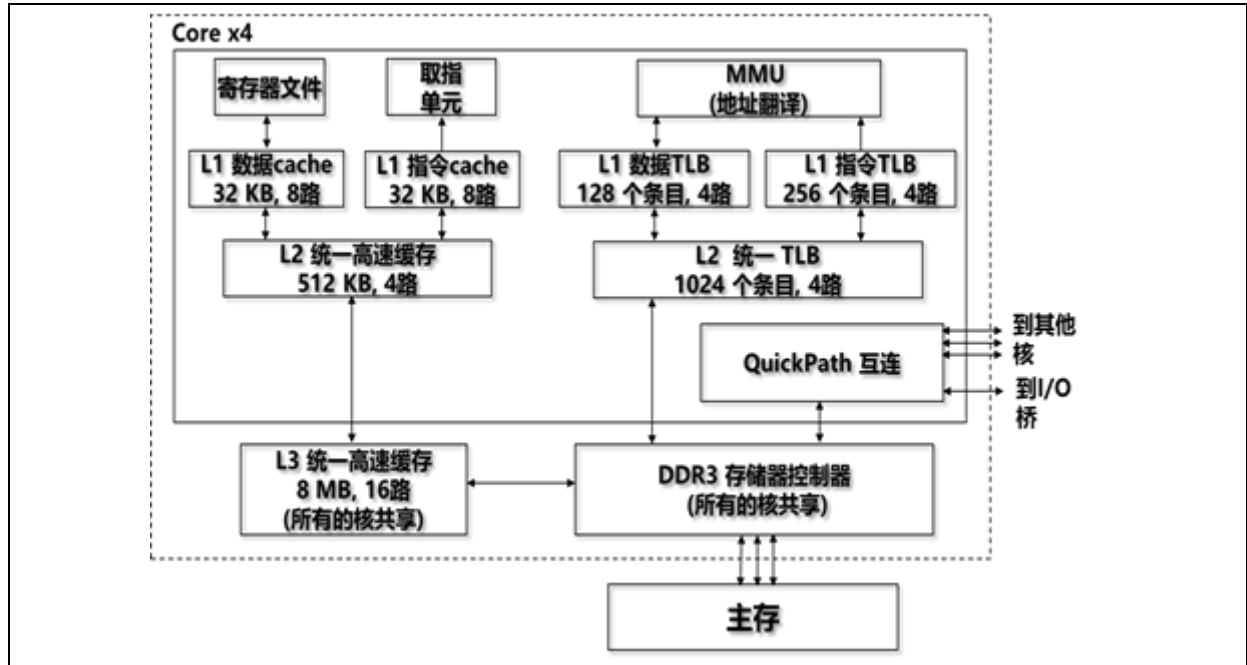
L 10,1 → M 20,1 → L 22,1 → S 18,1 → L 110,1 → L 210,1 → M 12,1

说明：L 10,1 表示从地址 0x10 处加载 1 个字节数据，其它同理。

5. 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_____（___位）；访问 L1 TLB 的 TLBI 为_____（___位）；

若指令 A 的物理地址为 0x86049b，则该地址对应的 PPN 为_____（___位）；访问 L1 cache 的 CT 为_____（___位），CO 为_____（___位）。



