ICS 问题求解

PKU 04832363：计算机系统导论讨论班

2021 年秋

北京大学
PEKING UNIVERSITY

编译于 2022-09-07 14:05

汇编：王畅
勘误：请写信到 wchang@pku.edu.cn

说明：以下内容是 2021 年秋计算机系统导论讨论班（04832363，班号 16）使用的材料。主要的练习题均来源于往年计算机系统导论课程，编者仅做了汇编和微小的改动。考题只选择了 2018 年及之前的内容，不选用更新的题目是因为 2019 年和 2020 年的试卷将留给大家复习备考时整套使用（个别除外）；另外质量较差的题目也未选入。根据本年度考试题型，不选入有关网络的大题。后面附有部分题目的分析和答案，“部分”选择的要旨是对较难的题目作详细解释；对于往年题我们只给出较难的题或者无答案题的分析。

目录

1 位级表示

要点

- ▷ 知道整数、浮点数在系统的存储方式（字节序）。
- ▷ 熟练掌握整数、浮点数的位级表示规则，快速完成其同十进制数的相互转换。
- ▷ 理解整数、浮点数规范中的各类特殊数的计算方法及其性质。
- ▷ 运用浮点数舍入的规则进行运算，知道类型转换的基本规则，能针对整数、浮点数的一些常见“反常情况”进行判断。

1. 在 x86-64 机器上，定义 `unsigned int A = 0x123456`。请画出 A 在内存中的存储方式：

…	低地址	A				高地址	…
…						…	

定义 `unsigned short B[2] = {0x1234, 0x5678}`。请画出 B 在内存中的存储方式：

…	低地址	B				高地址	…
…						…	

2. 在 x86-64 机器上，有下列 C 代码：

```
1   int main() {
2       unsigned int A = 0x11112222;
3       unsigned int B = 0x33336666;
4       void *x = (void *)&A;
5       void *y = 2 + (void *)&B;
6       unsigned short P = *(unsigned short *)x;
7       unsigned short Q = *(unsigned short *)y;
8       printf("0x%04x", P + Q);
9       return 0;
10  }
```

运行该代码，结果是什么？

3. 在 x86-64 机器上，有下列 C 代码：

```c
int main() {
    char A[12] = "11224455";
    char B[12] = "11445577";
    void *x = (void *)&A;
    void *y = 2 + (void *)&B;
    unsigned short P = *(unsigned short *)x;
    unsigned short Q = *(unsigned short *)y;
    printf("0x%04x", Q - P);
    return 0;
}
```

运行该代码，结果是什么？

4. 在 x86-64 机器上，有如下的定义：

```c
int x = ...; // 表达式 A
int y = ...; // 表达式 B
unsigned int ux = x;
unsigned int uy = y;
```

判断下表中的表达式是否等价：

序号	表达式 A	表达式 B
1	`x > y`	`ux > uy`
2	`(x > 0) || (x < ux)`	`1`
3	`x ^ y ^ x ^ y ^ x`	`x`
4	`((x >> 1) << 1) <= x`	`1`
5	`((x / 2) * 2) <= x`	`1`
6	`x ^ y ^ (~x) - y`	`y ^ x ^ (~y) - x`
7	`(x == 1) && (ux - 2 < 2)`	`(x == 1) && ((!!ux) - 2 < 2)`

提示 减法的运算优先级比按位异或高。布尔运算的结果都是有符号数。

5. 下列代码的目的是将字符串 A 的内容复制到字符串 B，覆盖 B 原有的内容，并输出“Hello World”；但实际运行输出是“Buggy Codes”。尝试找到代码中的错误。

```c
int main() {
    char A[12] = "Hello World";
    char B[12] = "Buggy Codes";
    int pos;
    for (pos = 0; pos - sizeof(B) < 0; pos++)
        B[pos] = A[pos];
    printf("%s\n", B);
}
```

6. 假设某浮点数格式为 1 位符号、3 位阶码、4 位小数。下表给出了用该格式表达的浮点数 $(-1)^S M \cdot 2^E$ 与其二进制表示的关系。完成下表。

描述	二进制表示	M（写成分数）	E	f
负零		/	/	-0.0
/	01000101			
最小的非规格化负数				
最大的规格化正数				
一				1.0
/				5.5
$+\infty$		/	/	/

7. 假设浮点数格式 A 为 1 位符号、3 位阶码、4 位小数，浮点数格式 B 为 1 位符号、4 位阶码、3 位小数。回答下列问题。

(1) 格式 A 中有多少个二进制表示对应于正无穷大?

(2) 考虑能精确表示的实数的最大绝对值。A 比 B 大还是比 B 小，还是两者一样?

(3) 考虑能精确表示的实数的最小非零绝对值。A 比 B 大还是比 B 小，还是两者一样?

(4) 考虑能精确表示的实数的个数。A 比 B 多还是比 B 少，还是两者一样?

8. 判断下列说法的正确性。

(1) 对于任意的单精度浮点数 `a` 和 `b`，如果 `a > b`，那么 `a + 1 > b`。

(2) 对于任意的单精度浮点数 `a` 和 `b`，如果 `a > b`，那么 `a + b > b + b`。

(3) 对于任意的单精度浮点数 `a` 和 `b`，如果 `a > b`，那么 `a + 1 > b + 1`。

(4) 对于任意的双精度浮点数 `d`，如果 `d < 0`，那么 `d * d > 0`。

(5) 对于任意的双精度浮点数 `d`，如果 `d < 0`，那么 `d * 2 < 0`。

(6) 对于任意的双精度浮点数 `d`，`d == d`。

(7) 将 `float` 转换成 `int` 时，既有可能造成舍入，又有可能造成溢出。

9. 遵循 IEEE 754 浮点数标准，考虑下列代码：

```
1  for (int x = 0; ; x++) {
2      float f = x;
3      if (x != (int)f) {
4          printf("%d", x);
5          break;
6      }
7  }
```

试问代码的运行结果是什么? 或者死循环?

10. 遵循 IEEE 754 浮点数标准，考虑下列代码：

```
int x = 33554466; // 2^25 + 34
int y = x + 8;
for ( ; x < y; x++) {
    float f = x;
    printf("%d ", x - (int)f);
}
```

写出程序的运行结果。

附：有关位运算的技巧

要点

- ▷ 了解位级操作中常见的小模块，包括 de Morgan 律、加法器、减法器、popcount 等，会运用常见策略处理非常规的位运算编程要求，如掩码设计、模拟、分治、位级表示展开等。
- ▷ 了解各种运算符的优先顺序。知道 ANSI C 与 C99 等标准的不同。初步了解整数在发生强制转换时的基本规则。
- ▷ 熟练运用 IEEE 754 标准，会使用该标准直接构造浮点数。
- ▷ 推荐的课外参考书：*Hacker's Delight* 和 *Matters Computational: Ideas, Algorithms, Source Code*，以及一个链接 https://graphics.stanford.edu/~seander/bithacks.html，里面介绍了一些位运算魔法。Datalab 和往年有一些比较过分的考题大量出自前一本书。因为 datalab 已经 due，故列出来供大家参考。

我们下面通过几个新例子来回顾 datalab 中用到的重要技巧，供大家回顾、反思和练习，以便于确保从 datalab 中学到了东西。当然，解法不唯一。

首先是大家已经熟练使用的掩码。它是指一串二进制数字，通过与目标数字的按位操作，达到屏蔽指定位而抽取信息的需求。例如，我们要取得某个二进制数 `x` 的最高位，可以使用 `(x >> 31) & 1`，这里 `1` 就是掩码。又如，获得 `x` 的所有偶数位，可以使用 `x & 0xcccccccc`，这里 `0xcccccccc` 就是掩码，等等。这段话中我们回顾了 `allOddBits` 的做法。

在第一个正式例子中，我们回顾 `bitNot, bitXor` 两个题的基本思路。

例 1.1 (de Morgan 律) 我们知道 `~, &, |` 可以表达“所有”的逻辑表达式，比方说异或 `x ^ y = (x & ~y) | (~x & y)`。而实际上 `~, |` 就足以做到这件事，因为我们有 `x & y = ~(~x | ~y)`。这样的情况称为**连接词的完备集**，以后大家还会反复学到。

当然，其实单纯一个与非或者一个或非也够了。假设**或非**用 ↓ 表示（即 `x ↓ y = ~(x | y)`），请你完成以下代码：唯一允许的位运算操作符是 ↓。

```
unsigned xor_with_nor(unsigned x, unsigned y) {
    return ____________________________; // return x ^ y with only NOR
}
```

在第二个例子中，我们总结和 `isLessOrEqual`, `sm2tc`, `counter1To5` 几个题有关的重要技巧。

例 1.2 (选择器) 试用位级运算的技术计算表达式 `cond ? t : f`。根据注释中的提示尝试补全下面的代码，假设 `cond` 的输入总是 1 或者 0。

```
int conditional(int cond, int t, int f) {
    /* Compute a mask that equals 0x00000000
     * or 0xFFFFFFFF depending on the value of cond */
    int mask = ____________________;
    /* Use the mask to toggle between returning t or returning f */
    return ____________________;
}
```

这个例子的意义是，如果你一定想要用 `if`，可以用这个办法避开 datalab 的限制。更显然地，减法、常数乘法都是事实上可以用的，此外你也可以回顾你在实验题中是怎样表达 == 的。

```
int equal(int x, int y) {
    return ____________________________; // return x == y
}

int minus(int x, int y) {
    return ____________________________; // return x - y
}
```

下面的例子则和 `fullSub`, `satAdd` 以及 `trueFiveEighths` 有关。

例 1.3 (加法器与减法器) 你可能以前听说过，如果要计算两个无符号数的平均 $\left\lfloor \frac{x+y}{2} \right\rfloor$ 而不发生溢出可以用：`(x & y) + ((x ^ y) >> 1)`。不难看出其原理：`x ^ y` 是半加（不考虑进位）的部分，不会发生溢出，右移即可；另一部分则包括进位，当且仅当两位均为 1 时发生，而如果将这个进位保留在原地，恰好就是除以 2 的效果。这样的思想来自于计算机的加法器。

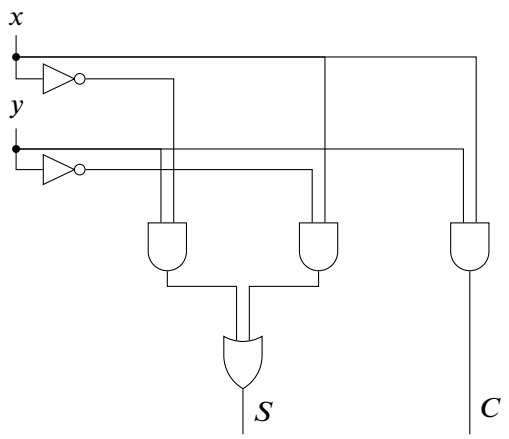


图 1.1: 半加器的结构，它计算两个一位整数的和 S 及其进位 C。可以看出 $S = x \oplus y, C = x\&y$。

假如我们要运算两个两位数相加，不难看出，只需要将第一位的进位 C 连接到下一位的加法中即可，即下一位是 $x \oplus y \oplus C$，以此类推。

如果要将加法改成减法，大家都很清楚怎么办，因为 `x - y = x + (~y + 1)`。

一般来说，有符号数的处理比无符号数稍微需要一些讨论。根据上面的方法，请你尝试给出两个有符号数的平均 $\left\lfloor \frac{x+y}{2} \right\rfloor$ 而不发生溢出的算法；对 $\left\lceil \frac{x+y}{2} \right\rceil$ 做同样的事。

与此相关，请问如何用位运算检查 $x+y$ 是否溢出了？ 〈提示〉检查符号位。

```
int addOK(int x, int y) {
    ____________________________________;
    ____________________________________;
    // You can add more lines
    return ____; // Determine if we can compute x + y without overflow
}
```

剩下的若干例子中，我们仔细回顾 `countConsecutive1` 和 `palindrome` 这两个比较难的问题需要用到的分治技巧。

例 1.4 (计数) 让我们来决定一个给定数的二进制表示中，1 的个数是偶数还是奇数；奇数返回 1，否则返回 0。允许使用 datalab 整数部分规定的所有运算符。

这里我们采用**分治**的策略，首先考察比较短的数。例如，当所处理的数只有两位时，采用的代码十分显然：

```
int bitParity2bit(int x) {
    int bit1 = 0b01 & x;
    int bit2 = 0b01 & (x >> 1);
    return bit1 ^ bit2;
}
```

将异或理解为 $\mathbb{F}_2$ 上的加法是非常有益的。

对于四位整数，我们两位两位操作，并让操作的过程某种意义上“并行”进行。首先分别确定 1、2 和 3、4 位的奇偶性，所得结果再设法异或一次。（思考：`mask2` 可以改成其他数吗？）

```
int bitParity4bit(int x) {
    int mask = 0b0101;
    int halfParity = (mask & x) ^ (mask & (x >> 1));
    int mask2 = 0b0011;
    return (mask2 & halfParity) ^ (mask2 & (halfParity >> 2));
}
```

现在，请你尝试根据上面的提示，补全下面针对八位整数的算法（要求使用操作符数目不超过 12 个）。

```
int bitParity8bit(int x) {
    int mask = ____________________;
    int quarterParity = ____________________;
    int mask2 = ____________________;
    int halfParity = ____________________;
    int mask3 = ____________________;
```

```
    return ____________________;
}
```

最后，试完成针对十六位或更长整数的算法（注意，根据 datalab 要求，立即数有效长度不能超过 8 位）。

作为模仿练习，请你对问题“决定一个给定数的二进制表示中有多少个 1”做同样的事。

```
/* Let's count how many bits are set in a number */
int bitCount8bit(int x) {
    int mask = ____________________;
    int quarterSum = ____________________;
    int mask2 = ____________________;
    int halfSum = ____________________;
    int mask3 = ____________________;
    return ____________________ + ____________________;
}
```

例 1.5 (模拟操作 2) 对于一个无符号整数，我们希望将它的位反过来（记为 rev(·)），例如 `0x01234567` 倒过来就是 `0xE6A2C480`。

这个某种意义上是一个广为流传的~~面试题~~。我们沿用**分治**的思想。假设二进制数 x 有 $2m$ 位 $(\underbrace{b_{2m-1}\cdots b_m}_{x_h}\underbrace{b_{m-1}\cdots b_0}_{x_l})_2$，那么显然有

$$\mathrm{rev}(x) = \mathrm{rev}(x_l)\,\mathrm{rev}(x_h).$$

根据这个思路，请你补全以下代码：

```
unsigned reverseBits(unsigned n) {
    n = (n >> 16) | (n << 16);
    n = ((n & ____________) >> 8) | ((n & ____________) << 8);
    n = ____________________________ | ____________________________;
    n = ____________________________________________________________;
    n = ____________________________________________________________;
    return n;
}
```

读过两个分治的操作之后，请思考：给一个整数，如何计算它有多少个前导零？允许的操作符包括所有的位级运算，以及加法和 !。 提示〉二分查找。

例 1.6 (算术运算) 考虑这样一个问题：对一个无符号整数 x, 计算 $x \bmod 5$。除了加法之外不能用其他算术操作。

方法上这很标准，假设 $x = (b_{31}\cdots b_1 b_0)_2$，通过计算 $2^i \pmod 5$ 的周期，我们知道

$$x \equiv \sum_{i=0}^{31} b_i 2^i \equiv 1b_0 + 2b_1 + 4b_2 + 3b_3 + 1b_4 + \cdots + 3b_{31} \pmod 5.$$

技术上，对一个四位无符号整数 $(b_3 b_2 b_1 b_0)_2$ 计算 $b_0 + 2b_1 + 4b_2 + 3b_3$ 可以非常直接：比如首先取得各位，然后用左移配合加法计算结果。你也可以思考有什么更省操作符数目的方法。

注意，当你需要求取 $x \bmod 2^i$ 时，总是可以简单使用 `x & ((1 << i) - 1)`。

2 位级表示—往年考题

1. (2018) 下列哪种类型转换既可能导致溢出，又可能导致舍入？

A. `int` 转 `float`

B. `float` 转 `int`

C. `int` 转 `double`

D. `float` 转 `double`

2. (2018) 在采用小端法存储机器上运行下面的代码，输出的结果是？（`int`、`unsigned` 为 32 位长，`short` 为 16 位长，0~9 的 ASCII 码是 `0x30~0x39`）

```
1    char *s = "2018";
2    int *p1 = (int *)s;
3    short s1 = (*p1) >> 12;
4    unsigned u1 = (unsigned) s1;
5    printf("0x%x\n", u1);
```

A. `0x00002303`

B. `0x00032303`

C. `0xffff8313`

D. `0x00008313`

3. (2018) 考虑如下函数

```
1    void XOR(int x, int y) {
2        y = x ^ y;
3        x = x ^ y;
4        y = x ^ y;
5        printf(x, y);
6    }
```

则 `XOR`(a, b) 的输出结果是什么？

A. a, b

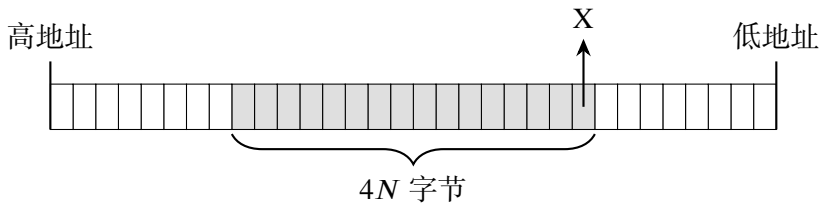
B. b, a

C. $b, 0$

D. $b, a \hat{} b$

4. (2017) 假定一个特殊设计的计算机，将 `int` 型数据的长度从 4 字节扩展为 $4N$ 字节，采用大

端法。现将该 int 型所能表示的最小负数写入内存中，如下图所示。其中每个小矩形代表一个字节，请问 X 位置这个字节中的值是多少？



A. 00000000
B. 01111111
C. 10000000
D. 11111111

5. (2017) 以下说法正确的是：
A. 负数加上负数结果都为负数
B. 正数加上正数结果都为正数
C. 用 & 和 ~ 可以表示所有的逻辑与或非操作
D. 用 & 和 | 可以表示所有的逻辑与或非操作

6. (2017, 2016) 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，则最小的规格化正数是：
A. $(1-2^{-n})\cdot 2^{-2^{k-1}+2}$
B. $2^{-2^{k-1}+2}$
C. $2^{-n}\cdot 2^{-2^{k-1}+2}$
D. $(1-2^{-n})\cdot 2^{-2^{k}+1}$

7. (2016) 假定编译器规定 int 和 short 型长度分别为 32 位和 16 位，执行下列语句：`unsigned short x = 65530; unsigned int y = x;`，得到 y 的机器数是 _____。（用 16 进制表示，勿省略前导的 0）

8. (2016) 一个 C 语言程序在一台 32 位机器上运行。程序中定义了三个变量 x, y, z，其中 x 和 z 为 int 型，y 为 short 型。当 $x = 127, y = -9$ 时，执行赋值语句 $z = x + y$ 后，z 的值是 _____。（用 16 进制表示，勿省略前导的 0）

9. (2016) 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 进制表示）

10. (2015) 给定一个实数，会因为该实数表示成单精度浮点数而发生误差。不考虑 NaN 和 Inf 的情况，该绝对误差的最大值为：
A. 2^{103}
B. 2^{104}
C. 2^{230}
D. 2^{231}

11. (2016) 现有一个二进制浮点的表示规则，其中 E 为指数部分（3 比特），bias 为 3；M 为小数部分（5 比特），采用二进制补码表示形式，且取值 $0.5 \le |M| < 1$，s 是浮点的符号位。该形式包含一个值为 1 的隐藏位。问 $+5_{10}$ 在该表示下的值是下列哪一个？