第七章 链接

一、选择题

1. 链接时两个文件同名的弱符号, 以 () 为基准
A. 连接时先出现的 B. 连接时后出现的 C. 任一个 D. 连接报错
2. 链接过程中, 赋初值的局部变量名, 正确的是 ()
A. 强符号 B. 弱符号 C. 若是静态的则为强符号 D. 以上都错
3. C 语句中的全局变量, 在 () 阶段被定位到一个确定的内存地址
A. 编译 B. 链接 C. 执行 D. 调试
4. 链接时两个同名的强符号, 以哪种方式处理? ()
A. 链接时先出现的符号为准 B. 链接时后出现的符号为准
C. 任一个符号为准 D. 链接报错
5. 链接过程中, 带 static 属性的全局变量属于 ()
A. 全局符号 B. 局部符号 C. 外部符号 D. 以上都错
6. 以下关于程序中链接“符号”的陈述, 错误的是 ()
A. 赋初值的非静态全局变量是全局强符号 B. 赋初值的静态全局变量是全局强符号
C. 未赋初值的非静态全局变量是全局弱符号 D. 未赋初值的静态全局变量是本地符号
7. 关于动态库的描述错误的是 ()
E. 可在加载时链接, 即当可执行文件首次加载和运行时进行动态链接。
F. 更新动态库, 即便接口不变, 也需要将使用该库的程序重新编译。
G. 可在运行时链接, 即在程序开始运行后通过程序指令进行动态链接。
H. 即便有多个正在运行的程序使用同一动态库, 系统也仅在内存中载入一份动态库。
8. 关于局部变量, 正确的叙述是 ()
E. 普通 (auto) 局部变量也是一种编程操作的数据, 存放在数据段
F. 非静态局部变量在链接时是本地符号
G. 静态局部变量是全局符号
H. 编译器可将 %rsp 减取一个数为局部变量分配空间
9. Linux 系统中将可执行目标文件 (.out 文件) 装入到存储空间时, 没有装入到 .text 段-只读代码段的是 ()
A. ELF 头 B. .init 节 C. .rodata 节 D. .symtab 节
10. 链接过程中, 赋初值的静态全局变量属于 ()
A. 强符号 B. 弱符号 C. 可能是强符号也可能是弱符号 D. 以上都不是
11. 链接过程中, 赋初值的非静态全局变量名, 属于 ()
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
12. 共享库 (动态链接库) 在程序的 () 阶段由动态链接器加载到任意的内存地址。
A. 编译 B. 链接 C. 运行 D. 运行或链接

二、填空题

1. 链接器经过_____和重定位两个阶段, 将可重定位目标文件生成可执行目标文件。

2. 若 `p.o->libx.a->liby`.且 `liby.a->libx.a->p.o` 则最小链接命令行_____。
3. 可重定位目标文件中代码地址从_____开始。
4. C 语言程序运行时, 局部变量存在放_____段。
5. C 语句中的全局变量, 在_____阶段被定位到一个确定的内存地址。

三、 判断题

1. () 链接时, 若一个强符号和多个弱符号同名, 则对弱符号的引用均将被解析成强符号。
2. () 一个 C 程序中的跳转表数据经链接后被映射到代码段。

四、 简答题

1. 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。

2. 什么是共享库(动态链接库)? 简述动态链接的实现方法。

3. 什么是静态库? 使用静态库的优点是什么?

五、 分析题

1-3 两个 C 语言程序 `main.c`、`test.c` 如下所示:

```
/* main.c */
#include <stdio.h>
int a[4]={-1,-2,2, 3};
extern int val;
int sum();
int main(int argc, char * argv[ ] ){

/* test.c */
extern int a[ ];
int val=0;
int sum( ){
    int i;
    for (i=0; i<4; i++)
```

```

    val=sum();                                val += a[i];
    printf("sum=%d\n",val);                    return val;
}                                              }

```

用如下两条指令编译、链接，生成可执行程序 test：

```

gcc -m64 -no-pie -fno-PIC -c test.c main.c
gcc -m64 -no-pie -fno-PIC -o test test.o main.o

```

运行指令 objdump -dxs main.o 输出的部分内容如下：

```

Contents of section .data:
0000 ffffffff feffffff 02000000 03000000 .....
Contents of section .rodata:
0000 73756d3d 25640a00          sum=%d..
...
Disassembly of section .text:
0000000000000000 <main>:
    0: 55          push    %rbp
    1: 48 89 e5    mov     %rsp,%rbp
    4: 48 83 ec 10 sub     $0x10,%rsp
    8: 89 7d fc    mov     %edi,-0x4(%rbp)
    b: 48 89 75 f0 mov     %rsi,-0x10(%rbp)
    f: b8 00 00 00 00 mov     $0x0,%eax
   14: e8 00 00 00 00 callq   19 <main+0x19>
       15: R X86 64 PC32  sum-0x4
   19: 89 05 00 00 00 00 mov     %eax,0x0(%rip) # 1f <main+0x1f>
       1b: R X86 64 PC32  val-0x4
   1f: 8b 05 00 00 00 00 mov     0x0(%rip),%eax # 25 <main+0x25>
       21: R X86 64 PC32  val-0x4
   25: 89 c6      mov     %eax,%esi
   27: bf 00 00 00 00 mov     $0x0,%edi
       28: R X86 64 32  .rodata
   2c: b8 00 00 00 00 mov     $0x0,%eax
   31: e8 00 00 00 00 callq   36 <main+0x36>
       32: R X86 64 PC32  printf-0x4
   36: b8 00 00 00 00 mov     $0x0,%eax
   3b: c9        leaveq
   3c: c3        retq

```

objdump -dxs test 输出的部分内容如下（■是没有显示的隐藏内容）：

```

SYMBOL TABLE:
0000000000400400 l      d .text 0000000000000000      .text
00000000004005e0 l      d .rodata 0000000000000000      .rodata
0000000000601020 l      d .data 0000000000000000      .data
0000000000601040 l      d .bss 0000000000000000      .bss
0000000000000000 F *UND* 0000000000000000      printf@@GLIBC_2.2.5
0000000000601044 g      O .bss 0000000000000004      val
0000000000601030 g      O .data 0000000000000010      a
00000000004004e7 g      F .text 0000000000000039      sum
0000000000400400 g      F .text 000000000000002b      _start
0000000000400520 g      F .text 000000000000003d      main
Contents of section .rodata:
 4005e0 01000200 73756d3d 25640a00      ....sum=%d..
...
Contents of section .data:
 601020 00000000 00000000 00000000 00000000 .....
 601030 ffffffff feffffff 02000000 03000000 .....
...
00000000004003f0 <printf@plt>:
 4003f0: ff 25 22 0c 20 00 jmpq   *0x200c22(%rip) # 601018
<printf@GLIBC_2.2.5>

```



```

4003f6: 68 00 00 00 00    pushq $0x0
4003fb: e9 e0 ff ff ff    jmpq 4003e0 <.plt>
    
```

Disassembly of section .text:

0000000000400400 <_start>:

```

400400: 31 ed             xor %ebp,%ebp
    
```

....

00000000004004e7 <sum>:

```

4004e7: 55                push %rbp          #①
4004e8: 48 89 e5          mov %rsp,%rbp      #②
4004eb: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp) #③
4004f2: eb 1e            jmp 400512 <sum+0x2b>
4004f4: 8b 45 fc          mov -0x4(%rbp),%eax
4004f7: 48 98            cltq
4004f9: 8b 14 85 30 10 60 00 mov 0x601030(,%rax,4),%edx
400500: 8b 05 3e 0b 20 00 mov 0x200b3e(%rip),%eax #601044 <val>
400506: 01 d0            add %edx,%eax
400508: 89 05 36 0b 20 00 mov %eax,0x200b36(%rip) #601044 <val>
40050e: 83 45 fc 01      addl $0x1,-0x4(%rbp)
400512: 83 7d fc 03      cmpl $0x3,-0x4(%rbp) #④
400516: 7e dc            jle 4004f4 <sum+0xd> #⑤
400518: 8b 05 26 0b 20 00 mov 0x200b26(%rip),%eax # 601044 <val>
40051e: 5d              pop %rbp
40051f: c3              retq
    
```

0000000000400520 <main>:

```

400520: 55                push %rbp
400521: 48 89 e5          mov %rsp,%rbp
400524: 48 83 ec 10       sub $0x10,%rsp
400528: 89 7d fc          mov %edi,-0x4(%rbp)
40052b: 48 89 75 f0       mov %rsi,-0x10(%rbp)
40052f: b8 00 00 00 00    mov $0x0,%eax
400534: e8 ( ① )          callq 4004e7 <sum>
400539: 89 05 ( ② )       mov %eax, (,%rip) #601044<val>
40053f: 8b 05 ( ③ )       mov (,%rip),%eax #601044<val>
400545: 89 c6            mov %eax,%esi
400547: bf ( ④ )          mov (,%edi)
40054c: b8 00 00 00 00    mov $0x0,%eax
400551: e8 ( ⑤ )          callq 4003f0 <printf@plt>
400556: b8 00 00 00 00    mov $0x0,%eax
40055b: c9              leaveq
40055c: c3              retq
40055d: 0f 1f 00         nopl (%rax)
    
```

1. 阅读的 sum 函数反汇编结果中带下划线的汇编代码(编号①-⑤), 解释每行指令的功能和作用

2. 根据上述信息, 链接程序从目标文文件 test.o 和 main.o 生成可执行程序 test, 对 main 函数中空格①-⑤所在语句所引用符号的重定位结果是什么? 以 16 进制 4 字节数值填写这些空格, 将机器指令补充完整(写出任意 2 个即可)。