A. `010001100`
B. `010100100`
C. `011011010`
D. `011110101`

12. (2015) 关于浮点数，以下说法正确的是：

A. 给定任意浮点数 a, b, x，如果 $a > b$ 成立 (指求值为 1)，则一定有 $a + x > b + x$ 成立
B. 给定任意浮点数 a, b, x，如果 $a > b$ 不成立 (指求值为 0)，则一定有 $a + x > b + x$ 不成立
C. 不考虑结果为 `NaN, Inf` 或运算过程发生溢出的情况，高精度浮点数一定得到比低精度浮点数更精确或相同的结果
D. 不考虑结果为 `NaN, Inf` 的情况，高精度浮点数一定得到比低精度浮点数更精确或相同的结果

13. (2015) 在 32 位平台上，按 C90 标准判定以下语句中结果为假的是：

A. `return INT_MIN < INT_MAX;`
B. `return -2147483648 < 2147483647;`
C. `int a = -2147483648; return a < 2147483647;`
D. `return -2147483647 - 1 < 2147483647;`

提示〉C90 中的类型转换顺序：`int -> long -> unsigned -> unsigned long`，$2^{31} = 2147483648$。

14. (2014) 设整数均为 32 位，其中 `unsigned x = 0x00000001; int y = 0x80000000; int z = 0x80000001;`。以下表达式求值为 1 的是：

A. `(-1) < x`
B. `(-y) > -1`
C. `~y + y == -1`
D. `(z << 4) > (z * 16)`

15. (2014) 下面说法正确的是：

A. 数 0 的反码表示是唯一的
B. 数 0 的补码表示不是唯一的
C. `1000_1111_1110_1111_1100_0000_0000_0000` 表示的唯一整数是 `0x8FEFC000`
D. `1000_1111_1110_1111_1100_0000_0000_0000` 如果是单精度浮点数，则其值是 $-(1.110111111)_2 \cdot 2^{31-127}$

16. (2016, 2014) 下面表达式中为真的是：

A. `(unsigned)-1 < -2`
B. `2147483647 > (int)2147483648U`
C. `(0x80005942 >> 4) == 0x09005942`
D. `2147483647 + 1 != 2147483648`

17. (2014) 下面关于 IEEE 浮点数标准说法正确的是哪个？

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. 若甲类浮点数有 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. 若甲类浮点数有 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。

18. (2014) 假设有下面 x 和 y 的程序定义：

```
1    int x = a >> 2;
2    int y = (x + a) / 4;
```

那么有 _____ 个位于闭区间 $[-8, 8]$ 内的整数 a 能使得 x 和 y 相等。

19. (2013) 对于 IEEE 浮点数，如果减少 1 位指数位，将其用于小数部分，下列叙述正确的是哪个？

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。

20. (2017) 考虑有一种基于 IEEE 浮点格式的 9 位浮点表示格式 A。格式 A 有 1 个符号位、k 个阶码位、n 个小数位。现在已知 −9 的位模式可以表示为 101100010。回答以下问题。（注：阶码偏移量为 $2^{k-1}-1$）

(1) 求 k 和 n 的值。

(2) 基于格式 A，请填写下表。值的表示可以写成整数（如 16），或者写成分数（如 17/64）。

描述	二进制表示	值
最大的非规格化数		
最小的正规格化数		
最大的规格化数		

(3) 假设格式 A 变为 1 个符号位、$k+1$ 个阶码位、$n-1$ 个小数位，那么能表示的实数数量会怎样变化？数值的精度会怎样变化？（回答增加、降低或不变即可）

21. (2016) 在 64 位机器上，判断表达式是否对代码生成的变量恒成立（指求值为 1）。

```
1    /* random_int() 函数返回一个随机的 int 类型值 */
2    int x = random_int();
3    int y = random_int();
4    int z = random_int();
5    unsigned ux = (unsigned)x;
6    long lx = (long)x; /* long 为 64 位 */
7    long ly = (long)y;
8    double dx = (double)x;
```

```
    double dy = (double)y;
    double dz = (double)z;
```

(1) `(x >= 0) || (3 * x < 0)`

(2) `(x >= 0) || (x < ux)`

(3) `((x >> 1) << 1) <= x`

(4) `((x - y) << 3) + (x >> 1) - y == 8 * x - 9 * y + x / 2`

(5) `(x - y > 0) == ((y + ~x + 1) >> 31 == 1)`

(6) `dx + dy == (double) (y + x)`

(7) `dx + dy + dz == dz + dy + dx`

(8) `(int)((lx + ly) >> 1) == ((x & y) + ((x ^ y) >> 1))`

22. (2016) 假设 C 语言中新定义了一种数据类型 T, 该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：符号位 (s) 1-bit、阶码字段 (exp) 6-bit、小数字段 (frac) 5-bit。

(1) 若将该格式下能表示的所有正规格数从小到大依次排列，则相邻两数之间差值的最小值为 _____，最大值为 _____。

(2) 现定义了如下变量：

```
    T a = -15.875;
    T b = (1 << 28) + (1 << 24) + (1 << 22);
    T c = a * b;
```

给出各变量的二进制表示。

23. (2015) 对于下面的每一个表达式，请选择以下选项中的**一个或多个**，使得该表达式恒成立，如果没有满足条件的选项则填写 none：A. `<`　B. `>`　C. `==`　D. `!=`　E. none。

题目中出现的变量定义如下（浮点数保证不是 `NaN` 或者 `Inf`）：`int x, y; unsigned ux = x; double d;`。

(1) 如果 `x > 0`，则 `x + 1` _____ `0`

(2) 如果 `x > y`，则 `ux` _____ `y`

(3) 如果 `((x << 31) >> 31) < 0`，则 `x & 1` _____ `0`

(4) 如果 `((unsigned char)x >> 1) < 64`，则 `(char)x` _____ `0`

(5) 如果 `d < 0`，则 `d * 2` _____ `0`

(6) 如果 `d < 0`，则 `d * d` _____ `0`

(7) `x ^ y ^ (~x) - y` _____ `y ^ x ^ (~y) - x`

(8) `(((!!ux)) << 31) >> 31)` _____ `(((!!x) << 31) >> 31)`

24. (2015) 考虑一种 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，浮点数的字段划分如下：符号位 (s) 1-bit、阶码字段 (exp) 4-bit、小数字段 (frac) 7-bit。回答下列问题。

(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：$[1,2);[2,3)$。

(2) 填写下表。

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

25. (2014) 回答下列问题。

(1) 假设下列 `unsigned` 和 `int` 数均为 5 位（有符号整型用补码运算表示）：`int y = -7; unsigned z = y;`。确定 `y, z` 和最小有符号整数的十进制表示和二进制表示。

(2) 按照 IEEE 浮点数标准，首先将下列两个数表示 $(-1)^s M \cdot 2^E$ 的形式，然后写出其二进制表示：$0.375, -12.5$。

3 汇编语言

要点

- ▷ 知道 x86-64 中指令、程序计数器、寄存器、条件码、内存等概念，记住重要的寄存器名称、含义和使用规范，清楚掌握数据传送指令和操作数的正确用法。
- ▷ 掌握条件分支、条件传送、各种循环、跳转表的翻译方式，能熟练地在汇编语言中识别控制流结构，快速完成机器码、汇编语言、C 代码的相互转换。
- ▷ 理解 x86-64 系统栈空间的分布和管理方式，知道过程调用中的重要寄存器和相关保存指令，会复述过程调用的整个过程并绘制栈空间的变化情况。
- ▷ 知道结构体、联合体在内存中的存储情况，掌握用对齐规则访问结构体和联合体，以及计算其实际大小的方法。会处理复杂的指针和函数指针问题。

1. 判断下列 x86-64 ATT 操作数格式是否合法。

(1) `8(%rax, , 2)`

(2) `$30(%rax, %rax, 2)`

(3) `0x30`

(4) `13(, %rdi, 4)`

(5) `(%rsi, %rdi, 6)`

(6) `%ecx`

(7) `(%ecx)`

(8) `(%rbp, %rsp)`

2. 假设 `%rax`、`%rbx` 的初始值都是 0。根据下列一段汇编代码，写出每执行一步后两个寄存器的值。

```
movabsq   $0x0123456789ABCDEF, %rax
movw      %ax, %bx
movswq    %bx, %rbx
movl      %ebx, %eax
movabsq   $0x0123456789ABCDEF, %rax
cltq
```

3. 下列操作不等价的是：

A. `movzbq` 和 `movzbl`

B. `movzwq` 和 `movzwl`

C. `movl` 和 `movslq`

D. `movslq %eax, %rax` 和 `cltq`

4. 判断下列 x86-64 ATT 数据传送指令是否合法。

(1) `movl $0x400010, $0x800010`

(2) `movl $0x400010, 0x800010`

(3) `movl 0x400010, 0x800010`

(4) `movq $-4, (%rsp)`

(5) `movq $0x123456789AB, %rax`

(6) `movabsq $0x123456789AB, %rdi`

(7) `movabsq $0x123456789AB, 16(%rcx)`

(8) `movq 8(%rsp), %rip`

5. 在 32 位机器中有如下定义 `int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};`。

某一时刻，`%ecx` 存着第一个元素的地址，`%ebx` 值为 3，那么下列操作中，哪一个将 `array[3]` 移入了 `%eax`?

A. `leal 12(%ecx), %eax`

B. `leal (%ecx, %ebx, 4), %eax`

C. `movl (%ecx, %ebx, 4), %eax`

D. `movl 8(%ecx, %ebx, 2), %eax`

6. 下面的汇编代码对应一个 C 函数，其原型为 `long func(long a, long b);`。将其翻译为 C 代码。

```
// a in %rdi, b in %rsi
func:
    movq   %rdi, %rax
    salq   $4, %rax
    subq   %rdi, %rax
    movq   %rax, %rdi
    leaq   0(, %rsi, 8), %rax
    subq   %rsi, %rax
    addq   %rdi, %rax
    ret
```

7. 指令 `setg %al` 会让寄存器 `%al` 得到：

A. `~(SF ^ OF) & ~ZF`

B. `~(SF | OF) & ~ZF`

C. `~(SF | OF)`

D. `~(SF ^ OF)`

8. 下面的汇编代码对应一个 C 函数，其原型为 `long func(long a, long b);`。将其翻译为 C 代码。

```
// a in %rdi, b in %rsi
func:
    movl    $1, %eax
    jmp     .L2
.L4:
    testb   $1, %sil
    je      .L3
    imulq   %rdi, %rax
.L3:
    sarq    %rsi
    imulq   %rdi, %rdi
.L2:
    testq   %rsi, %rsi
    jg      .L4
    rep ret
```

9. 对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

(1)
```
long f1(long a, long b) { return (++a > --b) ? a : b; }
```

(2)
```
long f2(long a, long b) { return (*a > *b) ? --(*a) : (*b)--; }
```

(3)
```
long f3(long a, long b) { return a ? *a : (b ? *b : 0); }
```

(4)
```
long f4(long a, long b) { return (a > b) ? a++ : ++b; }
```

10. 根据下面的汇编指令补充机器码中缺失的字节。

机器码	汇编指令
loop:	
4004d0: 48 89 f8	mov %rdi, %rax
4004d3: eb _____	jmp 4004d8 <loop+0x8>
4004d5: 48 d1 f8	sar %rax
4004d8: 48 85 c0	test %rax, %rax
4004db: 7f _____	jg 4004d5 <loop+0x5>
4004dd: f3 c3	repz retq

11. 使用 gdb 查看某个可执行文件，发现其一段内存为：

```
0x400598: 0x0000000000400488 0x0000000000400488
0x4005a8: 0x000000000040048b 0x0000000000400493
```

```
0x4005b8: 0x000000000040049a 0x0000000000400482
0x4005c8: 0x000000000040049a 0x0000000000400498
```

根据以下汇编代码，

```
0x400474: cmp   $0x7, %edi
0x400477: ja    0x40049a
0x400479: mov   %edi, %edi
0x40047b: jmpq  *0x400598(, %rdi, 8)
0x400482: mov   $0x15213, %eax
0x400487: retq
0x400488: sub   $0x5, %edx
0x40048b: lea   0x0(, %rdx, 4), %eax
0x400492: retq
0x400493: mov   $0x2, %edx
0x400498: and   %edx, %esi
0x40049a: lea   0x4(%rsi), %eax
0x40049d: retq
```

补全主函数的 C 代码。

```c
// a in %rdi, b in %rsi, c in %rdx
int main(int a, int b, int c) {
    int res = 4;
    switch (a) {
        case 0:
        case 1:
            __________;
        case ___:
            res = __________;
            break;
        case ___:
            res = __________;
            break;
        case 3:
            __________;
        case 7:
            __________;
        default:
            __________;
    }
    return res;
}
```

12. 将下列汇编代码翻译成 C 代码。

```
func:
    movq    %rsi, %rax
```

```
    testq   %rdi, %rdi
    jne     .L7
    rep ret
.L7:
    subq    $8, %rsp
    imulq   %rdi, %rax
    movq    %rax, %rsi
    subq    $1, %rdi
    call    func
    addq    $8, %rsp
    ret
```

```
long func(long n, long m) {
    if (__________)
        return __________;
    return func(__________, __________);
}
```

13. 将下列 C 代码翻译为汇编代码。

```
void callee(long *a, long *b) {
    if (a == b) return;
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
void caller(long n, long arr[]) {
    for (long i = 0; i < n/2; i++)
        callee(&arr[i], &arr[n-i]);
}
```

汇编代码为:

```
callee:
    cmpq    %rsi, %rdi
    je      .L1
    movq    (%rsi), %rax
    xorq    (%rdi), %rax
    movq    ____________
    xorq    (%rsi), %rax
    movq    %rax, (%rsi)
    xorq    %rax, (%rdi)
.L1:
    rep ret

caller:
    pushq   %r12
```

```
        pushq   %rbp
        pushq   %rbx
        movq    %rdi, %rbp
        movq    %rsi, %r12
        movl    $0, %ebx
        jmp     .L4
    .L5:
        movq    %rbp, %rax
        subq    %rbx, %rax
        ____    (%r12, %rax, ____), %rsi
        ____    (%r12, %rbx, ____), %rdi
        call    callee
        addq    $1, %rbx
    .L4:
        movq    %rbp, %rax
        shrq    $63, %rax
        addq    %rbp, %rax
        sarq    %rax
        cmpq    %rbx, %rax
        jg      .L5
        popq    ____
        popq    ____
        popq    ____
        ret
```

对于上面代码，在 x86-64、操作系统为 Linux 的情况下，假设 main 在 `0x4000ac` 处调用 caller，caller 在 `0x400088` 处调用 callee；调用函数（`call xx`）的代码长度为 5。在 main 即将调用 caller 时，部分寄存器的情况见下表左侧。请在下图右侧画出控制流第一次走到 `.L1` 时，堆栈的结构。

寄存器	调用前的值	地址	内容（不确定的空格不用填）
%rsp	0xffffffffff80	0xf...f88~8f	
%rax	0x0	0xf...f80~87	
%rbx	0x15	0xf...f78~7f	
%rbp	0x18	0xf...f70~77	
%r12	0x213	0xf...f68~6f	
%rsi	0x0	0xf...f60~67	
%rdi	0x0	0xf...f58~5f	
		0xf...f50~57	

14. 在 x86-64、Linux 操作系统下有如下 C 定义：

```
1   struct A {
2       char CC1[6];
3       int II1;
```

```
        long LL1;
        char CC2[10];
        long LL2;
        int II2;
    };
```

(1) sizeof(A) = _____。

(2) 将 A 重排后，令结构体尽可能小，那么得到的新的结构体大小为 _____ 字节。

15. 在 x86-64、LINUX 操作系统下，考虑如下的 C 定义：

```
typedef union {
    char c[7];
    short h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    int i;
} struct_e;

struct_e s;
```

(1) s.u.c 的首地址相对于 s 的首地址的偏移量是 _____ 字节。

(2) sizeof(union_e) = _____ 字节。

(3) s.i 的首地址相对于 s 的首地址的偏移量是 _____ 字节。

(4) sizeof(struct_e) = _____ 字节。

(5) 若只将 i 的类型改成 short，那么 sizeof(struct_e) = _____ 字节。

(6) 若只将 h 的类型改成 int，那么 sizeof(union_e) = _____ 字节。

(7) 若将 i 的类型改成 short、h 的类型改成 int，则 sizeof(union_e) = _____ 字节，sizeof(struct_e) = _____ 字节。

(8) 若只将 short h 的定义删除，那么 (1)~(4) 问的答案分别是 ____、____、____、____ 字节。

16. 以下提供了一段代码的 C 语言、汇编语言以及运行到某一时刻栈的情况。

```
0000000000400596 <func>:
    400596: sub    $0x28, %rsp
    40059a: mov    %fs:0x28, %rax
    4005a3: mov    %rax, 0x18(%rsp)
    4005a8: xor    %eax, %eax
    4005aa: mov    (%rdi), %rax
    4005ad: mov    0x8(%rdi), %rdx
```

```
    4005b1: cmp    %rdx, %rax
    4005b4: jge    _____(1)_____
    4005b6: mov    %rdx, (%rdi)
    4005b9: mov    %rax, 0x8(%rdi)
    4005bd: mov    0x8(%rdi), %rax
    4005c1: test   %rax, %rax
    4005c4: jne    4005cb <func+0x35>
    4005c6: mov    (%rdi), %rax
    4005c9: jmp    _____(2)_____
    4005cb: mov    (%rdi), %rdx
    4005ce: sub    %rax, %rdx
    4005d1: mov    %rdx, (%rsp)
    4005d5: mov    %rax, 0x8(%rsp)
    4005da: mov    _____(3)_____, %rdi
    4005dd: callq  400596 <func>
    4005e2: mov    0x18(%rsp), %rcx
    4005e7: xor    _____(4)_____, %rcx
    4005f0: _____(5)_____    4005f7 <func+0x61>
    4005f2: callq  400460 <__stack_chk_fail@plt>
    4005f7: add    _____(6)_____, %rsp
    4005fb: retq

00000000004005fc <main>:
    4005fc: sub    $0x28, %rsp
    400600: mov    %fs:0x28, %rax
    400609: mov    %rax, 0x18(%rsp)
    40060e: xor    %eax, %eax
    400610: movq   0x69, (%rsp)
    400618: movq   0xfc, 0x8(%rsp)
    400621: mov    %rsp, %rdi
    400624: callq  400596 <func>
    400629: mov    %rax, %rsi
    40062c: mov    $0x4006e4, %edi
    400631: mov    $0x0, %eax
    400636: callq  400470 <printf@plt>
    40063b: mov    0x18(%rsp), %rdx
    400640: xor    _____(4)_____, %rdx
    400649: _____(5)_____    400650 <main+0x54>
    40064b: callq  400460 <__stack_chk_fail@plt>
    400650: mov    $0x0, %eax
    400655: add    _____(6)_____, %rsp
    400659: retq
```

C 语言代码：

```
1    typedef struct{
2        long a;
```

```
3         long b;
4     } pair_type;
5
6     long func(pair_type *p) {
7         if (p -> a < p -> b) {
8             long temp = p -> a;
9             p -> a = p -> b;
10            p->b = temp;
11        }
12        if (_____(7)_____) {
13            return p->a;
14        }
15        pair_type np;
16        np.a = _____(8)_____;
17        np.b = _____(9)_____;
18        return func(&np);
19    }
20
21    int main(int argc, char* argv[]) {
22        pair_type np;
23        np.a = _____(10)_____;
24        np.b = _____(11)_____;
25        printf("%ld", func(&np));
26    }
```

堆栈情况如下（从高地址向低地址列举）：

1. 0x0000000000000000
2. 0xc76d5add7bbeaa00
3. 0x00007fffffffdf60
4. ____________________
5. ____________________
6. 0x0000000000400629
7. ____________________
8. ____________________
9. 0x0000000000000001
10. 0x0000000000000069
11. 0x0000000000000093
12. ____________________
13. 0x00000000ff000000
14. ____________________
15. 0x0000000000000000
16. ____________________
17. ____________________
18. ____________________

19. 0x0000000000000000
20. _____
21. _____
22. 0x000000000000002a
23. 0x000000000000003f
24. 0x00000000004005e2

一些可能用到的字符的 ASCII 码如下：

换行	空格	"	%	(	)	,	0	A	a
0x0a	0x20	0x22	0x25	0x28	0x29	0x2c	0x30	0x41	0x61

回答下列问题。

(1) Gdb 下使用命令 `x/4b 0x4006e4` 后（即查看 `0x4006e4` 开始的 4 个字节，用 16 进制表示）得到的输出结果是 _____。

(2) 互相翻译 C 语言代码和汇编代码，补充缺失的空格（标号相同的为同一格）。

(3) 补充栈的内容。使用 16 进制，可以不写前导多余的 0；对于给定已知条件后仍无法确定的值，填写“不确定”；已知程序运行过程中寄存器 `%fs` 的值没有改变。

(4) 程序运行结果是 _____。

4 汇编语言—往年考题

1. (2018) 在 x86-64 下，以下哪个选项的说法是错误的?