0000000000400520 <main>:

```

400520: 55          push  %rbp
400521: 48 89 e5    mov   %rsp,%rbp
400524: 48 83 ec 10  sub   $0x10,%rsp
400528: 89 7d fc    mov   %edi,-0x4(%rbp)
40052b: 48 89 75 f0  mov   %rsi,-0x10(%rbp)
40052f: b8 00 00 00 00 mov   $0x0,%eax
400534: e8( ① )    callq 4004e7 <sum>
400539: 89 05( ② )  mov   %eax,██████(%rip) #601044<val>
40053f: 8b 05( ③ )  mov   ██████(%rip),%eax #601044<val>
400545: 89 c6       mov   %eax,%esi
400547: bf ( ④ )    mov   ██████,%edi
40054c: b8 00 00 00 00 mov   $0x0,%eax
400551: e8 ( ⑤ )    callq 4003f0 <printf@plt>
400556: b8 00 00 00 00 mov   $0x0,%eax
40055b: c9         leaveq
40055c: c3         retq
40055d: 0f 1f 00    nopl  (%rax)
    
```

3. 在 sum 函数地址 4004f9 处的语句"mov 0x601030(,%rax,4),%edx"中,源操作数是什么类型、有效地址如何计算、对应 C 语言源程序中的什么量(或表达式)? 其中, rax 数值对应 C 语言源程序中的哪个量(或表达式)? 如何解释数字 4?

4-5 两个 C 语言程序 main2.c、addvec.c 如下所示:

| | |
|---|---|
| <pre> /* 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("z = [%d %d]\n", z[0], z[1]); return 0; } </pre> | <pre> /* 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 */ </pre> |
|---|---|

```
/* $end main2 */
```

用如下两条指令编译、链接，生成可执行程序 prog2：

```
gcc -m64 -no-pie -fno-PIC -c addvec.c main2.c
gcc -m64 -no-pie -fno-PIC -o prog2 addvec.o main2.o
```

运行指令 objdump -dxs main2.o 输出的部分内容如下：

Disassembly of section .text:

```
0000000000000000 <main>:
0: 48 83 ec 08          sub    $0x8,%rsp
4: b9 02 00 00 00      mov    $0x2,%ecx
9: ba 00 00 00 00      mov    $0x0,%edx
   a: R_X86_64_32      z
e: be 00 00 00 00      mov    $0x0,%esi
   f: R_X86_64_32      y
13: bf 00 00 00 00      mov    $0x0,%edi
   14: R_X86_64_32      x
18: e8 00 00 00 00      callq 1d <main+0x1d>
   19: R_X86_64_PC32     addvec-0x4
1d: 8b 0d 00 00 00 00      mov    0x0(%rip),%ecx    # 23 <main+0x23>
   1f: R_X86_64_PC32      z
23: 8b 15 00 00 00 00      mov    0x0(%rip),%edx    # 29 <main+0x29>
   25: R_X86_64_PC32      z-0x4
29: be 00 00 00 00      mov    $0x0,%esi
   2a: R_X86_64_32      .rodata.str1.1
2e: bf 01 00 00 00      mov    $0x1,%edi
33: b8 00 00 00 00      mov    $0x0,%eax
38: e8 00 00 00 00      callq 3d <main+0x3d>
   39: R_X86_64_PC32     __printf_chk-0x4
3d: b8 00 00 00 00      mov    $0x0,%eax
42: 48 83 c4 08          add    $0x8,%rsp
46: c3                  retq
```

objdump -dxs prog2 输出的部分内容如下（■是没有显示的隐藏内容）：

SYMBOL TABLE:

```
0000000000400238 l    d  .interp  0000000000000000          .interp
0000000000400254 l    d  .note.ABI-tag
0000000000000000 l    df *ABS* 0000000000000000          main2.c
0000000000601038 g    *ABS* 0000000000000000          _edata
000000000060103c g    O  .bss  0000000000000008          z
0000000000601030 g    O  .data 0000000000000008          x
0000000000000000 F *UND* 0000000000000000          addvec
0000000000601018 g    .data 0000000000000000          __data_start
00000000004007e0 g    O  .rodata 0000000000000004
_IO_stdin_used
0000000000601028 g    O  .data 0000000000000008          y
00000000004006f0 g    F  .text 0000000000000047          main
```

```
00000000004005c0 <addvec@plt>:
4005c0: ff 25 42 0a 20 00      jmpq   *0x200a42(%rip)    # 601008
```

```

        <_GLOBAL_OFFSET_TABLE_+0x20>
4005c6: 68 01 00 00 00      pushq  $0x1
4005cb: e9 d0 ff ff ff      jmpq   4005a0 <_init+0x18>

00000000004005d0 <__printf_chk@plt>:
4005d0: ff 25 3a 0a 20 00      jmpq   *0x200a3a(%rip)    # 601010

        <_GLOBAL_OFFSET_TABLE_+0x28>
        ....

00000000004006f0 <main>:
4006f0: 48 83 ec 08          sub    $0x8,%rsp
4006f4: b9 02 00 00 00      mov    $0x2,%ecx
4006f9: ba ① _ _ _ _      mov    ①,%edx
4006fe: be ② _ _ _ _      mov    ②,%esi
400703: bf ③ _ _ _ _      mov    ③,%edi
400708: e8 ④ _ _ _ _      callq  4005c0 <addvec@plt>
40070d: 8b 0d ⑤ _ _ _ _    mov    ⑤(%rip),%ecx #601040 <z+0x4>
400713: 8b 15 ⑥ _ _ _ _    mov    ⑥(%rip),%edx # 60103c <z>
400719: be e4 07 40 00      mov    $0x4007e4,%esi
40071e: bf 01 00 00 00      mov    $0x1,%edi
400723: b8 00 00 00 00      mov    $0x0,%eax
400728: e8 ⑦ _ _ _ _      callq  4005d0 <__printf_chk@plt>
40072d: b8 00 00 00 00      mov    $0x0,%eax
400732: 48 83 c4 08          add    $0x8,%rsp
400736: c3                  retq
    
```

4. 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. 根据上述信息,main 函数中空格①--⑦所在语句所引用符号的重定位结果是什么？以 16 进制 4 字节数值填写这些空格，将机器指令补充完整（写出任意 3 个即可）。（5 分）

| | |
|---------|---------|
| ② _____ | ② _____ |
| ③ _____ | ④ _____ |
| ⑤ _____ | ⑥ _____ |
| ⑦ _____ | |

6. 考虑下面的程序，它由两个模块组成：

| | |
|--------------------|----------------|
| /*main*/ | /*p1.c*/ |
| #include <stdio.h> | int x; |
| int x = 100; | int y; |
| int y; | void p1() |
| void p1(void); | { |
| int main() | x=1000; y=200; |

```
{
    int z=0;
    pl();
    y = 2000;
    printf("x=%d,y=%d\n",x,y);
    return 0;
}
```

请指出 main.o 中属于强符号的是? _____

程序最后的输出是什么? _____

第八章 异常控制流

一、选择题

1. 每个信号类型都有一个预定义的默认行为,可能是()
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
2. C 程序执行到整数或浮点变量除以 0 可能发生()
A.显示除法溢出错误直接退出 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
3. 进程 P1 是进程 P11 的父进程, P1 有全局变量 $x=0$, 下列说法错误的是()
A.P1 与 P11 有相同的地址空间
B.P1 与 P11 是并发执行的独立进程
C.若 P1 与 P11 均对 x 执行一次加 1 操作, 则 $x=2$
D.P1 与 P11 有相同的代码和数据段
4. 同步异常不包括()
A.终止 B.陷阱 C.停止 D.故障
5. 进程上下文切换不会发生在如下()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
6. 一个子进程终止或者停止时, 操作系统内核会发送()信号给父进程。
A. SIGKILL B. SIGQUIT C. SIGSTOP D. SIGCHLD
7. 内核为每个进程维持一个上下文, 不属于进程上下文的是()
A.寄存器 B.进程表 C.文件表 D.调度程序
8. Linux 进程终止的原因可能是()
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
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.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. 不属于同步异常的是()
A.中断 B.陷阱 C.故障 D.终止
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
12. 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.复位 D.故障
13. 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

14. 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. 键盘中断 B. 陷阱 C. 故障 D. 终止

15. 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

16. 下列不属于进程上下文的是 ()

A. 页全局目录 pgd B. 通用寄存器 C. 内核代码 D. 用户栈

17. 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

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

19. 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

20. 不属于进程上下文的是 ()

A. 页全局目录 pgd B. 内核栈 C. 内核代码 D. 打开的文件表

21. 下列函数中属于系统调用且调用一次，从不返回的是 ()

A. fork B. execve C. setjmp D. longjmp

22. 属于异步异常的是 ()