A. `movl` 指令以寄存器作为目的时，会将该寄存器的高位 4 字节设置为 0

B. `cltq` 指令的作用是将 `%eax` 符号扩展到 `%rax`

C. `movabsq` 指令只能以寄存器作为目的

D. `movswq` 指令的作用是将零扩展的字传送到四字节目的

2. (2018) 下列关于程序控制结构的机器代码实现的说法中，正确的是：

A. 使用条件跳转语句实现的程序片段比使用条件赋值语句实现的同一程序片段的运行效率高

B. 使用条件跳转语句实现的程序片段与使用条件赋值语句实现的同一程序片段虽然效率可能不同，但在 C 语言的层面上看总是有着相同的行为

C. 一些 `switch` 语句不会被 gcc 用跳转表的方式实现

D. 以上说法都不正确

3. (2018) 下列关于条件码的叙述中，不正确的是：

A. 所有算术指令都会改变条件码

B. 所有比较指令都会改变条件码

C. 所有与数据传送有关的指令都会改变条件码

D. 条件码一般不会直接读取，但可以直接修改

4. (2017) 在下列的 x86-64 汇编代码中，错误的是：

A. `movq %rax, (%rsp)`

B. `movl $0xFF, (%ebx)`

C. `movsbl (%rdi), %eax`

D. `leaq (%rdx, 1), %rdx`

5. (2017) 在下列关于条件传送的说法中，正确的是：

A. 条件传送可以用来传送字节、字、双字和四字的数据

B. C 语言中的“`?:`”条件表达式都可以编译成条件传送

C. 使用条件传送总可以提高代码的执行效率

D. 条件传送指令不需要用后缀 (例如 `b`, `w`, `l`, `q`) 来表明操作数的长度

6. (2017) 在下列指令中，其执行会影响条件码中的 CF 位的是：

A. `jmp NEXT`

B. `jc NEXT`
C. `inc %bx`
D. `shl $1, %ax`

7. (2016) 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

8. (2016) 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`

9. (2016) 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. (2016) 将 `%ax` 清零，下列指令不能实现该效果的是：
A. `sub %ax, %ax`
B. `xor %ax, %ax`
C. `test %ax, %ax`
D. `and $0, %ax`

11. (2016) 在如下 `switch` 语句翻译得到的跳转表中，哪些标号没有出现在分支中？

```
addq    $1, %rdi
cmpq    $8, %rdi
ja      .L2
jmp     *.L4(, %rdi, 8)
.L4:
    .quad .L9
    .quad .L5
    .quad .L6
    .quad. L7
    .quad .L2
    .quad .L7
```

```
        .quad .L8
        .quad .L2
        .quad .L5
```

A. 3, 6
B. −1, 4
C. 0, 7
D. 2, 4

12. (2016) 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. (2015) 下列寻址模式中，正确的是：

A. `(%eax, , 4)`
B. `(%eax, %esp, 3)`
C. `123`
D. `$1(%ebx, %ebp, 1)`

14. (2015) 假设某条 C 语言 `switch` 语句编译后产生了如下的汇编代码及跳转表：

```
    movl   8(%ebp), %eax
    subl   $48, %eax
    cmpl   $8, %eax
    ja     .L2
    jmp    *.L7(, %eax, 4)
    .L7:
        .long .L3
        .long .L2
        .long .L2
        .long .L5
        .long .L4
        .long .L5
        .long .L6
        .long .L2
        .long .L3
```

在源程序中，下面的哪些标号出现过？

A. `'2', '7'`
B. `1`
C. `'3'`
D. `5`

15. (2015, 2014) 下列的指令组中，哪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. `LEAL, CMP`

16. (2014) 简单的 `switch` 语句常采用跳转表的方式实现，在 x86-64 系统中，下述最有可能是正确的 `switch` 分支跳转汇编指令的是哪个?

A. `jmp .L3(, %eax, 4)`
B. `jmp .L3(, %eax, 8)`
C. `jmp *.L3(, %eax, 4)`
D. `jmp *.L3(, %eax, 8)`

17. (2018) 以下代码的输出结果不可能是

```
union {
    double d;
    struct {
        int i;
        char c[4];
    } s;
} u;
u.d = 1;
printf("%d\n", u.s.c[2]);
```

A. 0
B. −16
C. 240
D. 191

18. (2018) 下列关于 C 语言中的结构体（struct）以及联合（union）的说法中，正确的是：

A. 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列不一定会使之内存占用最小
B. 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列一定不会使之内存占用最大
C. 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列不一定会使之内存占用最小
D. 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列一定不会使之内存占用最大

19. (2017) 有如下代码段：

```
int func(int x, int y);
int (*p) (int a, int b);
p = func;
```

```
4     p(0, 0);
```

对应的下列 x86-64 过程调用正确的是：

A. `call *%rax`
B. `call *(%rax)`
C. `call (%rax)`
D. `call func`

20. (2017) 有定义：`int A[3][2] = {{1,2}, {3,3}, {2,1}};`，则 `A[2]` 是：

A. `&A + 16`
B. `A + 16`
C. `*A + 4`
D. `*A + 2`

21. (2015) 已知下面的数据结构，假设在 Linux/IA32 下要求对齐，这个结构的总的大小是多少个字节？如果重新排列其中的字段，最少可以达到多少个字节？

```
1     struct {
2         char a;
3         double *b;
4         double c;
5         short d;
6         long long e;
7         short f;
8     };
```

A. 32, 28
B. 36, 32
C. 28, 26
D. 26, 26

22. (2015) 在“大端法”下，已知如下的 C 语言数据结构：`union { char c[2]; int i; };`。当 `c` 的值为 `0x01, 0x23` 时，`i` 的值为：

A. `0x0123`
B. `0x2301`
C. `0x01230000`
D. 不确定

23. (2014) 有如下定义的结构，在 x86-64 下，下述结论中错误的是？

```
1     struct {
2         char c;
3         union {
4             char vc;
5             double value;
6             int vi;
```

```
        } u;
        int i;
    } sa;
```

A. `sizeof(sa) == 24`

B. `(&sa.i - &sa.u.vi) == 8`

C. `(&sa.u.vc - &sa.c) == 8`

D. 优化成员变量的顺序，可以做到 `sizeof(sa) == 16`

24. (2013) 32 位 x86、Windows 操作系统下定义 struct `S` 包含: `double a, int b, char c`, 请问 `S` 在内存空间中最多和最少分别能占据多少个字节（32 位 Windows 系统按 1、4、8 的原则对齐 `char, int, double`）?

A. 16, 13

B. 16, 16

C. 24, 13

D. 24, 16

25. (2018) 假设在 64 位 Linux 机器上有数据结构定义如下：

```
typedef struct s1 {
    char cc[N];
    int ii[N];
    int *ip;
} S1;

S1 t1[N];
```

(1) 分别确定当 $N = 3, 4, 5$ 时，`sizeof(S1)` 和 `sizeof(T1)` 的输出。

(2) 当 $N = 4$ 时，该数据结构初始化代码如下:

```
void init(int n) {
    int i;
    for (i = 0; i < n; i++) {
        t1[i].ip = &(t1[i].ii[i]);
    }
}
```

根据上述代码，填写下面汇编中缺失的内容：

```
init:
    movl    $0, %ecx
    jmp     .L2

.L3:
    movslq  %ecx, %rax
    leaq    (_____, %rax, 8), %rsi
    leaq    0(, %rsi, 4), %rdx
```

```
        addq    $t1+4, %rdx
        salq    _____, %rax
        movq    _____, _____
        addl    $1, %ecx

    .L2
        cmpl    _____, %ecx
        jl      .L3
        rep ret
```

(3) 当 $N = 3$ 时，函数 fun 的汇编代码如下：

```
    fun:
        movslq   %esi, %rax
        movslq   %edi, %rdi
        leaq     (%rdi, %rdi), %rdx
        leaq     (%rdx, %rdi), %r8
        leaq     (%r8, %r8), %rcx
        addq     %rcx, %rax
        movl     %esi, t1+4(, %rax, 4)
        addq     %rdx, %rdi
        leaq     0(, %rdi, 8), %rax
        movq     t1+16(%rax), %rax
        movl     %esi, (%rax)
        ret
```

根据上述代码，填写函数 fun 的 C 语言代码：

```
1   void fun(int x, int y) {
2       __________ = __________;
3       __________ = __________;
4   }
```

26. (2017) 分析下面 C 语言程序和相应的 x86-64 汇编程序，请填写缺失的内容。

```
1   #include <stdio.h>
2   #include "string.h"
3
4   void myprint(char *str) {
5       char buffer[16];
6       __________(buffer, str);
7       printf("%s \n", buffer);
8   }
9
10  void alert(void) {
11      printf("__________\n");
12  }
13
```

```c
int main(int argc, char *argv[]) {
    myprint("123456712345671234567\xaa\x84\x04\x08");
    return 0;
}
```

```
    .section .rodata
.LC0:
    .string  "___________"
    .text
    .globl   myprint
    .type    myprint, @function
myprint:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    subq     $48, __________
    movq     %rdi, -40(%rbp)
    movq     %fs:40, __________
    movq     %rax, -8(%rbp)
    xorl     %eax, %eax
    movq     __________, %rdx
    leaq     -32(%rbp), %rax
    movq     %rdx, __________
    _______________________________
    call     strcpy
    ____     -32(%rbp), %rax
    movq     %rax, %rsi
    movl     $.LC0, %edi
    movl     $0, %eax
    call     printf
    nop
    ____     __________, %rax
    xorq     __________, __________
    je       __________
    call     __stack_chk_fail
.L2:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

.LFE0:
```

```
    .size    myprint, .-myprint
    .section .rodata
.LC1:
    .string  "Where am I?"
    .text
    .globl   alert
    .type    alert, @function
alert:
.LFB1:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    movl     $.LC1, %edi
    call     puts
    nop
    popq     %rbp
    .cfi_def_cfa 7, 8
    _____________________________
    .cfi_endproc

.LFE1:
    .size    alert, .-alert
    .section .rodata
    .align 8
.LC2:
    .string  "123456712345671234567123456712345671234567\252\___\004\b"
    .text
    .globl   main
    .type    main, @function
main:
    .LFB2:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $.LC2, %edi
    _____________________________
    movl    $0, %eax
```

```
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
    .LFE2:
        .size   main, .-main
```

27. (2015) 在 32 位机器中，有如下声明。

```
union ELE {
    struct {
        int x;
        int *p;
    } e1;
    struct {
        union ELE *next;
        int y;
    } e2;
};
```

(1) 上面的 union 有几个字节?

(2) 假设编译器为 process 函数产生了如下了代码，请补充完整下面的过程。已知只有一个不需要任何强制类型转换且不违反任何类型限制的答案。

```
    movl    8(%ebp), %eax
    movl    (%eax), %ecx
    movl    4(%ecx), %edx
    movl    (%edx), %edx
    subl    4(%eax), %edx
    movl    %edx, (%ecx)
```

```
void process(union ELE *up) {
    up->__________ = __________ - __________;
}
```

(3) 假设整型变量 n 在栈中 %ebp + 8 位置，ELE* 型变量 up 在栈中 %ebp + 12 位置。已知以 *up 为头元素（记为第 0 个），由声明中的 next 指针连接形成了一个链表，现在希望将第 n 个元素（假设链表足够长）的 x 的值放入 %eax 中。以下完成该功能的汇编代码有错，请找出所有错误并改正。

```
        xorl    %ecx, %ecx
        movl    8(%edx), %ebp
        movl    12(%ebp), %eax
    LOOP:
        movl    (%eax), %eax
        add     $1, %ecx
        test    %ecx, %edx
```

```
        jne     LOOP
        movl    (%eax), %eax
```

(4) 阅读下列代码，回答后面的问题。

```
1   typedef struct {
2       short x[A][B];
3       int y;
4   } str1;
5
6   typedef struct {
7       char array[B];
8       int t;
9       short s[B];
10      int u;
11  } str2;
12
13  void setVal(str1 *p, str2 *q) {
14      int v1 = q->t;
15      int v2 = q->u;
16      p->y = v1 + v2;
17  }
```

编译器为 setVal 产生下面的代码：

```
    movl    12(%ebp), %eax
    movl    28(%eax), %edx
    addl    8(%eax), %edx
    movl    8(%ebp), %eax
    movl    %edx, 44(%eax)
```

给出 A，B 的值。

28. (2014) 一个函数如下，其中部分代码被隐去。

```
1   int f(int n, int m) {
2       if (m > 0) {
3           if (_____________________) {
4               int r = _____________________;
5               return _____________________;
6           }
7           else if (_____________________) {
8               return 1;
9           }
10      }
11      return 0;
12  }
```

如下是通过 gcc g O2 命令编译后，在 gdb 中通过 disas f 命令得到的反汇编代码，其中

有两个汇编指令不全。

```
0x00000000004004e0 <f+0>:     mov     %rbx, -0x10(%rsp)
0x00000000004004e5 <f+5>:     mov     ____________________
0x00000000004004ea <f+10>:    xor     %eax, %eax
0x00000000004004ec <f+12>:    sub     $0x10, %rsp
0x00000000004004f0 <f+16>:    test    %esi, %esi
0x00000000004004f2 <f+18>:    mov     %edi, %ebp
0x00000000004004f4 <f+20>:    mov     %esi, %ebx
0x00000000004004f6 <f+22>:    jle     0x400513 <f+51>
0x00000000004004f8 <f+24>:    cmp     $0x1, %edi
0x00000000004004fb <f+27>:    jle     0x400521 <f+65>
0x00000000004004fd <f+29>:    lea     -0x1(%rbp), %edi
0x0000000000400500 <f+32>:    callq   0x4004e0 <f>
0x0000000000400505 <f+37>:    lea     -0x1(%rax, %rbx, 1), %edx
0x0000000000400509 <f+41>:    mov     %edx, %eax
0x000000000040050b <f+43>:    sar     $0x1f, %edx
0x000000000040050e <f+46>:    idiv    %ebp
0x0000000000400510 <f+48>:    lea     0x1(%rdx), %eax
0x0000000000400513 <f+51>:    mov     ____________________
0x0000000000400517 <f+55>:    mov     0x8(%rsp), %rbp
0x000000000040051c <f+60>:    add     $0x10, %rsp
0x0000000000400520 <f+64>:    retq
0x0000000000400521 <f+65>:    sete    %al
0x0000000000400524 <f+68>:    movzbl  %al, %eax
0x0000000000400527 <f+71>:    jmp     0x400513 <f+51>
```

(1) 补全 C 代码和汇编指令。

(2) 已知在调用函数 `f(4, 3)` 时，我们在函数 `f` 中指令 `retq` 处设置了断点，下面列出的是程序在第一次运行到断点处暂停时时，相关通用寄存器的值。

- ▷ `%rax 0x3`
- ▷ `%rcx 0x3`
- ▷ `%rdx 0x309c552970`
- ▷ `%rsi 0x3`
- ▷ `%rdi 0x1`
- ▷ `%rbp 0x2`
- ▷ `%rsp 0x7fffffffe340`
- ▷ `%rip 0x400520`

请根据你对函数及其汇编代码的理解，填写当前栈中的内容。如果某些内存位置处内容不确定，请填写 `X`。

- ▷ `0x7fffffffe38c` _____
- ▷ `0x7fffffffe388` _____
- ▷ `0x7fffffffe384` _____
- ▷ `0x7fffffffe380` _____

▷ 0x7fffffffe37c _____
▷ 0x7fffffffe378 _____
▷ 0x7fffffffe374 _____
▷ 0x7fffffffe370 _____
▷ 0x7fffffffe36c _____
▷ 0x7fffffffe368 _____
▷ 0x7fffffffe364 _____
▷ 0x7fffffffe360 _____
▷ 0x7fffffffe35c _____
▷ 0x7fffffffe358 _____
▷ 0x7fffffffe354 _____
▷ 0x7fffffffe350 _____
▷ 0x7fffffffe34c _____
▷ 0x7fffffffe348 _____
▷ 0x7fffffffe344 _____
▷ 0x7fffffffe340 _____
▷ 0x7fffffffe33c _____
▷ 0x7fffffffe338 _____
▷ 0x7fffffffe334 _____
▷ 0x7fffffffe330 _____
▷ 0x7fffffffe32c _____
▷ 0x7fffffffe328 _____
▷ 0x7fffffffe324 _____
▷ 0x7fffffffe320 _____

29. (2014) 阅读下面的汇编代码，根据汇编代码填写 C 代码中缺失的部分，然后描述该程序的功能。

```
        pushl    %ebp
        movl     %esp, %ebp
        movl     $0x0, %ecx
        cmpl     $0x0, 8(%ebp)
        jle      .L1
    .L2
        movl     $0x0, %edx
        movl     8(%ebp), %eax
        divl     $0x0a
        addl     %edx, %ecx
        movl     %eax, 8(%ebp)
        cmpl     $0x0, 8(%ebp)
        jg       .L2
    .L1
        movl     0x0, %edx
```

```
        movl    %ecx, %eax
        divl    0x3
        cmpl    0x0, %edx
        jne     .L3
        movl    0x1, %eax
        jmp     .L4
    .L3
        movl    0x0, %eax
    .L4
```

```
1   int fun(_____ x) {
2       int bit_sum = 0;
3       while (_________________) {
4           _____________________;
5           _____________________;
6       }
7       if (_________________) {
8           return 1;
9       } else {
10          return 0;
11      }
12  }
```

30. (2013) 阅读下面的 C 代码：

```
1   /**
2    * int_sqrt - rough approximation to sqrt
3    * @x: integer of which to calculate the sqrt
4    *
5    * A very rough approximation to the sqrt() function.
6    */
7   unsigned long int_sqrt(unsigned long x) {
8       unsigned long b, m, y = 0;
9       if (x <= 1)
10          return x;
11      m = 1UL << (BITS_PER_LONG - 2);
12      while (m != 0) {
13          b = y + m;
14          y >>= 1;
15          if (x >= b) {
16              x -= b;
17              y += m;
18          }
19          m >>= 2;
20      }
21      return y;
```

```
22      }
```

已知在 64 位的机器上 BITS_PER_LONG 的定义为 long 类型的位宽。请根据代码填写下面的汇编指令：

```
4004c4: push   %rbp
4004c5: mov    %rsp, %rbp
4004c8: mov    %rdi, -0x28(%rbp)
4004cc: movq   __________, -0x8(%rbp)
4004d4: cmpq   $0x1, -0x28(%rbp)
4004d9: ja     __________ <int_sqrt+??>
4004db: mov    -0x28(%rbp), %rax
4004df: jmp    __________ <int_sqrt+??>
4004e1: movl   $0x0, -0x10(%rbp)
4004e8: movl   __________, -0xc(%rbp)
4004ef: jmp    __________ <int_sqrt+??>
4004f1: mov    -0x10(%rbp), %rax
4004f5: mov    -0x8(%rbp), %rdx
4004f9: lea    __________, %rax
4004fd: mov    %rax, -0x18(%rbp)
400501: shrq   -0x8(%rbp)
400505: mov    -0x28(%rbp), %rax
400509: cmp    -0x18(%rbp), %rax
40050d: jb     __________ <int_sqrt+??>
40050f: mov    -0x18(%rbp), %rax
400513: sub    %rax, -0x28(%rbp)
400517: mov    -0x10(%rbp), %rax
40051b: add    %rax, -0x8(%rbp)
40051f: shrq   __________, -0x10(%rbp)
400524: cmpq   $0x0, -0x10(%rbp)
400529: jne    __________ <int_sqrt+??>
40052b: mov    -0x8(%rbp), __________
40052f: leaveq
400530: retq
```

31. (2013) 某单参数函数 f 的主体的汇编代码如下：

```
4004c4: push   %rbp
4004c5: mov    %rsp, %rbp
4004c8: sub    $0x10, %rsp
4004cc: mov    %edi, -0x4(%rbp)
4004cf: cmpl   $0x1, -0x4(%rbp)
4004d3: ja     4004dc <f+0x18>
4004d5: mov    $0x1, %eax
4004da: jmp    40052d <f+0x69>
4004dc: mov    -0x4(%rbp), %eax
4004df: and    $0x1, %eax
```

```
4004e2: test   %eax, %eax
4004e4: jne    4004f5 <f+0x31>
4004e6: mov    0x200440(%rip), %eax        # 60092c <x.1604>
4004ec: add    $0x1, %eax
4004ef: mov    %eax, 0x200437(%rip)        # 60092c <x.1604>
4004f5: mov    -0x4(%rbp), %eax
4004f8: and    $0x1, %eax
4004fb: test   %al, %al
4004fd: je     40050e <f+0x4a>
4004ff: mov    0x20042b(%rip), %eax        # 600930 <y.1605>
400505: add    $0x1, %eax
400508: mov    %eax, 0x200422(%rip)        # 600930 <y.1605>
40050e: mov    -0x4(%rbp), %eax
400511: sub    $0x1, %eax
400514: mov    %eax, %edi
400516: callq  4004c4 <f>
40051b: mov    0x20040f(%rip), %edx        # 600930 <y.1605>
400521: lea    (%rax, %rdx, 1), %edx
400524: mov    0x200402(%rip), %eax        # 60092c <x.1604>
40052a: lea    (%rdx, %rax, 1), %eax
40052d: leaveq
40052e: retq
```

对应的 C 代码为：

```c
#define N _________________
#define M _________________
struct P1 { char c[N]; char *d[N]; char e[N]; } P1;
struct P2 {int i[M]; char j[M]; short k[M]; } P2;

unsigned int f(unsigned int n) {
    _____ unsigned int x = sizeof(P1);
    _____ unsigned int y = sizeof(P2);
    if (_________________) return 1;
    if (_________________) x++;
    if (_________________) y++;
    return _________________;
}
```

(1) 补全上面的空缺。

(2) 执行 `printf("%x, %x\n", f(2), f(2));` 得到的输出是 _____。

5 体系结构初步

要点

- ▷ 了解体系结构的简要发展史，熟悉 RISC、CISC 的区别和应用。了解基本的门电路、算术运算器、选择器、触发器和寄存器的原理和实现。
- ▷ 熟悉 Y86-64 “体系结构”中，各种指令的编码规则（包括操作数、指令子类型等），会将汇编代码和指令编码相互转换。简单了解 MIPS 体系结构。
- ▷ 在上一项的基础上，理解并熟练记忆 Y86-64 SEQ 处理器执行各种指令的几个阶段及其详细的功能（含重要的中间状态变化）。理解并熟练记忆 Y86-64 顺序处理器的实现（HCL）。
- ▷ 知道流水线设计的通用原理，会进行相关的计算和分析。
- ▷ 在 SEQ 处理器的基础上，理解并熟练记忆 Y86-64 PIPE 处理器和 SEQ 处理器的不同。会分析流水线冒险和转发逻辑。理解并熟练记忆 Y86-64 顺序处理器的实现（HCL）。
- ▷ 理解编译器进行程序性能优化的常见变换手段和不进行优化的保守原因。会通过数据相关图和关键路径分析对性能进行计算（如 CPE 数值）。

1. 下列描述更符合（早期）RISC 还是 CISC?

(1) 指令机器码长度固定。

(2) 指令类型多、功能丰富。

(3) 不采用条件码。

(4) 实现同一功能，需要的汇编代码较多。

(5) 译码电路复杂。

(6) 访存模式多样。

(7) 参数、返回地址都使用寄存器进行保存。

(8) x86-64。

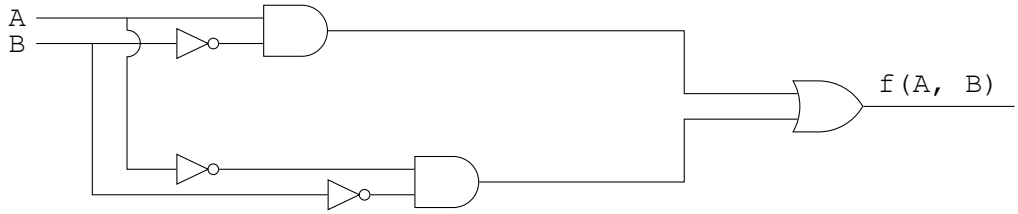
(9) MIPS。

(10) 广泛用于嵌入式系统。

(11) 已知某个体系结构使用 `add R1, R2, R3` 来完成加法运算。当要将数据从寄存器 `S` 移动至寄存器 `D` 时，需要使用 `add S, #ZR, D` 进行操作（`#ZR` 是一个恒为 0 的寄存器）。

(12) 已知某个体系结构提供了 `xlat` 指令，它以一个固定的寄存器 `A` 为基地址，以另一个固定的寄存器 `B` 为偏移量，在 `A` 对应的数组中取出下标为 `B` 的项的内容，放回寄存器 `A` 中。

2. 写出下列电路对应的逻辑表达式：



3. 根据 Y86-64 体系结构完成下表中的更新逻辑：

		call	jXX
Fetch (F)	icode, ifun		
	rA, rB		
	valC		
	valP		
Decode (D)	valA, srcA		
	valB, srcB		
Execute (E)	valE		
	Cond Code		
Memory (M)	valM		
Writeback (W)	dstE		
	dstM		
PC	PC		

4. 已知 `valC` 为指令中的常数值，`valM` 为访存得到的数据，`valP` 为 `PC` 自增得到的值，完成以下的 PC 更新逻辑：

```
int new_pc = [
    icode == ICALL : _____;
    icode == IJXX && Cnd: _____;
    icode == IRET : _____;
    1: _____;
];
```

5. 判断下列说法的正确性：

(1) 流水线的深度越深，总吞吐率越大，因此流水线应当越深越好。

(2) 流水线的吞吐率取决于最慢的流水级，因此流水线的划分应当尽量均匀。

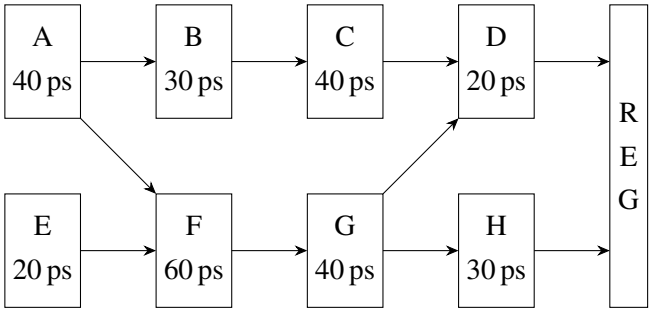
(3) 假设寄存器延迟为 20 ps，那么总吞吐率不可能达到或超过 50 GIPS。

(4) 数据冒险总是可以只通过转发来解决。

(5) 数据冒险总是可以只通过暂停流水线来解决。

6. 一条三级流水线，包括延迟为 50 ps、100 ps、100 ps 的三个流水级，每个寄存器的延迟为 10 ps。那么这条流水线的总延迟是 _____ ps，吞吐率是 _____ GIPS。

7. 在下图中，A~H 为 8 个基本逻辑单元，图中标出了每个单元的延迟，以及用箭头标出了单元之间的数据依赖关系。寄存器的延迟均为 10 ps。



(1) 计算目前的电路的总延迟。

(2) 通过插入寄存器，可以对这个电路进行流水化改造。现在想将其改造为两级流水线，为了达到尽可能高的吞吐率，问寄存器应插在何处？获得的吞吐率是多少？

(3) 现在想将其改造为三级流水线，问最优改造所获得的吞吐率是多少？

8. 一个只使用流水线暂停、没有数据前递的 Y86-64 流水线处理器，为了执行以下的语句，至少需要累计停顿多少个周期？

(1)
```
    irmovq   $1, %rax
    irmovq   $2, %rbx
    addq     %rax, %rcx
    addq     %rbx, %rdx
    halt
```

(2)
```
    rrmovq   %rax, %rdx
    mrmovl   (%rcx), %rax
    addq     %rdx, %rax
    halt
```

(3)
```
    irmovq   $0x40, %rax
    mrmovq   (%rax), %rbx
    subq     %rbx, %rcx
    halt
```

9. 考虑 Y86-64 中的 `ret` 与 `jXX` 指令。`jXX` 总是预测分支跳转。

(1) 写出流水线需要处理 `ret` 的条件（`ret` 对应的常量为 `IRET`）。

(2) 写出发现上述条件以后，各阶段的流水线寄存器应设置的状态。

(3) 写出流水线需要处理 jXX 分支错误的条件（jXX 对应的常量为 IJXX）。

(4) 写出发现上述条件以后，各阶段的流水线寄存器应设置的状态。

(5) 同时考虑上述两条指令，补全下面有关下一条指令地址 f_pc 的控制逻辑。

```
int f_pc = [
    M_icode == IJXX && !M_Cnd : ___________;
    W_icode == IRET : ___________;
    1 : ___________;
];

# 已知有如下的代码，其中 valC 为指令中的常数值
# valM 为访存得到的数据，valP 为 PC 自增得到的值：
int f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];
int d_valA = [
    D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
    # ... 省略部分数据前递代码
    1 : d_rvalA; # Use value read from register file
];
```

10. 有如下定义：

```
// 以下都是局部变量
int i, j, temp, ians;
int *p, *q, *r;
double dans;
// 以下都是全局变量
int iMat[100][100];
double dMat[100][100];
// 以下都是函数
int foo(int x);
```

判断编译器是否会自动进行以下代码优化：

(1)

```
// 优化前
ians = 0;
for (j = 0; j < 100; j++)
for (i = 0; i < 100; i++) ians += iMat[i][j];
// 优化后
ians = 0;
for (i = 0; i < 100; i++)
for (j = 0; j < 100; j++) ians += iMat[i][j];
```

(2)

```
// 优化前
dans = 0;
for (j = 0; j < 100; j++)
```

```
    for (i = 0; i < 100; i++) dans += dMat[i][j];
    // 优化后
    dans = 0;
    for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++) dans += dMat[i][j];
```

(3)

```
    // 优化前
    for (i = 0; i < foo(100); i++) ians += iMat[0][i];
    // 优化后
    temp = foo(100);
    for (i = 0; i < temp; i++) ians += iMat[0][i];
```

(4)

```
    // 优化前
    *p += *q;
    *p += *r
    // 优化后
    temp = *q + *r
    *p += temp;
```

11. 阅读下列 C 代码以及它编译生成的汇编语言。

```
long func() {
    long ans = 1;
    long i;
    for (i = 0; i < 1000; i += 2) ans = ans _?_ (A[i] _?_ A[i+1]);
    return ans;
}
```

```
func:
    movl    $0, %edx
    movl    $1, %eax
    leaq    A(%rip), %rsi
    jmp     .L2
.L3:
    movq    8(%rsi, %rdx, 8), %rcx      // 2 cycles
    __?__   (%rsi, %rdx, 8), %rcx       // k + 1 cycles
    __?__   %rcx, %rax                  // k cycles
    addq    $2, %rdx                    // 1 cycle
.L2:
    cmpq    $999, %rdx                  // 1 cycle
    jle     .L3
    rep ret
```

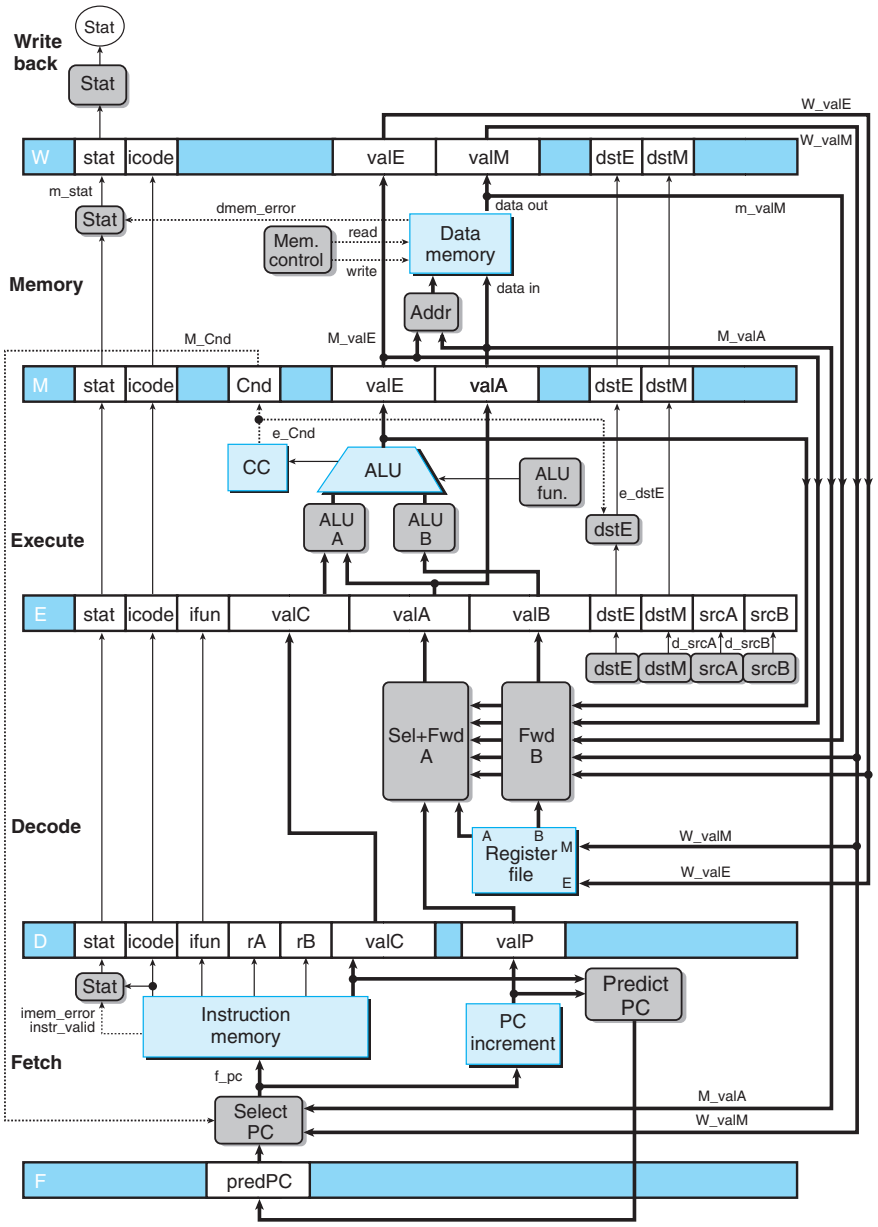
该程序每轮循环处理两个元素。在理想的机器上（执行单元足够多），每条指令消耗的时间周期如右边所示。

(1) 当问号处为乘法时，$k = 8$。此时这段程序的 CPE 为多少？

(2) 当问号处为加法时，$k = 1$。此时这段程序的 CPE 为多少?

6 体系结构初步—往年考题

1. (2018) 以下是一款 Y86 流水线处理器的结构图（局部），请以此为基础，依次回答下列问题。



(1) 该处理器设计采用了前递（forwarding）技术，一定程度上解决了数据相关的问题，在上

图中体现在 Sel+FwdA 和 FwdB 部件上。前者输出的信号会存到流水线寄存器 E 的 valA 域（即 E_valA 信号），请补全该信号的 HCL 语言描述。

```
int E_valA = [
    D_icode in { ICALL, IJXX } : ____________________;
    d_srcA == e_dstE : ____________________;
    d_srcA == M_dstM : ____________________;
    d_srcA == M_dstE : M_valE;
    d_srcA == W_dstM : W_valM;
    ...
];
```

(2) 如果在该处理器上运行下面的程序，每条指令在不同时钟周期所处的流水线阶段如下表所示。在这种情况下，哪条指令的执行结果会有错误？写出该指令的地址 ＿＿＿＿＿＿＿＿。

```
0x000: irmovl $128, %edx
0x006: irmovl $3, %ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10, %ebx
0x018: mrmovl 0(%edx),%eax
0x01e: addl %ebx, %eax
0x020: halt
```

1	2	3	4	5	6	7	8	9	10	11
F	D	E	M	W						
	F	D	E	M	W					
		F	D	E	M	W				
			F	D	E	M	W			
				F	D	E	M	W		
					F	D	E	M	W	
						F	D	E	M	W

(3) 如需检测出这个情况，需要增加逻辑电路，用 HCL 语言表达如下：

```
E_icode in {IMRMOVL, IPOPL} && ___________  in { ___________ }
```

(4) 当新增的电路检测出这个情况后，应对各流水线寄存器进行不同的设置，以便在尽可能少影响性能的前提下解决该问题。请填写下表，可选的设置包括 normal/bubble/stall 三种。

F	D	E	M	W

(5) 如果遇到下面程序代码所展示的情况，该处理器运行时仍然存在问题。因此，还需要新增检测电路。当新增的电路检测出这个情况后，应对各流水线寄存器进行不同的设置，以便在尽可能少影响性能的前提下解决该问题。请填写下表，可选的设置包括 normal/bubble/stall 三种。

```
0x018: rmmovl   %ecx,0(%edx)
0x01e: irmovl   $10, %ebx
0x024: popl     %esp
0x026: ret
```

F	D	E	M	W

2. (2018) Y86 指令 `popl rA` 的 SEQ 实现中，在取指阶段，`valP <-  (  )`，在执行阶段，`valE <-  (  )`。

A. `PC+4、valA+4`

B. `PC+4、valA+(-4)`

C. `PC+2、valB+4`

D. `PC+2、valB+(-4)`

3. (2018) 假设已有声明

```
int i, int sum, int *p, int *q, int *r, const int n = 100, float a[n],
        float b[n], float c[n], int foo(int), void bar()
```

以下哪项程序优化编译器总是可以进行？

A.
```
// 优化前
for(i = 0; i < n; ++i) { a[i] += b[i]; a[i] += c[i]; }
// 优化后
float tmp;
for(i = 0; i < n; ++i) { tmp = b[i] + c[i]; a[i] += tmp; }
```

B.
```
// 优化前
*p += *q; *p += *r;
// 优化后
int tmp;
tmp = *q + *r;
*p += tmp;
```

C.
```
// 优化前
for(i = 0; i < n; ++i) sum += i * 4;
// 优化后
int N = n * 4;
for(i = 0; i < N; i += 4) sum += i;
```

D.
```
// 优化前
for(i = 0; i < foo(n); ++i) bar();
// 优化后
int tmp = foo(n);
for(i = 0; i < tmp; ++i) bar();
```

4. (2018) 在 PIPE 处理器上运行如下 Y86 代码

```
    .L1
        mrmov   (%eax) %ebx
        addl    %ebx, %ecx
        addl    %ecx, %eax
        xorl    %ecx, %edx
        jne     .L1
        irmov   $1, %eax
```

```
        irmov   $1, %eax
```

(1) 假设上述代码一共循环执行了 N 次后跳出循环，未采用数据前递（data forwarding），分支每次都预测正确，访存每次都命中缓存，命中缓存时访存需要 1 个周期。请问共需执行多少个周期?

(2) 假设上述代码一共循环执行了 N 次后跳出循环，采用数据前递（data forwarding），分支每次都预测跳转（taken），访存每次都命中缓存，命中缓存时访存需要 1 个周期。请问共需执行多少个周期?