A. 中断 B. 陷阱 C. 故障 D. 终止

二、 填空题

1. 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，需要通过 _____ 进行实现。

2. 进程创建函数 fork 执行后返回 _____ 次。

3. 非本地跳转中的 setjmp 函数调用一次，返回 _____ 次。

4. 进程加载函数 execve，如调用成功则返回 _____ 次。

5. 子程序运行结束会向父进程发送 _____ 信号。

6. 向指定进程发送信号的 linux 命令是 _____。

7. CPU 在执行异常处理程序时其模式为 _____。

三、 判断题

1. () Linux 系统调用中的功能号 n 就是异常号 n。

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

3. () 进程一旦终止就不再占用内存资源。

4. () `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。
5. () 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。
6. () 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。
7. () 进程是并发执行的，所以能够并发执行的都是进程。
8. () 进程在进行上下文切换时一定会运行内核函数。

四、简答题

1. 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。

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

3. 简述 shell 的主要原理与过程

4. 在 shell 命令行输入命令：Ubuntu>./test-trans-M 32 -N 32[回车]。shell 命令行解释器将构造参数 `argv` 和 `envp`，请写出参数 `argv` 的内容。

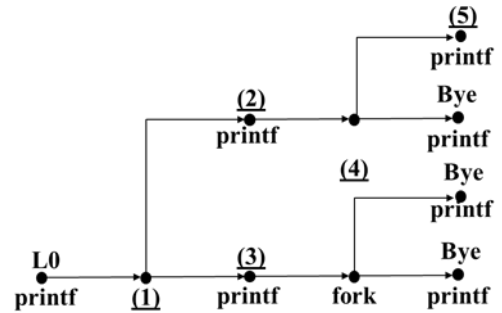
| | |
|----------------------|----------|
| ... | |
| <code>argv[5]</code> | -> _____ |
| <code>argv[4]</code> | -> _____ |
| <code>argv[3]</code> | -> _____ |
| <code>argv[2]</code> | -> _____ |
| <code>argv[1]</code> | -> _____ |
| <code>argv[0]</code> | -> _____ |

五、分析题

1. 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");
}
    
```



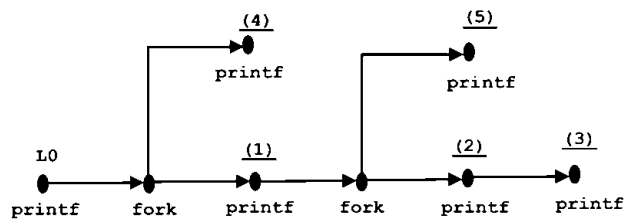
请写出上述进程图中空白处的内容

(1) _____ (2) _____ (3) _____ (4) _____ (5) _____

2. 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) _____ (2) _____ (3) _____ (4) _____ (5) _____

3. 一个C程序的main()函数如下:

```

int main() {
    if(fork() == 0) {
        printf("a"); fflush(stdout);
        exit(0);
    }
    else {
        printf("b"); fflush(stdout);
        waitpid(-1, NULL, 0);
    }
    printf("c"); fflush(stdout);
    exit(0);
}
    
```

(1) 请画出该程序的进程图

(2) 该程序运行后, 可能的输出数列是什么?

4. 设一个 C 语言源程序 p.c 编译链接后生成执行程序 p, 反汇编如下:

| C 程序 | 反汇编程序的 main 部分 (还有系统代码) 如下 |
|-------------------------|-------------------------------|
| #include <stdio.h> | 1 movw \$0x3ff, 0x80497d0 |
| unsigned short b[2500]; | 2 movw 0x804a324, %cx //k->cx |
| unsigned short k; | 3 mov \$0x801, %eax |
| void main(){ | 4 xorw %dx, %dx |
| b[1000] = 1023; | 5 div %ecx //2049/d |
| b[2000] = 2049*k; | 6 movw %dx, 0x804a324 |
| b[10000] = 20000; | 7 movw \$0x4e20, 0x804de20 |
| } | 8 ret |

现代 Intel 桌面系统, 采用虚拟页式存储管理, 每页 4KB, p 首次运行时系统中无其他进程。请结合进程与虚拟存储管理的知识, 分析上述程序的执行过程中:

- (1) 在取指令时发生的缺页异常次数为_____。
- (2) 写出已恢复的故障指令序号与故障类型_____。
- (3) 写出没有恢复的故障指令序号与故障类型_____。

5. 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);
}
```

6. 一段 C 语言程序如下:

```
#include "csapp.h"
int counter = 0;
jmp_buf buf;
void handler_alarm(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);
}
```