(3) 假设上述代码一共循环执行了 N 次后跳出循环（N 为偶数)，采用数据前递（data forwarding），分支每次都预测不跳转（not taken），数据访存时缓存命中率为 50%，指令访存全部命中缓存，命中缓存时访存需要 1 个周期，未命中缓存时访存需要 3 个周期。请问共需执行多少个周期?

5. (2017) 在 Y86 的 SEQ 实现中，对仅考虑 IRMMOVQ、ICALL、IPOPQ、IRET 指令，对 mem_addr 的 HCL 描述正确的是：

```
word mem_addr = [
    icode in { (1), (2) } : valE;
    icode in { (3), (4) } : valA;
];
```

A. (1) IRMMOVQ (2) IPOPQ (3) IRET (4) ICALL
B. (1) IRMMOVQ (2) IRET (3) IPOPQ (4) ICALL
C. (1) ICALL (2) IPOPQ (3) IRMMOVQ (4) IRET
D. (1) IRMMOVQ (2) ICALL (3) IPOPQ (4) IRET

6. (2017) 关于流水线技术的描述，错误的是:

A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间。
B. 增加流水线级数，不一定能获得总体性能的提升。
C. 指令间数据相关引发的数据冒险，不一定可以通过暂停流水线来解决。
D. 流水级划分应尽量均衡，吞吐率会受到最慢的流水级影响，均衡的流水线能提高吞吐量。

7. (2017) 分析 64 位的 Y86 ISA 中新加入的条件内存传送指令：crmmovqXX 和 cmrmovqXX。crmmovqXX 和 cmrmovqXX 指令在条件码满足所需要的约束时，分别执行和 rmmovq 以及 mrmovq 同样的语义。其格式如下：

rmmovq	4	0	rA	rB	D (8 bytes)
crmmovqXX	4	fn	rA	rB	D (8 bytes)
mrmovq	5	0	rA	rB	D (8 bytes)
cmrmovqXX	5	fn	rA	rB	D (8 bytes)

(1) 请按下表补全每个阶段的操作。需说明的信号可能会包括：icode、ifun、rA、rB、valA、valB、valC、valE、valP、Cnd；寄存器堆 R[]、存储器 M[]、程序计数器

PC、条件码 CC。其中对存储器的引用必须标明字节数。

阶段	rmmovq rA, D(rB)	cmrmovqXX D(rB), rA
取指		
译码	valA <- R[rA], valB <- R[rB]	
执行		
访存		
写回	N/A	
更新 PC	PC <- valP	

(2) 为了执行上述新增指令，我们需要改进教材所描述的 PIPE 处理器，在回写（W: Write Back）阶段引入寄存器以保持流水线信号 _____（请参考教材对信号的命名规则书写），以便有条件地更新寄存器内容。在如此改进的 PIPE 处理器上，请写出如下信号的 HCL 代码。

```
// F_stall 的 HCL 代码:
(E_icode in {IMRMOVQ, IPOPQ} || (_____1_____))
    && E_dstM in _____2_____ || IRET in _____3_____
// E_bubble 的 HCL 代码:
(_____4_____) || (E_icode in {IMRMOVQ, IPOPQ}
    || (_____1_____)) && E_dstM in _____2_____
// M_bubble 的 HCL 代码:
m_stat in _____5_____ || W_stat in _____5_____
```

(3) 对于下面的 Y86 汇编代码，请使用上述条件内存传送指令将其修改为不带跳转的汇编代码序列。假设下面的代码片段在教材所描述的 PIPE 处理器上运行，不考虑该片段前后代码的影响以及高速缓存（cache）失效的情况，假设 %rsi 初值为 0，处理器设计使用总是选择（always taken）的预测策略。原始代码片段预计运行 _____ 周期，改进代码片段预计执行 _____ 周期。

```
        andq    %rsi %rsi
        jne     L1
        mrmovq  8(%rdx), %rax
        j       L2
    L1:
        mrmovq  8(%rdx), %rbx
    L2:
        addq    %rax, %rbx
```

8. (2016) 下面对指令系统的描述中，错误的是：

A. CISC 指令系统中的指令数目较多，有些指令的执行周期很长；而 RISC 指令系统中通常指令数目较少，指令的执行周期都较短。

B. CISC 指令系统中的指令编码长度不固定；RISC 指令系统中的指令编码长度固定，这样使得 CISC 机器可以获得了更短的代码长度。

C. CISC 指令系统支持多种寻址方式，RISC 指令系统支持的寻址方式较少。

D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问。

9. (2016) 下面对流水线技术的描述，正确的是：

A. 流水线技术不仅能够提高执行指令的吞吐率，还能减少单条指令的执行时间。

B. 不断加深流水线级数，总能获得性能上的提升。

C. 流水级划分应尽量均衡，吞吐率会受到最慢的流水级影响。

D. 指令间的数据相关可能会引发流水线停顿，但总是可以通过调度指令来解决。

10. (2016) 若处理器实现了三级流水线，每一级流水线实际需要的运行时间分别为 1 ns、2 ns 和 3 ns，则此处理器不停顿地执行完毕 10 条指令需要的时间为：

A. 21 ns

B. 12 ns

C. 24 ns

D. 36 ns

11. (2016) 假设流水线数据通路中的转移预测策略为总是预测跳转。如果转移预测错误，需要恢复流水线，并从正确的目标地址开始取值。其中用来判断转移预测是否正确的信号是 _____ 和 _____，用来获得正确的目标地址的信号是 _____。

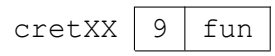
A. `M_icode` `M_Bch` `M_valA`

B. `W_icode` `M_Bch` `M_valA`

C. `W_icode` `M_Bch` `W_valM`

D. `M_icode` `M_Bch` `W_valM`

12. (2016) 请分析 32 位的 Y86 ISA 中新加入的一组条件返回指令 `cretXX`，其格式如下。



类似 `cmovXX`，该组指令只有当条件码 `Cnd` 满足时，才执行函数返回；如果条件不满足，则顺序执行。

(1) 若在教材所描述的 SEQ 处理器上执行这条指令，请按下表补全每个阶段的操作。需说明的信号可能会包括：`icode`、`ifun`、`rA`、`rB`、`valA`、`valB`、`valC`、`valE`、`valP`、`Cnd`；寄存器堆 `R[]`、存储器 `M[]`、程序计数器 `PC`、条件码 `CC`。其中对存储器的引用必须标明字节数。如果在某一阶段没有任何操作，请填写 `none` 指明。

阶段	cretXX offset
取指	
译码	
执行	
访存	
写回	
更新 PC	

(2) 为了执行 cretXX 指令，我们需要改进教材所描述的 PIPE 处理器，在 W（Write Back）阶段引入流水线寄存器 _____，并将其连接到 PC 选择器（Select PC）以便有条件地更新 PC。假设改进后的处理器总是预测函数返回条件不满足，则如果返回条件满足时，一共会错误取指 _____ 条指令。

(3) 在 (2) 中改进的 PIPE 处理器上执行 cretXX 指令时，发生预测错误时的判断条件为：

```
(_____ = ICRETXX && _____) ||
    (_____ = ICRETXX && _____)
```

此时各级流水线寄存器的控制信号应如何设置？填写下表。

F	D	E	M	W
			normal	normal

(4) 若在 PIPE 处理器上处理器上执行如下代码片段：

```
0x000: xorl   %eax, %eax
0x002: popl   %esp
0x004: cretne
```

a) 是否会发生 load-use 和 misprediction cret 组合的 hazard 情况？

b) 如果此时 popl %esp 指令处在流水线的 Execute 阶段，请问此时各级流水线寄存器的控制信号应如何设置？填写下表。

F	D	E	M	W
			normal	normal

13. (2015) 下面有关指令系统设计的描述正确的是：

A. 采用 CISC 指令比 RISC 指令代码更长。
B. 采用 CISC 指令比 RISC 指令运行时间更短。
C. 采用 CISC 指令比 RISC 指令译码电路更加复杂。
D. 采用 CISC 指令比 RISC 指令的流水线吞吐更高。

14. (2015) 一个功能模块包含组合逻辑和寄存器，组合逻辑单元的总延迟是 100 ps，单个寄存器的延时是 20ps，该功能模块执行一次并保存执行结果，理论上能达到的最短延时和最大吞吐分别是多少？

A. 20 ns、50 GIPS
B. 120 ns、50 GIPS
C. 120 ns、10 GIPS
D. 20 ps、10 GIPS

15. (2015) 关于流水线技术的描述，错误的是：

A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间。
B. 减少流水线的级数，能够减少数据冒险发生的几率。
C. 指令间数据相关引发的数据冒险，都可以通过数据转发来解决。
D. 现代处理器支持一个时钟内取指、执行多条指令，会增加控制冒险的开销。

16. (2015) 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], float m
```

都是局部变量，`N` 是一个整数型常量，`int foo(int)` 是一个函数。

A.
```
// 优化前
for (j = 0; j < N; j++) m += i * N * j;
// 优化后
int temp = i * N;
for (j = 0; j < N; j++) m += temp * j;
```

B.
```
// 优化前
for (j = 0; j < N; j++) B[i] *= A[j];
// 优化后
int temp = B[i];
for (j = 0; j < N; j++) temp *= A[j];
B[i] = temp;
```

C.
```
// 优化前
for(j = 0; j < N; j++) m = (m + A[j]) + B[j];
// 优化后
for(j = 0; j < N; j++) m = m + (A[j] + B[j]);
```

D.
```
// 优化前
for(j = 0; j < foo(N); j++) m++;
// 优化后
```

```
4      int temp = foo(N);
5      for(j = 0; j < temp; j++) m++;
```

17. (2015) 请分析 Y86 ISA 中新加入的一条指令：NewJE，其格式如下。

NewJE	C	0	rA	rB	dst

其功能为：如果 R[rA] = R[rB]，则跳转到 dst 继续执行，否则顺序执行。

(1) 若在教材所描述的 SEQ 处理器上执行这条指令，请按下表补全每个阶段的操作。需说明的信号可能会包括：icode、ifun、rA、rB、valA、valB、valC、valE、valP、Cnd；寄存器堆 R[]、存储器 M[]、程序计数器 PC、条件码 CC。其中对存储器的引用必须标明字节数。如果在某一阶段没有任何操作，请填写 none 指明。

阶段	cretXX offset
取指	
译码	
执行	
访存	
写回	
更新 PC	

(2) 若在教材所描述的 PIPE 处理器上执行 NewJE 指令，如果跳转条件不满足，一共会错误执行 _____ 条指令。

为了减小错误预测的代价，现将教材所描述的 PIPE 处理器做如下改进：在 Decode 阶段增加一个比较器，用于判断 R[rA] = R[rB] 条件，比较器的输出信号为 d_equal。如果相等，则 d_equal = 1，反之 d_equal = 0。此时，如果执行 NewJE 指令时跳转条件不满足，一共会错误执行 _____ 条指令。

(3) 已知在教材所描述的 PIPE 处理器上执行 jXX 指令时，发生转移预测错误的判断条件和此时各级流水线寄存器的控制信号如下所示：

```
E_icode = IJXX & !e_Cnd
```

F	D	E	M	W
normal	bubble	bubble	normal	normal

据此，在第 (2) 小题所述的改进后的处理器上执行 `NewJE` 指令，发生转移预测错误的判断条件和此时各级流水线寄存器的控制信号应如何设置？

```
_____________ = INewJE && _____________
```

F	D	E	M	W

(4) 在第 (2) 小题所述的改进后的处理器上执行如下代码：

```
0x000: mrmovl   0(%eax), %edx
0x006: NewJE    %edx, %eax, t
0x00c: irmovl   $1, %eax        # Fall through
0x012: nop
0x013: nop
0x014: nop
0x015: halt
0x016: t:  irmovl $3, %edx      # Target (Should not execute)
0x01c: irmovl   $4, %ecx        # Should not execute
0x022: irmovl   $5, %edx        # Should not execute
```

会发生 load-use 和 misprediction 组合的 hazard 情况。请问此时，各级流水线寄存器的控制信号应如何设置？

F	D	E	M	W
			normal	normal

(5) 在教材 PIPE 处理器设计中，data memory 实际是高速缓存（cache）。假设在执行上述 (4) 中代码时，`0x000` 指令中的 `0(%eax)` 地址中的数据不在 data memory 中，则 data memory 会将输出信号 `m_datamiss` 置为 1，直到数据从内存中取回到 data memory，再将 `m_datamiss` 置为 0。设 `m_datamiss` 的默认值为 0。这种情况的判断条件如下，请问各级流水线寄存器的控制信号应如何设置？

```
M_icode in { IMRMOVL, IPOPL } && m_datamiss
```

F	D	E	M	W

18. (2014) 若处理器实现了三级流水线，每一级流水线实际需要的运行时间分别为 2 ns、2 ns 和 1 ns，则此处理器不停顿地执行完毕 10 条指令需要的时间为：

A. 21 ns

B. 22 ns
C. 23 ns
D. 24 ns

19. (2014) 关于 RISC 和 CISC 的描述，正确的是：
A. CISC 指令系统的指令编码可以很短，例如最短的指令可能只有一个字节，因此 CISC 的取指部件设计会比 RISC 更为简单。
B. CISC 指令系统中的指令数目较多，因此程序代码通常会比较长；而 RISC 指令系统中通常指令数目较少，因此程序代码通常会比较短。
C. CISC 指令系统支持的寻址方式较多，RISC 指令系统支持的寻址方式较少，因此用 CISC 在程序中实现访存的功能更容易。
D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数。

20. (2014) 关于流水线技术的描述，正确的是：
A. 指令间数据相关引发的数据冒险，一定可以通过暂停流水线来解决。
B. 流水线技术不仅能够提高执行指令的吞吐率，还能减少单条指令的执行时间。
C. 增加流水线的级数，一定能获得性能上的提升。
D. 流水级划分应尽量均衡，不均衡的流水线会增加控制冒险。

21. (2014) 下面关于程序性能的说法中，哪个是正确的?
A. 处理器内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）。

22. (2014) 请分析 Y86 ISA 中新加入的一条指令：`caddXX` 条件加法。其功能可以参考 `add` 和 `cmovXX` 两条指令。

caddXX	C	fn	rA	rB

若在教材所描述的 SEQ 处理器上执行这条指令，请按下表补全每个阶段的操作。需说明的信号可能会包括：`icode`、`ifun`、`rA`、`rB`、`valA`、`valB`、`valC`、`valE`、`valP`、`Cnd`；寄存器堆 `R[]`、存储器 `M[]`、程序计数器 `PC`、条件码 `CC`。其中对存储器的引用必须标明字节数。如果在某一阶段没有任何操作，请填写 `none` 指明。

阶段	caddXX offset
取指	
译码	
执行	
访存	
写回	
更新 PC	

7 存储器层次结构

要点

- ▷ 了解 SRAM、DRAM 等易失性存储器及其变种（如 DDR SDARM 等）的功能和历史；了解 PROM、EPROM 等非易失性存储器及其变种的功能和历史。
- ▷ 知道传统旋转磁盘的结构相关概念和工作原理，会计算其存储容量和使用磁盘访问数据的延迟。了解固态硬盘的工作原理。
- ▷ 了解计算机体系结构中总线的作用以及含总线的计算机基本结构。
- ▷ 熟悉时间局部性和空间局部性的概念，会阅读程序判断局部性的概念。

1. 对于下列描述，是 SRAM 更符合还是 DRAM 更符合，还是均符合？

(1) 访问速度更快

(2) 每比特需要的晶体管数目少

(3) 单位容量造价更便宜

(4) 常用作主存

(5) 需要定期刷新

(6) 断电后失去存储的信息

(7) 支持随机访问

(8) 一种变种是 SDRAM

2. 下列存储器中，属于易失性存储介质的有：

A. DRAM

B. SRAM

C. ROM

D. 软盘

E. SSD

F. U 盘

3. 已知一个双面磁盘有 2 个盘片、10000 个柱面，每条磁道有 400 个扇区，每个扇区容量为 512 字节，则它的存储容量是 _____ GB。

4. 已知一个磁盘的平均寻道时间为 6 ms，旋转速度为 7500 RPM，那么它的平均访问时间大约

为 _____ ms。

5. 已知一个磁盘每条磁道平均有 400 个扇区，旋转速度为 6000 RPM，那么它的平均传送时间大约为 _____ ms。

6. 考虑如下程序：

```
for (int i = 0; i < n; i++) {
    B[i] = 0;
    for (int j = 0; j < m; j++)
        B[i] += A[i][j];
}
```

判断下列说法的正确性。

(1) 对于数组 A 的访问体现了时间局部性。

(2) 对于数组 A 的访问体现了空间局部性。

(3) 对于数组 B 的访问体现了时间局部性。

(4) 对于数组 B 的访问体现了空间局部性。

7. 回答下列有关缓存结构的问题：

(1) 一个容量为 8 K 的直接映射高速缓存，每一行的容量为 32 B，那么它有 _____ 组，每组有 _____ 行。

(2) 一个容量为 8 K 的全相联高速缓存，每一行的容量为 32 B，那么它有 _____ 组，每组有 _____ 行。

(3) 一个容量为 16 K 的 4 路组相联告诉缓存，每一行的容量为 64 B，那么一个 16 位地址 0xCAFE 应映射在第 _____ 组内。

8. 判断下列说法的正确性。

(1) 保持块大小与路数不变，增大组数，命中率一定不会降低。

(2) 保持总容量与块大小不变，增大路数，命中率一定不会降低。

(3) 保持总容量与路数不变，增大块大小，命中率一定不会降低。

(4) 使用随机替换代替 LRU，期望命中率可能会提高。

9. 考虑如下程序：

```
int A[MAXN];
for (int i = 0; i < 25; i++) {
    int x = A[i];
    int y = A[i + 1];
    int z = A[i + 2];
    A[i + 3] = x + y + z;
}
```

假设上述代码编译成汇编语言的时候没有任何优化，变量 x、y、z 均放在寄存器中。系统的高速缓存采用写分配策略，运行之前写 cache 所有行都是无效的。设 A 的起始地址为 0，且 MAXN 充分大，保证没有越界问题。

(1) 假设 cache 的容量为 8 字节，每一行的容量为 4 字节，替换策略为 LRU，组策略为直接映射高速缓存。在这个 cache 上运行上述代码，得到的 cache 命中率是 _____%。

(2) 假设 cache 的容量为 8 字节，每一行的容量为 4 字节，替换策略为 LRU，组策略为全相联高速缓存。在这个 cache 上运行上述代码，得到的 cache 命中率是 _____%。

(3) 假设 cache 的容量为 32 字节，每一行的容量为 8 字节，替换策略为 LRU，组策略为 2 路组相联。画出程序运行结束时 cache 的情况（用 M[0-7] 表示第 0 到第 7 字节的地址）；cache 命中率是 _____%。

(4) 假设 cache 的每一行的容量为 4 字节，运行该程序，得到的 cache 命中率的可能最大值为 _____%。

8 存储器层次结构—往年考题

1. (2018) 以下关于存储的叙述中，正确的是：

A. 由于基于 SRAM 的内存性能与 CPU 的性能有很大差距，因此现代计算机使用更快的基于 DRAM 的高速缓存，试图弥补 CPU 和内存间性能的差距。

B. SSD 相对于旋转磁盘而言具有更好的读性能，但是 SSD 写的速度通常比读的速度慢得多，而且 SSD 比旋转磁盘单位容量的价格更贵，此外 SSD 底层基于 EEPROM 的闪存会磨损。

C. 一个有 2 个盘片，10000 个柱面，每条磁道平均有 400 个扇区，每个扇区有 512 个字节的双面磁盘的容量为 8 GB。

D. 访问一个磁盘扇区的平均时间主要取决于寻道时间和旋转延迟，因此一个旋转速率为 6000 RPM、平均寻道时间为 9 ms 的磁盘的平均访问时间大约为 19 ms。

2. (2017) 以下计算机部件中，通常不用于存储器层次结构（Memory Hierarchy）的是：

A. 高速缓存

B. 内存

C. 硬盘

D. 优盘（U 盘）

3. (2017) 关于局部性（locality）的描述，正确的是：

A. 数据的时间局部性或数据空间局部性，在任何有意义的程序中都能体现。

B. 指令的时间局部性或数据空间局部性，在任何有意义的程序中都能体现。

C. 数据的时间局部性，在任何循环操作中都能体现。

D. 数据的空间局部性，在任何数组操作中都能体现。

4. (2016) 以下关于存储结构的讨论，哪个是正确的?

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。

5. (2016) 关于局部性（locality）的描述，不正确的是：

A. 循环通常具有很好的时间局部性。

B. 循环通常具有很好的空间局部性。

C. 数组通常具有很好的时间局部性。

D. 数组通常具有很好的空间局部性。

6. (2015) 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。

7. (2015) 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 _____ ms。

8. (2018) 一缓存共包含 4 个缓存块，每个块 32 字节，采用 LRU 替换算法。设缓存初始为空，对其进行下列 `char` 型内存地址序列访存：`0x5a7, 0x5b7, 0x6a6, 0x5b8, 0x7a5, 0x5b9`。对于直接映射缓存和 2 路组相联缓存，分别能产生几次命中？

A. 1, 3

B. 1, 4

C. 2, 3

D. 2, 4

9. (2018) 设一种全相联缓存共包含 4 个缓存块，如果循环地顺序访问 5 个不同的内存块（大小和一个缓存块一样），下列哪种替换算法会产生最多的命中？（考虑渐近值即可）

A. 最近最少使用替换策略 LRU

B. 先入先出替换策略 FIFO

C. 随机替换策略 Random

D. 后入先出替换策略 LIFO

10. (2018) 某计算机地址空间 12 位，L1 cache 大小为 256 字节，组数 $S = 4$，路数 $E = 2$，现在在地址 `0x0` 处开始有一个 N 行 M 列的 `int` 类型的数组 `int A[N][M]`，有如下 C 代码：