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：

`counter` 的数值是：

(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。

第九章 虚拟内存

一、选择题

1. Intel X86-64 的现代 CPU, 采用 () 级页表
A. 2 B. 3 C. 4 D. 由 BIOS 设置确定
2. 存储器垃圾回收时, 内存被视为一张有向图, 不能作为根结点的是 ()
A. 寄存器 B. 栈里的局部变量 C. 全局变量 D. 堆里的变量
3. "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
4. 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 ()
A. 程序代码和数据区 B. 栈 C. 共享库 D. 内核虚拟内存区
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
6. 某 CPU 使用 32 位虚拟地址和 4KB 大小的页时, 需要 PTE 的数量是 ()
A. 16 B. 8 C. 1M D. 512K
7. 动态内存分配时的块结构中, 关于填充字段的作用不可能的是 ()
A. 减少外部碎片 B. 满足对齐 C. 标识分配状态 D. 可选的
8. 虚拟内存系统中的虚拟地址与物理地址之间的关系是 ()
A. 1 对 1 B. 多对 1 C. 1 对多 D. 多对多
9. 虚拟内存发生缺页时, 缺页中断是由 () 触发
A. 内存 B. Cache L1 C. Cache L2 D. MMU
10. 当调用 malloc 这样的 C 标准库函数时, () 可以在运行时动态的扩展和收缩。
A. 堆 B. 栈 C. 共享库 D. 内核虚拟存储器
11. 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、载入物理内存
12. 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;
13. 虚拟内存发生缺页时, 正确的叙述是 ()
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
14. 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, 错误的叙述是 ()

- 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；
15. 某进程在成功执行函数 `malloc(24)` 后，下列说法正确的是（ ）
- A. 进程一定获得一个大小 24 字节的块
 - B. 进程一定获得一个大于 24 字节的块
 - C. 进程一定获得一个不小于 24 字节的块
 - D. 进程可能获得一个小于 24 字节的块
16. 虚拟页面的状态不可能是（ ）
- A. 未分配
 - B. 已分配未缓存
 - C. 已分配已缓存
 - D. 已缓存未分配
17. 动态内存分配时产生内部碎片的原因不包括（ ）
- A. 维护数据结构的开销
 - B. 满足对齐约束
 - C. 分配策略要求
 - D. 超出空闲块大小的分配请求

二、 填空题

1. C 语言函数中的整数常量都存放在程序虚拟地址空间的_____段。
2. TLB(翻译后备缓冲器)俗称快表，是_____的缓存。
3. 虚拟页面的状态有_____、已缓存、未缓存共 3 种
4. 虚拟内存发生缺页时，缺页中断是由_____触发的。
5. 虚拟内存系统借助_____这一数据结构将虚拟页映射到物理页。
6. Linux 虚拟内存区域可以映射到普通文件和_____，这两种类型的对象中的一种。
7. 程序运行时，指令中的立即操作数存放的内存段是：_____段。
8. 虚拟内存在内存映射时，映射到匿名文件的页面是_____的页。
9. 虚拟内存发生缺页时，MMU 将触发_____。

三、 判断题

1. () 在动态内存分配中，内部碎片不会降低内存利用率。
2. () 系统中当前运行进程能够分配的虚拟页面的总数取决于虚拟地址空间的大小。
3. () 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。
4. () 虚拟内存系统能有效工作的前提是软件系统具有局部性。
5. () 动态存储器分配时显式空闲链表比隐式空闲链表的实现节省空间。
6. () 动态内存隐式分配是指应用隐式地分配块并隐式地释放已分配块。
7. () 显式空闲链表的优点是在对堆块进行搜索时，搜索时间只与堆中的空闲块数量成正比。
8. () 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。
9. () 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。
10. () 隐式空闲链表的优点是在对堆块进行搜索时，搜索时间只与堆中的空闲块数量成正比。

四、 简答题

1. 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，

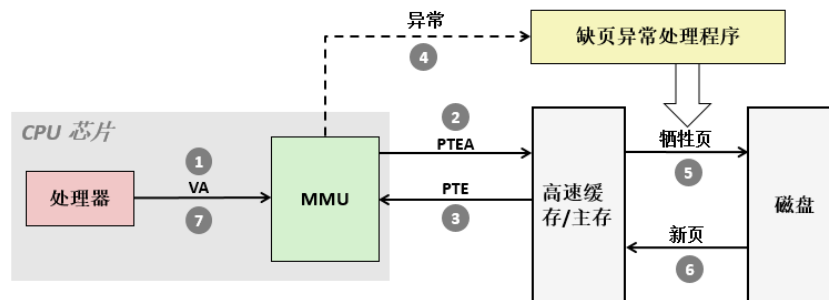
共 16 个条目；L1 数据 Cache 是物理寻址、直接映射，行大小为 4 字节，总共 16 个组。分析如下项目：

(1) 虚拟地址中的 VPN 占_____位；物理地址的 PPN 占_____位。

(2) TLB 的组索引位数 TLBI 为_____位。

(3) 用物理地址访问 L1 数据 Cache 时,Cache 的组索引 CI 占_____位,Cache 标记 CT 占_____位。

2. 结合下图，简述虚拟内存地址翻译的过程。

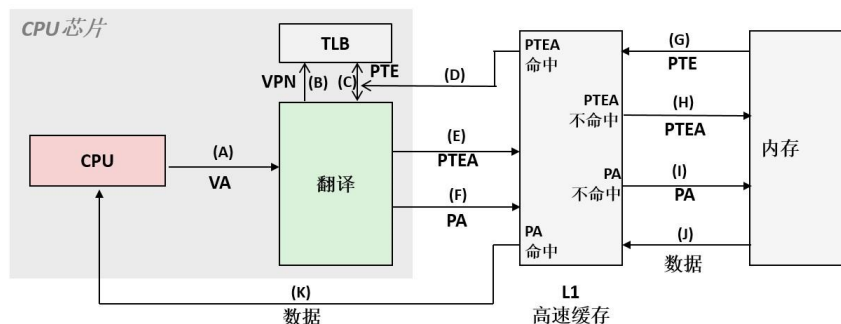


3. 下图展示了一个虚拟地址的访存过程，每个步骤采用不同的字母表示。请分别针对下述情况，用字母序列写出每种情况下的执行流程：

(1) TLB 命中、缓存 物理地址命中；

(2) TLB 不命中，缓存页表命中，缓存物理地址命中；

(3) TLB 不命中，缓存页表不命中，缓存物理地址不命中。



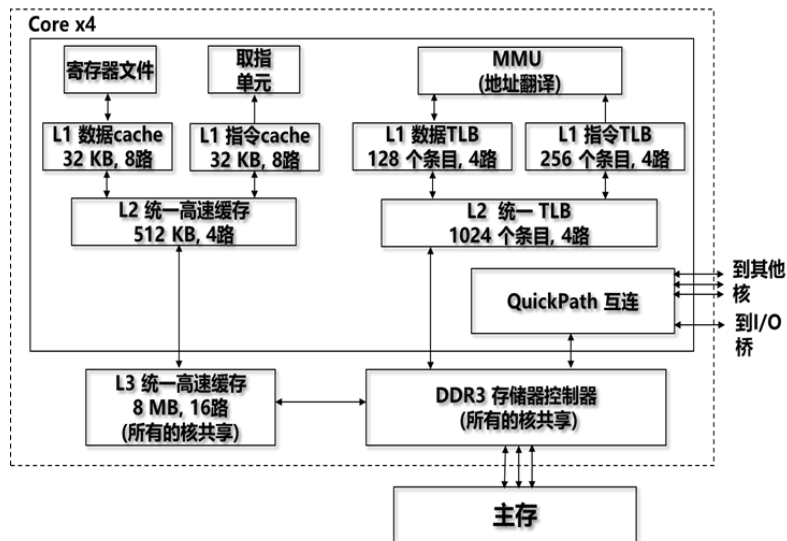
4. Intel I7 CPU 的虚拟地址 48 位，物理地址 52 位。其内部结构如下图所示，依据此结构，

每一页面 4KB，分析如下项目：

虚拟地址中的 VPN 占_____位；其一级页表为_____项。

L1 数据 TLB 的组索引位数 TLBI 为_____位，L1 数据 Cache 共_____组。

用物理地址访问 L1 数据 Cache 时，Cache 标记 CT 占_____位



五、 分析题

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

0x0000413433323110: 30 31 32 33 34 41 00 00 33 31 32 33 34 41 00 00

0x0000413433323120: 35 31 32 33 34 41 00 00 00 00 00 00 00 00 00 00

0x0000413433323130: 31 43 00 30 00 32 42 00 38 00 31 31 32 32 00 30

0x0000413433323140: 32 33 00 61 41 00 31 00 32 00 33 00 31 00 00 31

请写出程序名：_____，本程序的参数个数有_____个。

按顺序写出各个参数为_____、_____。

2. 在终端中的命令行运行显示“Hello World”的执行程序 hello，结合进程创建、加载、缺页中断、到存储访问（虚存）.....等等，论述 hello 是怎么一步步执行的。