```
1    int ans = 0;
2    for(int j = 0; j < M; ++j)
3        for(int i = 0; i < N; ++i) ans += A[i][j];
```

不考虑并行、编译优化等等会影响访问数组元素顺序的因素，只使用 L1 cache，则如下 (N, M) 对中会导致全部 cache miss 的有几个？$(64, 4), (32, 8), (16, 16), (2, 128)$

A. 4 个

B. 3 个

C. 2 个

D. 1 个

11. (2018) Cache 为处理器提供了一个高性能的存储器层次框架。下面是一个 8 位存储器地址引用的列表（地址单位为字节，数字为 10 进制）：`3, 180, 43, 2, 191, 88, 190, 14, 181, 44`。

(1) 考虑如下 cache：$S = 2, E = 2$，块大小为 2 字节；初始状态为空，替换策略为 LRU。经

历上面的访问序列后，请在下表中填写缓存的状态。格式要求：tag 使用二进制；data 使用十进制，例如 `M[6-7]` 表示地址 6 和 7 对应的数据。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

(2) 现在有另外两种直接映射的 cache 设计方案 C1 和 C2，每种方案的 cache 总大小都为 8 字节，C1 块大小为 2 字节，C2 块大小为 4 字节。假设从内存加载一次数据到 cache 的时间为 25 个周期，访问一次 C1 的时间为 3 个周期，访问一次 C2 的时间为 5 个周期。针对 (1) 的地址访问序列，哪一种 cache 的设计更好？请分别给出两种 cache 访问第一问地址序列的总时间以及 miss rate。

(3) 现在考虑另外一个计算机系统。在该系统中，存储器地址为 32 位，并采用容量 32 KB、块大小 8 字节的直接映射高速缓存，则此 cache 实际至少要占用 _____ 字节的空间。（datasize + (valid bit size + tag size) × blocks）

12. (2017) 在高速缓存存储器中，关于全相联和直接映射结构，以下论述正确的是：

A. 如果配备同样容量、技术的高速缓存，配备全相联高速缓存的计算机总是比配备直接映射高速缓存的计算机性能低。

B. 如果配备同样容量、技术的高速缓存，配备全相联高速缓存的计算机总是比配备直接映射高速缓存的计算机性能高。

C. 如果配备同样容量、技术的高速缓存，当数据在缓存中时，配备全相联高速缓存的计算机总是比配备直接映射高速缓存的计算机读数据慢。

D. 如果配备同样容量、技术的高速缓存，当数据在缓存中时，配备全相联高速缓存的计算机总是比配备直接映射高速缓存的计算机读数据快。

13. (2017) 现有一个能够存储 4 个 block 的 cache，每一个 cache block 的大小为 2 Byte（$B = 2$）。内存空间的大小是 32 Byte，即其地址范围为 0_{10} (00000_2) --- 31_{10} (11111_2)。此设定下一程序的访问内存地址序列如下所示，单位是 Byte，数字为十进制：

0, 3, 4, 7, 16, 19, 21, 22, 8, 10, 13, 14, 24, 26, 29, 30.

(1) 若 cache 的结构如下图所示（$S = 2, E = 2$），初始状态为空，替换策略为 LRU。请在下图空白处填入上述数据访问后 cache 的状态。格式要求：tag 使用二进制；data 使用十进制，例如 `M[6-7]` 表示地址 6 和 7 对应的数据。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

上述数据访问一共产生了 _____ 次命中。

(2) 在 (1) 的基础上增加一条数据预取规则：每当 cache 访问出现 miss 时，被访问地址及其后续的一个 cache block 都会被放入缓存，例如当 `M[0-1]` 访问发生 miss，则把 `M[0-1]` 和 `M[2-3]` 都放入缓存中。此时这 16 次数据访问一共产生了 _____ 次命中。

(3) 在 (1) 的基础上将每一个 cache block 的大小扩大为 4 Byte（即 $B = 4$，cache 大小变为原来的 2 倍），此时这 16 次数据访问一共产生了 _____ 次命中。

(4) 在 (3) 的基础上，考虑增加 (2) 中的数据预取规则，则这 16 次数据访问一共产生了 _____ 次命中。

14. (2016) 现有一个能够存储 4 个 block 的 cache，每一个 cache block 的大小为 2 Byte（$B = 2$）。内存空间的大小是 32 Byte，即其地址范围为 0_{10} (00000_2) --- 31_{10} (11111_2)。此设定下一程序的访问内存地址序列如下所示，单位是 Byte，数字为十进制：

2, 23, 10, 9, 9, 11, 3.

(1) 若 cache 的结构如下图所示（$S = 2, E = 2$），初始状态为空，替换策略为 LRU。请在下图空白处填入上述数据访问后 cache 的状态。格式要求：tag 使用二进制；data 使用十进制，例如 M[6-7] 表示地址 6 和 7 对应的数据。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

上述数据访问一共产生了 _____ 次命中。

(2) 若替换策略为 MRU，其余和 (1) 相同，请在下图空白处填入上述数据访问后 cache 的状态。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

上述数据访问一共产生了 _____ 次命中。

(3) 在 (2) 的基础上增加一条新规则：地址区间包含 5 的倍数的 block 将不会被缓存。仍使用 MRU 替换策略，上述数据访问一共产生了 _____ 次命中。

(4) 在 (3) 的基础上又增加一条数据预取规则：每当地址为 10 的数据被访问时，地址为 8 的数据将会被放入缓存，上述数据访问一共产生了 _____ 次命中。

15. (2015) 某高速缓存满足 $E = 2, B = 4, S = 16$，地址宽度为 14。当引用地址 0x9D28 起始的 1 个字节时，tag 位为：

A. 01110100
B. 001110000
C. 1110000
D. 10011100

16. (2015) 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。

17. (2015) 考虑下面的程序：

```
1  #define LENGTH 8
2  void clear4x4 (char array[LENGTH][LENGTH]) {
3      int row, col;
4      for (col = 0; col < 4; col++)
5          for (row = 0; row < 4; row++)
6              array[row][col] = 0;
7  }
```

(1) 设机器的地址宽度为 7，传入的数组 `array` 的起始地址为 `0x1000000`。两路组相联高速缓存的块大小为 4 字节，一共 4 组，替换算法使用的是 LRU。

a) 以上程序执行会引起多少次失效？

b) 如果 `LENGTH` 改为 16，会引起多少次失效？

c) 如果 `LENGTH` 变为 17，与 b) 相比，下面描述正确的是：_____，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。

d) 请画出 c) 运行后 cache 的最终状态。格式要求：tag 使用二进制；data 使用十进制，例如 `M[6-7]` 表示地址 6 和 7 对应的数据。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

(2) 设机器的地址宽度为 8，传入的数组 `array` 的起始地址为 `0x10000000`。全相联高速缓存的块大小为 4 字节，总容量 16 字节，替换算法使用的是 LRU。

a) Tag 字段的位数是 _____。

b) 以上程序执行会引起多少次失效？

c) 如果 `LENGTH` 改为 16，会引起多少次失效？

d) 如果 `LENGTH` 变为 17，与 c) 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。

e) 请画出 d) 运行后 cache 的最终状态。格式要求：tag 使用二进制；data 使用十进制，例如 `M[6-7]` 表示地址 6 和 7 对应的数据。

V	Tag	Data
1		

V	Tag	Data
1		

V	Tag	Data
1		

V	Tag	Data
1		

18. (2014) 关于 cache 的 miss rate，下面哪些说法是错误的?

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，增大 E，miss rate 一定不会增加。

C. 保持总容量和 E 不变，增大 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。

19. (2014) 现有一个能够存储 4 个 block 的 cache，每一个 cache block 的大小为 2 Byte（$B = 2$）。内存空间的大小是 16 Byte，即其地址范围为 0_{10} (0000_2) --- 15_{10} (1111_2)。此设定下一程序的访问内存地址序列如下所示，单位是 Byte，数字为十进制：

2, 3, 10, 9, 6, 8.

(1) 若 cache 的结构如下图所示（$S = 2, E = 2$），初始状态为空，替换策略为 LRU。请在下图空白处填入上述数据访问后 cache 的状态。格式要求：tag 使用二进制；data 使用十进制，例如 M[6-7] 表示地址 6 和 7 对应的数据。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

上述数据访问一共产生了 _____ 次 miss。

(2) 若替换策略为 MRU，其余和 (1) 相同，请在下图空白处填入上述数据访问后 cache 的状态。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

上述数据访问一共产生了 _____ 次 miss。

20. (2013) 如果直接映射高速缓存大小是 4 KB，块大小为 32 字节，则它每组有多少行?

A. 128

B. 68

C. 32

D. 1

21. (2013) 现有一个能够存储 4 个 block 的 cache，每一个 cache block 的大小为 2 Byte（$B = 2$）。内存空间的大小是 32 Byte，即其地址范围为 0_{10} (00000_2) --- 31_{10} (11111_2)。此设定下一程序的访问内存地址序列如下所示，单位是 Byte，数字为十进制：

1, 4, 17, 2, 8, 16, 9, 0.

(1) 如果 cache 的结构是直接映射，满足 $S = 4, E = 1$，请在下表空白处填入访问上述数据序列访问后 cache 的状态。格式要求：tag 使用二进制；data 使用十进制，例如 M[6-7] 表示地址 6 和 7 对应的数据。

V	Tag	Data
1		

V	Tag	Data
1		

V	Tag	Data
1		

V	Tag	Data
1		

(2) 如果 cache 的结构如下表所示，满足 $S = 2, E = 2$，请在下表空白处填入访问上述数据序列访问后 cache 的状态。

	V	Tag	Data	V	Tag	Data
SET 0						
SET 1						

上述数据访问一共产生了 _____ 次 miss。

(3) 如果 cache 的结构改为 $S = 1, E = 4$，最终存储在 cache 里面的数据有哪些？只需要填写数据部分，顺序不限。

_____、 _____、 _____、 _____。

9 讲座课

1. 下列有关微处理器及其设计自动化的叙述不妥当的是：

A. 芯片设计自动化英文全称为 Electronic Design Automation，是一个涉及物理学、计算机科学、应用数学的跨领域研究。

B. 微处理器的最早的先驱一般认为是 Intel 公司 1971 年出产的 4004 型，它有大约 2300 个晶体管，位宽为 4。

C. 硬件描述语言有 `SystemC`，`Verilog`，`VHDL`，`HCL` 等；将这些语言翻译为实际生产中的描述的工具称为“硬件编译器”，翻译过程一般称为 synthesis。

D. EDA 过程中面临许多问题，包括 placement、routing 等，这些实际问题的求解中的困难主要是没有多项式时间的算法和问题规模太大。

2. 摩尔定律（Moore's law）是由英特尔创始人之一戈登·摩尔提出的。根据摩尔定律，在过去几十年中，单块集成电路的集成度大约每 _____ 个月翻一番，性能达到原来的 _____ 倍。

A. 24、2

B. 24、1.5

C. 18、2

D. 18、1.5

3. 下列有关 placement 问题的叙述，错误的是：

A. 问题的数学模型是在一个二维平面上放置若干几何块，使得它们互不重叠且相互之间的距离之和（wirelength）尽量小。

B. 历史上出现过很多针对该问题的算法，例如划分、模拟退火、最小割、优化方法等，其中模拟退火结果不好，划分法效率较低。

C. 使用优化方法的一个主要困难是优化问题的目标和约束光滑性不足，或者是非凸的，解决方案之一是采用一些光滑函数近似。

D. 若用优化方法求解，则可以引入势函数、梯度下降、深度神经网络等方法启发式地求解。

4. 下列有关 routing 问题的叙述，错误的是：

A. Routing 一般是在各 cell 的 placement 后决定的，问题的数学模型是决定各个 cell 之间的最短路，有多项式时间的算法。

B. 问题常见的约束包括：关键 cell 网络中延迟界、总 wirelength 长度、芯片面积和其他几何约束、和光刻有关的约束等等。

C. 最短路问题可以用大家熟知的迷宫问题建模，多项式时间算法可以是 Lee 算法（即 BFS）。在问题规模较大时可以用 Aker 的数据结构等进行优化。

D. Routing 问题在引脚较多时需要引入斯坦纳树等结构，具体而言问题是 NP-完全的，不过可以用深度强化学习来启发式求解。

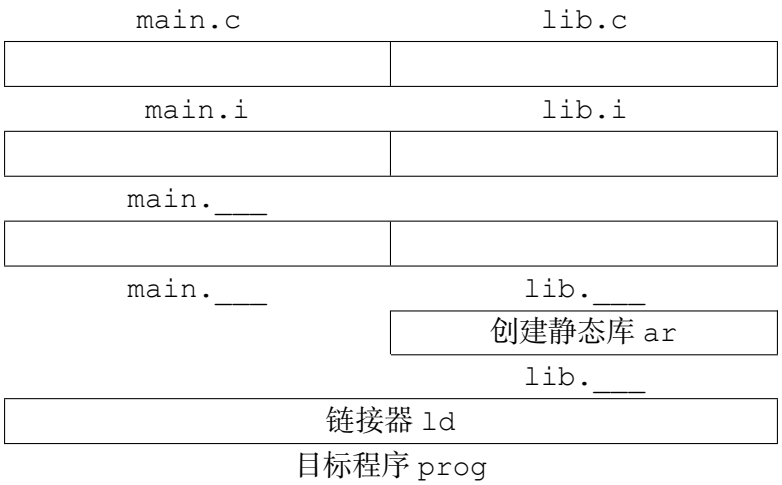
5. 北京大学计划筹建新校区，请你帮忙规划新校区的建筑分布。假设新校区在 xy 平面上，其边界是多边形曲线，每段都平行于 x 或 y 轴。已知宿舍楼 50 幢、食堂 10 个、教学楼 10 幢、绿地 20 块，均为边界平行于 x, y 轴的已知大小的矩形。要求宿舍楼、教学楼和食堂尽可能靠近，且不同宿舍楼、教学楼尽量靠近不同的食堂，不同建筑不能重叠。试为问题建立基于优化的数学模型。

10 链接

要点

- ▷ 知道源代码编译为可执行文件的全过程，知道静态链接和动态链接的概念，会正确书写含有多文件的代码的编译命令（静态库的顺序问题）。
- ▷ 熟悉 Linux 下典型目标文件格式 ELF 的各个部分，掌握符号表、局部符号、全局符号、强符号、弱符号等概念及其存储的区域。
- ▷ 掌握符号解析的过程，会判断链接是否成功以及失败的原因。
- ▷ 熟练掌握重定位的条件和过程，会根据重定位条目计算相对引用和绝对引用。
- ▷ 了解目标程序被加载到内存的过程，知道动态链接的概念及其和静态链接的区别和优劣，了解位置无关代码和库打桩的基本技术。

1. 下图为一个典型的编译过程。将正确的过程填上，并补充缺失的拓展名。可供选择的过程有：汇编器 `as`、预处理器 `cpp`、编译器 `cc1`。



2. 判断下面关于静态链接的说法是否正确。

(1) 链接时，链接器会拷贝静态库（`.a`）中的所有模块（`.o`）。

(2) 链接时，链接器只会从每个模块（`.o`）中拷贝出被用到的函数。

(3) 链接时，如果所有的输入文件都是 `.o` 或 `.c` 文件，那么任意交换输入文件的顺序，都不会影响链接是否成功。

(4) 链接时，通过合理地安排静态库和模块的顺序，每个静态库都可以在命令中出现至多一次。

3. 有下面两个程序。将他们先分别编译为 .o 文件，再链接为可执行文件。

```c
/* main.c */
#include <stdio.h>
__________A__________
int foo(int n) {
    static int ans = 0;
    ans = ans + x;
    return n + ans;
}
int bar(int n);
void op(void) { x = x + 1; }

int main() {
    for (int i = 0; i < 3; i++) {
        int a1 = foo(0);
        int a2 = bar(0);
        op();
        printf("%d %d ", a1, a2);
    }
    return 0;
}
```

```c
/* count.c */
__________B__________
int bar(int n) {
    static int ans = 0;
    ans = ans + x;
    return n + ans;
}
```

(1) 当 A 处为 `int x = 1;`，B 处为 `int x;` 时，完成下表。如果某个变量不在符号表中，那么在名字那一栏划 X；如果它在符号表中的名字含有随机数字，那么请用不同的四位数字区分多个不同的符号。对于局部符号，不需要填最后一栏。

文件名	变量名	在符号表中的名字	是局部符号吗?	是强符号吗?
main.c	x			
	bar			
	ans			
count.c	x			
	bar			
	ans			

程序能够链接成功吗？如果可以，程序的运行结果是什么？如果不可以，链接器报什么错？

(2) 当 A 处为 `static int x = 1;`，B 处为 `static int x = 1;` 时，完成下表。

文件名	变量名	在符号表中的名字	是局部符号吗？	是强符号吗？
main.c	x			
	bar			
	ans			
count.c	x			
	bar			
	ans			

程序能够链接成功吗？如果可以，程序的运行结果是什么？如果不可以，链接器报什么错？

(3) 当 A 处为 `int x = 1;`，B 处为 `int x = 1;` 时。程序能够链接成功吗？如果可以，程序的运行结果是什么？如果不可以，链接器报什么错？

4. 在链接时，哪类符号一定不需要重定位？

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. 同一函数中定义的不带 `static` 的变量
D. 同一函数中定义的带 `static` 的变量

5. 在 x86-64 的机器上用 gcc-7 编译可以顺利编译并运行以下两个文件。若某次运行时得到输出 `0x48\n`，则这个 16 进制的 48 产生自：

```c
// f1.c
void p2(void);
int main() {
    p2();
    return 0;
}
// f2.c
#include <stdio.h>
char main;
void p2() { printf("0x%x\n", main); }
```

A. 垃圾值
B. `main` 函数汇编地址的最低字节按有符号补齐的结果
C. `main` 函数汇编地址的最高字节按有符号补齐的结果
D. `main` 函数汇编的第一个字节按有符号补齐的结果

6. 有如下 C 代码：

```c
#define k 100
long foo(long n);
long bar(long n) {
    static long ans = 0;
    long acc = 0;
    for (int i = 0; i < n; i++) {
        ans += i;
        acc += ans * n;
    }
    return ans + acc;
}
long t;
static long y;
extern long z;
int main() {
    long x;
    myScanf("%ld%ld%ld", &x, &y, &z);
    myPrintf("%ld %ld\n", foo(x + y + t), bar(z + k));
    return 0;
}
```

采用命令 `gcc test.c -c -Og -no-pie -fno-pie` 与 `readelf -a test.o > t.txt` 后得到解析文件。

`t.txt` 中的部分节头部表信息如下：

号	名称	类型	地址	偏移量
1	.text	PROGBITS	0000000000000000	00000040
3	.data	PROGBITS	0000000000000000	000000ff
4	.bss	NOBITS	0000000000000000	00000100
5	.rodata.str1.1	PROGBITS	0000000000000000	00000100
10	.symtab	SYMTAB	0000000000000000	00000190

提示 节头部表各条目含义可参 http://www.skyfree.org/linux/references/ELF_Format.pdf 第 9 页，特别关注 sh_addr、sh_offset、sh_size。

`t.txt` 中的部分符号表如下：

Num	Value	Size	Type	Bind	Vis	Ndx	Name
5	0000000000000000		OBJECT		DEFAULT		ans.1797
7	0000000000000008		OBJECT		DEFAULT		y
11	0000000000000000	52	FUNC		DEFAULT		bar
12	0000000000000034	139	FUNC	GLOBAL	DEFAULT		main
13	0000000000000000	0	NOTYPE		DEFAULT		z
15	0000000000000008		OBJECT		DEFAULT		t

(1) 除了上述已经列出的符号外，判断下列名字是否在符号表中。

名称	k	ans	acc	foo	y.????	x	n
是否出现							

(2) 补全上述符号表中漏掉的信息。其中 Bind 可以是 LOCAL 或者 GLOBAL，Ndx 可以是表示节头标号的数字，也可以是 UND (undefined) 或 COM (common)。

(3) 字符串 "%ld %ld\n" 位于哪个节中?

(4) 假设在全局区域定义 long A[1000000]，那么在 test.o 中，.bss 节占用的空间为多少字节?

(5) 使用 objdump -dx test.o 查看发现有如下的汇编代码：

```
0000000000000000 <bar>:
    0: b9 00 00 00 00          mov    $0x0, %ecx
...
0000000000000034 <main>:
    34: 53                     push   %rbx
...
    6b: e8 00 00 00 00         callq  70 <main+0x3c>
    6c: R_X86_64_PC32 bar-0x4
...
    90: bf 00 00 00 00         mov    $0x0, %edi
    91: R_X86_64_32 .rodata.str1.1+0xa
```

现在将若干个 .o 文件链接成可执行文件 done。假设链接器已经确定 test.o 的 .text 节在 done 中的起始地址为 ADDR(.text) = 0x400517。

a) 链接后，test.o 中的 6b 处的指令变为 done 中如下的指令：

```
_____________: e8 ___ ___ ___ ___          callq 400517 <bar>
```

请补充以上五个空格（其中第一个空格是指令地址，之后四个空格是机器码）。

b) test.o 中 90 处的指令变为 done 中如下的指令：

```
4005a7: bf 9e 06 40 00          mov $0x40069e, %edi
```

则可执行文件 done 中，.rodata.str1.1 的起始地址为 0x_____。

(6) 对 done 使用 objdump，发现有如下的函数：

```
0000000000400430 <_start>
000000000040054b <main>
0000000000600ff0 <__libc_start_main@GLIBC_2.2.5>
```

则 done 的入口点地址是 0x_____。

7. (丁睿助教提供) 判断下列说法的正确性。

(1) 动态链接可以在加载时或者运行时完成，它比静态库更节省内存和磁盘上的储存空间。

(2) 动态库可以不编译成位置无关代码。

(3) 通过代码段的全局偏移量表 GOT 和数据段的过程链接表 PLT，动态链接器可以完成延迟绑定 (lazy binding)。

(4) ASLR 不会影响代码段和数据段间的相对偏移，这样位置无关代码才能正确使用。

8. (丁睿助教提供) 本题基于下列 m.c 及 foo.c 文件，使用 gcc foo.c m.c 生成 a.out。

```c
// m.c
void foo();
int buf[2] = {1, 2};
int main() {
    foo();
    return 0;
}
// foo.c
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;
void foo() {
    static int count = 0;
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
    count++;
}
```

(1) 读取 a.out 的节头部表，部分信息如下。

号	名称	类型	地址	偏移量	大小
1	.interp	PROGBITS	00000000000002a8	000002a8	000000000000001c
14	.text	PROGBITS	0000000000001050	00001050	0000000000000205
16	.rodata	PROGBITS	0000000000002000	00002000	000000000000000a
23	.data	PROGBITS	0000000000004000	00003000	0000000000000020
24	.bss	NOBITS	0000000000004020	00003020	0000000000000010

Num	Value	Size	Type	Bind	Ndx	Name
35	00...004024					count.1797
54	00...004010	8	OBJECT			bufp0
59	00...00115a	78	FUNC	GLOBAL		foo
62			OBJECT	GLOBAL		buf
64	00...0011a8	54		GLOBAL	14	main
68	00...004028	8	OBJECT	GLOBAL		bufp1
51	00...000000	0	FUNC		UND	printf@@GLIBC_2.2.5

a) 补全符号表中的空缺部分。

b) 读取 a.out 中的 .interp 节，发现是一个可读字符串，请填空补全读出的结果：/lib64/____-linux-x86-64.____.2。

c) 读取 a.out 中的 .bss 节，其存储时占用的空间为 _____ 字节，运行时占用的空间为 _____ 字节。

(2) 现在通过 gcc -c m.c; objdump -dx m.o 我们看到如下重定位信息。

```
0000000000000000 <main>:
0:   55                    push %rbp
...
10:  8b 15 00 00 00 00 mov 0x0(%rip), %edx # 16 <main+0x16>
                12: R_X86_64_PC32       buf
...
1e:  48 8d 3d 00 00 00 00 lea 0x0(%rip), %rdi # 25 <main+0x25>
                21: R_X86_64_PC32       .rodata-0x4
...
2a:  e8 00 00 00 00        callq  2f <main+0x2f>
                2b: R_X86_64_PLT32      printf-0x4
...
```

a) 假设链接器生成 a.out 时已经确定 m.o 的 .text 节在 a.out 中的起始地址为 0x11a8。请写出重定位后的对应于原本 main+0x10 位置的代码。

b) 而 main+0x1e 处的指令变成

```
11c6: 48 8d 3d 37 0e 00 00 lea 0xe37(%rip), %rdi
```

可见字符串 "%d %d"在 a.out 中的起始地址是 0x_____。

(3) 使用 objdump -d a.out 可以看到如下 .plt 节的代码。

```
Disassembly of section .plt:

0000000000001020 <.plt>:
1020: ff 35 9a 2f 00 00 pushq 0x2f9a(%rip)
        # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
1026: ff 25 9c 2f 00 00 jmpq *0x2f9c(%rip)
        # 3fc8 <_GLOBAL_OFFSET_TABLE_+0x10>
102c: 0f 1f 40 00 nopl 0x0(%rax)

0000000000001030 <printf@plt>:
1030: ff 25 9a 2f 00 00 jmpq *0x2f9a(%rip)
        # 3fd0 <printf@GLIBC_2.2.5>
1036: 68 00 00 00 00 pushq $0x0
103b: e9 e0 ff ff ff jmpq 1020 <.plt>
```

a) 完成 main+0x2a 处的重定位。

```
_____: e8 ___ ___ ___ ___     callq  1030 <printf@plt>
```

b) printf 的 PLT 表条目是 PLT[____]，GOT 表条目是 GOT[____]（均填写数字）。

c) 使用 gdb 对 a.out 进行调试。某次运行时 main 的起始地址为 0x5555555551a8，那么当加载器载入内存而尚未重定位 printf 地址前，printf 的 GOT 表项的内容是 _____。你填写的这个值是 _____（填静态/动态）链接器设置的。而重定位后可以使用 disas _____ 读出 printf 动态链接进来的代码。
提示〉 disas 是 gdb 中用于反汇编的指令。Gdb 如果通过立即数直接访问内存地址，直接使用该数即可。如果需要从一个地址中读值并以此间接访问内存，可以使用 *(long *) 0xImm 的格式，其中 Imm 表示该立即数。

11 链接—往年考题

1. (2018) 下列关于链接技术的描述，错误的是：

A. 在 Linux 系统中，对程序中全局符号的不当定义，会在链接时刻进行报告。

B. 在使用 Linux 的默认链接器时，如果有多个弱符号同名，那么会从这些弱符号中任选一个占用空间最大的符号。

C. 编译时打桩需要能够访问程序的源代码，链接时打桩需要能够访问程序的可重定位对象文件，运行时打桩只需要能访问可执行目标文件。

D. 链接器的两个主要任务是符号解析和重定位。符号解析将目标文件中的全局符号都绑定到唯一的定义，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。

2. (2018) 本题基于下列 `m.c` 及 `foo.c` 文件所编译生成的 `m.o` 和 `foo.o`，编译未加优化选项。

```
1   // m.c
2   void foo();
3   int buf[2] = {1, 2};
4   int main() {
5       foo();
6       return 0;
7   }
8   // foo.c
9   extern int buf[];
10  int *bufp0 = &buf[0];
11  int *bufp1;
12  void foo() {
13      static int count = 0;
14      int temp;
15      bufp1 = &buf[1];
16      temp = *bufp0;
17      *bufp0 = *bufp1;
18      *bufp1 = temp;
19      count++;
20  }
```

(1) 对于每个 `foo.o` 中定义和引用的符号，请用“是”或“否”指出它是否在模块 `foo.o` 的 `.symtab` 节中有符号表条目。如果存在条目，则请指出定义该符号的模块（`foo.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；如果不存在条目，则

请将该行后继空白处标记为“/”。

符号	是否为.symtab 条目	符号类型	定义符号的模块	节
bufp0	是	全局	foo.o	.data
buf				
bufp1				
foo				
temp				
count				

```
0000... <main>:
55                  push %rbp
48 89 e5            mov %rsp, %rbp
b8 00 00 00 00      mov 0x0, %eax
e8 00 00 00 00      callq e <main+0xe>
b8 00 00 00 00      mov 0x0, %eax
5d                  pop %rbp
c3                  retq
```

```
0000...fe8 <main>:
fe8: 55                     push %rbp
fe9: 48 89 e5               mov %rsp, %rbp
fec: b8 00 00 00 00         mov 0x0, %eax
ff1: e8 ①                   callq 1000 <foo>
ff6: b8 00 00 00 00         mov 0x0, %eax
ffb: 5d                     pop %rbp
ffc: c3                     retq
          ... 略去和答题无关的部分信息 ...
```

```
0000... <foo>:
55                  push %rbp
48 89 e5            mov %rsp, %rbp
48 c7 05 00 00 00 00 00 00 00 00
                    movq 0x0, 0x0(%rip)
48 8b 05 00 00 00 00
                    movq 0x0(%rip), %rax
8b 00               mov (%rax), %eax
89 45 fc            mov %eax, -0x4(%rbp)
48 8b 05 00 00 00 00
                    movq 0x0(%rip), %rax
48 8b 15 00 00 00 00
                    movq 0x0(%rip), %rdx
8b 12               mov (%rdx), %edx
89 10               mov %edx, (%rax)
48 8b 05 00 00 00 00
                    movq 0x0(%rip), %rax
8b 55 fc            mov -0x4(%rbp), %edx
89 10               mov %edx, (%rax)
8b 05 00 00 00 00
                    movq 0x0(%rip), %eax
83 c0 01            add 0x1, %eax
89 05 00 00 00 00
                    mov %eax, 0x0(%rip)
90                  nop
5d                  pop %rbp
c3                  retq
```

```
0000...1000 <foo>:
1000: 55                    push %rbp
1001: 48 89 e5              mov %rsp, %rbp
1004: 48 c7 05 ...
                            movq ②, ③(%rip)
100f: 48 8b 05 ...
                            movq ___(%rip), %rax
1016: 8b 00                 mov (%rax), %eax
1018: 89 45 fc              mov %eax, -0x4(%rbp)
101b: 48 8b 05 ...
                            movq ④(%rip), %rax
1022: 48 8b 15 ...
                            movq ___(%rip), %rdx
1029: 8b 12                 mov (%rdx), %edx
102b: 89 10                 mov %edx, (%rax)
102d: 48 8b 05 ...
                            movq ___(%rip), %rax
1034: 8b 55 fc              mov -0x4(%rbp), %edx
1037: 89 10                 mov %edx, (%rax)
1039: 8b 05 ...
                            movq ___(%rip), %eax
103f: 83 c0 01              add 0x1, %eax
1042: 89 05 ...
                            mov %eax, ⑤(%rip)
1048: 90                    nop
1049: 5d                    pop %rbp
1050: c3                    retq
          ... 略去和答题无关的部分信息 ...
0000...2330 <buf>:
          ... 略去和答题无关的部分信息 ...
0000...2338 <bufp0>:
          ... 略去和答题无关的部分信息 ...
0000...3024 <count.1837>:
          ... 略去和答题无关的部分信息 ...
0000...3028 <bufp1>:
```

(2) 上图左边给出了 m.o 和 foo.o 的反汇编文件，右边给出了采用某个配置链接成可执行程序后再反汇编出来的文件。根据答题需要，其中的信息略有删减。图中对所涉及到的重定位条目用带圈数字或者省略的方式进行了标记，请根据下表所提供的重定位条目信息，计算相应的重定位引用值并填写下表（只需要填写有标号的重定位条目）。

编号	重定位条目信息	应填入的重定位引用值
①	r.offset = 0xa r.symbol = ??? r.type = R_X86_64_PC32 r.addend = -4	
②	r.offset = 0xb r.symbol = buf r.type = R_X86_64_32 r.addend = +4	
③	r.offset = 0x7 r.symbol = bufp1 r.type = R_X86_64_PC32 r.addend = -8	
④	r.offset = 0x1e r.symbol = bufp0 r.type = R_X86_64_PC32 r.addend = -4	
⑤	r.offset = 0x44 r.symbol = ??? r.type = R_X86_64_PC32 r.addend = -4	

3. (2016) C 源文件 f1.c 和 f2.c 的代码分别如下所示，编译链接生成可执行文件后执行，输出结果为：

```c
// f1.c
#include <stdio.h>
static int var = 100;
int main(void) {
    extern int var;
    extern void f();
    f();
    printf("%d\n", var);
    return 0;
}
// f2.c
int var = 200;
```

```
13    void f() { var++; }
```

A. 100
B. 200
C. 201
D. 链接错误

4. (2016) C 源文件 m1.c 和 m2.c 的代码分别如下所示，

```
1     // m1.c
2     #include <stdio.h>
3     int a1;
4     int a2 = 2;
5     extern int a4;
6     void hello() {
7         printf("%p ", &a1);
8         printf("%p ", &a2);
9         printf("%p\n", &a4);
10    }
11    // m2.c
12    int a4 = 10 ;
13    int main() {
14        extern void hello();
15        hello();
16        return 0;
17    }
```

编译链接生成可执行文件后执行：`$ gcc -o a.out m2.c m1.c; ./a.out`，其结果最可能为：

A. 0x1083018, 0x108301c
B. 0x1083028, 0x1083024
C. 0x1083024, 0x1083028
D. 0x108301c, 0x1083018

5. (2016) 在 x86-64 环境下，考虑如下 4 个文件（main.c, value.c, f1.c, f2.c）:

```
1     /* main.c */
2     #include <stdio.h>
3     extern void f_void_void();
4     extern int f_int_void();
5     void *f;
6     int main() {
7         int a = 1, b, c;
8
9         f = (void *)f_void_void;
10        ((void (*)(int))f)(a);
11        b = ((int (*)())f)();
```

```
12          printf("b = %d\n", b);
13
14          f = (void *)f_int_void;
15          ((void (*)())f)(a);
16          c = ((int (*)(int))f)(a);
17          printf("c = %d\n", c);
18          return 0;
19      }
20
21      /* value.c */
22      int BIG;
23
24      /* f1.c */
25      #include <stdio.h>
26      extern int BIG;
27      int small = 1;
28      void f_void_void() {
29          small += 1;
30          BIG += 1;
31          printf("small = %d, BIG = %d\n", small, BIG);
32      }
33
34      /* f2.c*/
35      #include <stdio.h>
36      extern int BIG;
37      static int small;
38      int f_int_void() {
39          small += 1;
40          BIG += 1;
41          printf("small = %d, BIG = %d\n", small, BIG);
42          return small + 1;
43      }
```

使用命令 `gcc -o main main.c f1.c f2.c value.c` 编译这四个文件，再使用 `./main` 运行编译好的程序。

(1) 请在下表中给出程序中相关符号的属性（局部或全局、强符号或弱符号），其中不确定的请在表格中填 X。

源文件	符号名	局部/全局	强符号/弱符号
main.c	f		
value.c	BIG		
f1.c	small		
f2.c	small		

(2) 请补全程序运行的输出，对于不确定的空请填 X。

```
small = _____, BIG = _____
small = _____, BIG = _____
b = _____
small = _____, BIG = _____
small = _____, BIG = _____
c = _____
```

6. (2015) 在 `foo.c` 文件中包含如下代码：

```c
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`

7. (2015) 在 x86-64 环境下，考虑如下 2 个文件 `main.c` 和 `foo.c`：

```c
/* main.c */
#include <stdio.h>
long long ____________________________;
const char* foo(int);
int main(int argc, char **argv){
    int n = 0;
    sscanf(argv[1], "%d", &n);
    printf(foo(n));
    printf("%llx\n", a);
}

/* foo.c */
#include <stdio.h>
int a[2];
static void swapper(int num){
    int swapper;
    if (num % 2){
        swapper = a[0];
        a[0] = a[1];
        a[1] = swapper;
    }
}

const char* foo(int num){
```

```
25          static char out_buf[50];
26          swapper(num);
27          sprintf(out_buf, "%x\n", ____________________________);
28          return out_buf;
29      }
```

(1) 请在下表中给出程序中相关符号的属性（局部或全局、强符号或弱符号），其中不确定的请在表格中填 X。

源文件	符号名	局部/全局	强符号/弱符号
main.c	a		
	foo		
foo.c	a		
	foo		
	out_buf		

(2) 根据如下的程序运行结果，补全代码（在上面源代码的空白处填空即可）。

```
$ gcc -o test main.c foo.c
$ ./test 1
bffedead
cafebffedeadbeef

$ ./test 2
beefcafe
deadbeefcafebffe
```

(3) 现在有一位程序员要为这个程序编写头文件。假设新的头文件名称为 foo.h，内容如下：

```
1   extern long long a;
2   extern char *foo(int);
```

然后在 main.c 和 foo.c 中分别引用该头文件，请问编译链接能通过吗？说明理由。

8. (2014) 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。

9. (2014) 已知有源文件 foo.c，在该文件中的函数外，如果添加语句

```
static int count = 0xdeadbeef;
```

那么它在编译为 foo.o 后，会影响到 ELF 可重定位目标文件中的除 .text 以外的哪些 section?

A. .rodata
B. .data, .symtab,

C. .data, .symtab, .rel.data

D. .rodata, .symtab, .rel.data

10. (2014) 考虑如下 3 个文件：main.c、fib.c 和 bignat.c：

```c
/* main.c */
void fib (int n);
int main (int argc, char** argv) {
    int n = 0;
    sscanf(argv[1], "%d", &n);
    fib(n);
}

/* fib.c */
#define N 16
static unsigned int ring[3][N];

static void print_bignat(unsigned int *a) {
    int i;
    for (i = N-1; i >= 0; i--)
    printf("%u ", a[i]); /* print a[i] as unsigned int */
    printf("\n");
}

void fib (int n) {
    int i, carry;
    from_int(N, 0, ring[0]); /* fib(0) = 0 */
    from_int(N, 1, ring[1]); /* fib(1) = 1 */
    for (i = 0; i <= n - 2; i++) {
        carry = plus(N, ring[i % 3],
                                ring[(i + 1) % 3], ring[(i + 2) % 3]);
        if (carry) {
            printf("Overflow at fib(%d)\n", i + 2);
            exit(0);
        }
    }
    print_bignat(ring[n%3]);
}
```

另外，假设在文件 bignat.c 中定义了如下两个函数 plus 和 from_int（定义的具体内容略）：

```c
int plus (int n, unsigned int *a, unsigned int *b, unsigned int *c);
void from_int (int n, unsigned int k, unsigned int *a);
```

(1) 请在下表中给出程序中相关符号的属性（局部或全局、强符号或弱符号），其中不确定的请在表格中填 X。

源文件	符号名	局部/全局	强符号/弱符号
main.c	fib		
	main		
foo.c	ring		
	fib		
	plus		

(2) 假设文件 bignat.c 被编译为一个静态库 bignat.a，对于如下的 gcc 调用，会得到什么样的结果？请用下面的 A、B、C 三个选项填空。

A. 编译和链接都正确
B. 链接失败（原因是包含未定义的引用）
C. 链接失败（原因是包含重复定义）

命令	结果（填选项）
gcc -o fib main.c fib.c bignat.a	
gcc -o fib bignat.a main.c fib.c	
gcc -o fib fib.c main.c bignat.a	

(3) 若在 fib.c 中，程序员在声明变量 ring 时把它写成了 static int ring[3][N];，会不会影响这些文件的编译、链接和运行结果？说明理由。

11. (2013) 下列程序运行的结果是什么？

```
1   /* main.c */
2   int i = 0;
3   int main() {
4       foo();
5       return 0;
6   }
7
8   /* foo.c */
9   int i = 1;
10  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

12. (2013) 考虑如下两个程序 fact1.c 和 fact.c：

```
1   /* fact1.c */
2   #define MAXNUM 12
3   int table[MAXNUM];
4   int fact(int n);
```

```c
int main(int argc, char **argv) {
    int n;
    table[0] = 0;
    table[1] = 1;
    if (argc == 1) {
        printf("Error: missing argument\n");
        exit(0);
    }
    argv++;
    if (sscanf(*argv, "%d", &n) != 1 || n < 0 || n >= MAXNUM) {
        printf ("Error: %s not an int or out of range\n", *argv);
        exit (0);
    }
    printf("fact(%d) = %d\n", n, fact(n));
}

/* fact2.c */
int *table;
int fact(int n) {
    static int num = 2;
    if (n >= num) {
        int i = num;
        while (i <= n) {
            table[i] = table[i-1] * i;
            i++;
        }
        num = i;
    }
    return table[n];
}
```

(1) 请在下表中给出程序中相关符号的属性（局部符号、强全局符号或弱全局符号），并给出它们在链接后 ELF 文件中所属的 section 名，其中不确定的请在表格中填 X。

源文件	符号名	属性	ELF Section
fact1.c	table		
	fact		
	num		
fact2.c	table		
	fact		
	num		

(2) 对上述两个文件进行链接之后，会对每个符号进行解析。请给出链接后下列符号分别被定义的模块（fact1.c/fact2.c）。

符号	定义模块
table	
fact	
num	

(3) 使用 `gcc -o fact fact1.c fact2.c` 编译之后得到的可执行文件是否能够正确执行？说明理由。

12 异常控制流与系统 I/O

要点

- ▷ 知道控制流和异常控制流的定义，知道异常及其处理是由软硬件结合实现的。
- ▷ 熟悉异常处理的整个过程，包括异常表等数据结构，会以某个异常为例，复述整个异常处理的过程。会区分异常和过程调用的区别。
- ▷ 熟练掌握异常的四种类别及其特征、常见例子，熟悉异常结束后的返回机制。了解 x86-64 系统调用的基本范式。
- ▷ 熟练掌握进程的概念，理解上下文、上下文切换、调度、抢占等概念及其内容，知道内核模式和用户模式的区别。熟悉并行和并发的相同和不同。
- ▷ 会使用 Linux 系统下提供的进程管理函数，记忆其参数填法、返回值和行为（特别是 `waitpid`），熟练掌握 `fork` 函数，会通过画进程图和拓扑排序的方式进行分析。
- ▷ 知道 Linux 的常见信号，熟悉信号处理的过程及其细节（例如 `SIGKILL`，`SIGSTOP` 的默认行为不能修改、待处理信号至多只有一个、进程可以发信号给自己等），知道发送信号的机制和多种 API。
- ▷ 理解信号处理例程和主“程序”的并发性质，熟悉编写信号处理程序中需要注意的五个要点并理解其基本原理（软中断和调度问题）。信号不能用来计数。
- ▷ 了解非本地跳转的基本机制，知道 `setjmp` 的返回值不能赋值等。
- ▷ 知道 Linux 系统的文件模型和文件类型，会用各种文件读写系统调用编写程序。熟悉文件元数据的结构和读写方法。了解 RIO 包的用法和目的。
- ▷ 熟练掌握描述符表（理解与打开文件的关系）、打开文件表、v-node 表、引用计数的概念，会结合异常控制流（`fork`）和重定向（`dup2`，`dup`）的知识分析程序。
- ▷ 知道标准输出和底层系统调用的区别，熟练掌握输出缓冲的机制和 `fflush`。

1. 区分异常的四种类型：填写下表（打勾）。

异常的种类	是否同步	可能的返回行为		
		重复当前指令	执行下一条指令	结束进程运行
中断（interrupt）				
陷阱（trap）				
故障（fault）				
终止（abort）				

2. 下列行为分别触发什么类型的异常？

(1) 执行指令 `mov $57, %eax; syscall`

(2) 程序执行过程中，发现它所使用的物理内存损坏了

(3) 程序执行过程中，试图往 `main` 函数的内存中写入数据

(4) 按下键盘

(5) 磁盘读出了一块数据

(6) 用 `read` 函数发起磁盘读

(7) 用户程序执行了指令 `lgdt`，但是这个指令只能在内核模式下执行

3. 在下面的程序中，父进程和子进程的输出分别是什么？

```
int main() {
    int a = 9;
    if (Fork() == 0)
        printf("p1: a=%d\n", a--);
    printf("p2: a=%d\n", a++);
    exit(0);
}
```

4. 阅读下述程序。

```
int main() {
    char c = 'A';
    printf("%c", c); fflush(stdout);
    if (fork() == 0) {
        c++;
        printf("%c", c); fflush(stdout);
    } else {
        printf("%c", c); fflush(stdout);
        fork();
    }
    c++;
    printf("%c", c); fflush(stdout);
    return 0;
}
```

假设系统调用成功，所有子进程都正常运行。判断下列哪些输出是可能的：AABBBC、ABCABB、ABBABC、AACBBC、ABABCB、ABCBAB。

5. 阅读下述程序。

```
int main() {
    int child_status;
    char c = 'A';
    printf("%c", c); fflush(stdout);
```

```
        c++;
        if (fork() == 0) {
            printf("%c", c); fflush(stdout);
            c++;
            fork();
        } else {
            printf("%c", c); fflush(stdout);
            c += 2;
            wait(&child_status);
        }
        printf("%c", c); fflush(stdout);
        exit(0);
    }
```

假设系统调用成功，所有子进程都正常运行。判断下列哪些输出是可能的：ABBCCD、ABBCDC、ABBDCC、ABDBCC、ABCDBC、ABCDCB。

6. 阅读下述程序。

```
    void handler() {
        printf("D\n");
        return;
    }
    int main() {
        signal(SIGCHLD, handler);
        if (fork() > 0) {
            printf("A\n");
        } else {
            printf("B\n");
        }
        printf("C\n");
        exit(0);
    }
```

假设系统调用成功，所有子进程都正常运行。判断下列哪些输出是可能的：ACBC、ABCCD、ACBDC、ABDCC、BCDAC、ABCC。

7. 在某年的 ICS 课堂上，老师给同学布置了一个作业，在 Linux 上写出一份代码，使得运行它以后可输出系统能创建的进程的最大数目。下面是几位同学的答案。

(1) Alice 同学的答案是：

```
    int main() {
        int pid;
        int count = 1;
        while ((pid = fork()) != 0) {
            // parent process
            count++;
```

```
        }
        if (pid == 0) {
            // child process
            exit(0);
        }
        printf("max = %d", count);
    }
```

这段代码不能够正确运行，原因在于对 `fork` 的返回值处理得不正确。请修改至多一处代码，使得程序正确运行。

(2) Bob 同学对 Alice 同学修改过后的正确代码发出了疑问。Bob 同学认为，由于进程的调度时间和顺序都是不确定的，因此有的时候会调度到子进程，子进程执行 `exit(0)` 以后就结束了，因此父进程可以创建更多的进程，所以 Alice 的代码输出的答案大于真实上限。请问，Bob 的说法正确吗？如果正确，请指出 Alice 应当如何修改代码，以避免 Bob 提到的问题。如果 Bob 的说法错误，请指出他错在何处。

(3) Carol 同学的答案是：

```
    int main() {
        int pid;
        int count = 1;
        while ((pid = fork()) > 0) {
            // parent process
            count++;
        }
        if (pid == 0) {
            // child process
            while(1)
                sleep(1);
        }
        printf("max = %d", count);
    }
```

连续运行 Carol 同学的答案两次，发现结果分别如下：

```
$ ./test
max = 1795
$ ./test
max = 1
```

a) 解释为什么会发生这种情况。

b) 为了解决第一次运行后的遗留问题，可以不修改代码，而直接在 Linux 终端中使用指令来解决。假设在第一次程序运行完以后，使用 `ps` 指令，得到的列表前几项如下：

```
$ ./test
max = 1795
$ ps
22698 pts/0    00:00:00 bash
```

```
    22725 pts/0      00:00:00 test
    22726 pts/0      00:00:00 test
    22727 pts/0      00:00:00 test
    ............
```

假设 test 程序开始运行后，没有任何新的进程被创建，并且所有进程号均按照顺序递增、逐个地分配。于是，输入下列的指令，就可以让第二次运行得到正确的结果。其中 -9 表示 SIGKILL。请在横线上填入正确的值。

```
    $ kill -9 _________
```

(4) Dave 同学修改了 Carol 同学的答案。他将 Carol 的最后一句 printf 改为如下代码：

```
1   if (pid < 0) {
2       printf("max = %d", count);
3       kill(0, SIGKILL);
4   }
```

这段代码有时无法输出任何答案。Dave 想了一想，将 printf 中的字符串做了些修改，这样这段代码就能正确运行了。他修改了什么？

8. 以下问题中我们均假设缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区；程序运行过程中的所有系统调用均成功。

(1) 考虑下面的程序。

```
1   int main() {
2       printf("a");
3       fork();
4       printf("b");
5       fork();
6       printf("c");
7       return 0;
8   }
```

写出它的一个可能的输出 _____，这个输出是否是唯一可能的？说明理由。

(2) 考虑下面的程序。

```
1   int main() {
2       write(1, "a", 1);
3       fork();
4       write(1, "b", 1);
5       fork();
6       write(1, "c", 1);
7       return 0;
8   }
```

以上程序的输出中有 _____ 个 a，_____ 个 b，_____ 个 c；其第一个输出的字符一定是 _____。

(3) 考虑下面的程序。

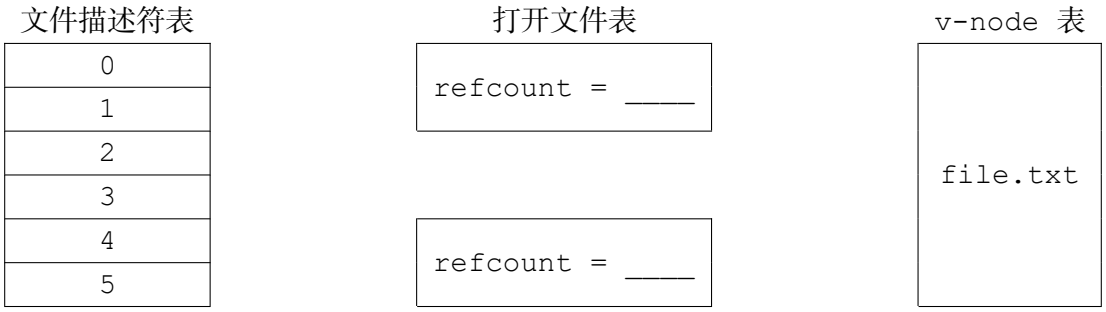
```c
int main() {
    printf("a");
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

以上程序的输出中有 _____ 个 a，_____ 个 b，_____ 个 c；其第一个输出的字符一定是 _____。

9. 假设磁盘上有空文件 file.txt，以下程序运行过程中的所有系统调用均成功。

```c
int main() {
    int fd1 = open("file.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    int fd2 = open("file.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    printf("%d %d\n", fd1, fd2);
    write(fd1, "123", 3); write(fd2, "45", 2);
    close(fd1); close(fd2);
    return 0;
}
```

(1) 程序关闭 fd1 前，补全下面的 Linux 三级表结构；填写打开文件表中的 refcnt 值。



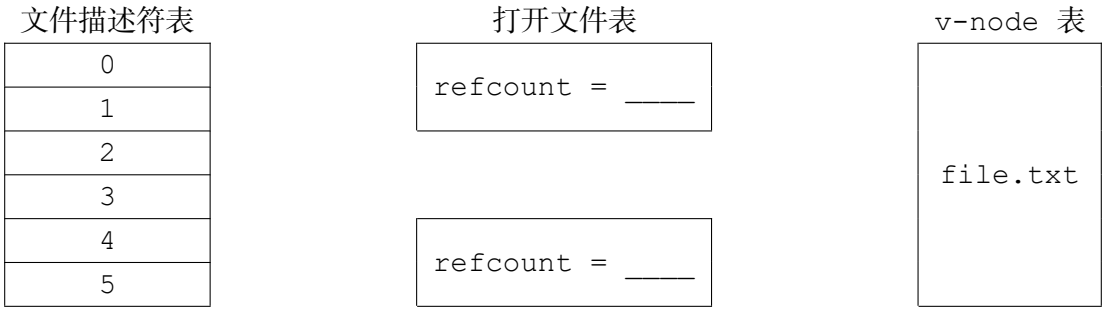
(2) 程序结束时，标准输出上的内容是 _____，file.txt 中的内容是 _____。

10. 假设磁盘上有空文件 file.txt，以下程序运行过程中的所有系统调用均成功。

```c
int main() {
    int fd1 = open("file.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    int fd2 = open("file.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    dup2(fd2, fd1);
    printf("%d %d\n", fd1, fd2);
    write(fd1, "123", 3); write(fd2, "45", 2);
    close(fd1); close(fd2);
```

```
8         return 0;
9     }
```

(1) 程序关闭 fd1 前，补全下面的 Linux 三级表结构；填写打开文件表中的 refcnt 值。

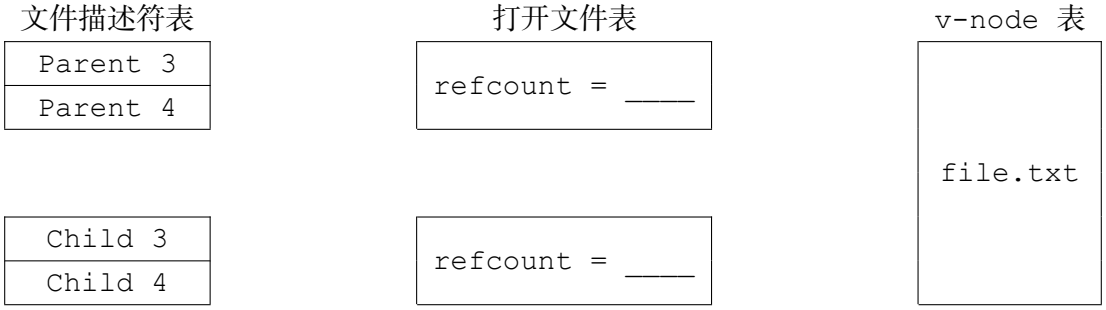


(2) 程序结束时，标准输出上的内容是 _____，file.txt 中的内容是 _____。

11. 假设磁盘上有空文件 file.txt，以下程序运行过程中的所有系统调用均成功；缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。

```
1     int main() {
2         pid_t pid;
3         int child_status;
4         int fd1 = open("file.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
5         if ((pid = fork()) > 0) {
6             printf("P:%d ", fd1);
7             write(fd1, "123", 3);
8             waitpid(pid, &child_status, 0);
9         } else {
10            printf("C:%d ", fd1);
11            write(fd1, "45", 2);
12        }
13        close(fd1);
14        return 0;
15    }
```

(1) 子进程关闭 fd1 前，补全下面的 Linux 三级表结构；填写打开文件表中的 refcnt 值。



(2) 程序结束时，标准输出上的内容是 _____，`file.txt` 中的内容是 _____。（都写出所有可能答案）

13 异常控制流与系统 I/O—往年考题

1. (2018) 关于进程，以下说法正确的是：

A. 没有设置模式位时，进程运行在用户模式中，允许执行特权指令，例如发起 I/O 操作。

B. 调用 `waitpid(-1, NULL, WNOHANG & WUNTRACED)` 会立即返回：如果调用进程的所有子进程都没有被停止或终止，则返回 `0`；如果有停止或终止的子进程，则返回其中一个的 `PID`。

C. `execve` 函数的第三个参数 `envp` 指向一个以 `null` 结尾的指针数组，其中每一个指针指向一个形如“`name=value`”的环境变量字符串。

D. 进程可以通过使用 `signal` 函数修改和信号相关联的默认行为，唯一的例外是 `SIGKILL`，它的默认行为是不能修改的。

2. (2018) 假设某进程恰有五个已打开的文件描述符 0~4，分别引用五个不同文件，尝试运行以下代码：`dup2(3, 2); dup2(0, 3); dup2(1, 10); dup2(10, 4); dup2(4, 0);`。关于得到的结果，说法正确的是：

A. 运行正常完成，现在有四个描述符引用同一个文件。

B. 运行正常完成，现在进程共引用四个不同的文件。

C. 由于试图从一个未打开的描述符进行复制，发生错误。

D. 由于试图向一个未打开的描述符进行复制，发生错误。

3. (2018) Bob 是一名刚刚学完异常的同学，他希望通过配合 `kill` 和 `signal` 的使用，能让两个进程向同一个文件中交替地打印出字符。可惜他的 tshlab 做得不过关，导致他写的这个程序有各种问题。

```
1   #include "csapp.h"
2   #define MAXN 6
3   int parentPID = 0;
4   int childPID = 0;
5   int count = 1;
6   int fd1 = 1;
7   void handler1() {
8       if (count > MAXN)
9           return;
10      for (int i = 0; i < count; i++)
11          write(fd1, "+", 1);
12      __________X__________
13      kill(parentPID, SIGUSR2);
```

```
14      }
15      void handler2() {
16          if (count > MAXN)
17              return;
18          for (int i = 0; i < count; i++)
19              write(fd1, "-", 1);
20          __________Y__________
21          kill(childPID, SIGUSR1);
22      }
23
24      int main() {
25          signal(SIGUSR1, handler1);
26          signal(SIGUSR2, handler2);
27          parentPID = getpid();
28          childPID = fork();
29          fd1 = open("file.txt", O_RDWR);
30          if (childPID) {
31              __________Z__________
32              kill(childPID, SIGUSR1);
33          }
34          exit(0);
35      }
```

假设程序能在任意时刻被系统打断、调度，并且调度的时间切片大小是不确定的，可以充分长。在每次程序执行前，`file.txt` 是一个已经存在的空文件。

(1) 此时，`X` 处语句和 `Y` 处语句都是 `count++;`，`Z` 处语句是空语句。Alice 测试该代码，发现有时 `file.txt` 中没有任何输出。请解释原因。

提示〉考虑第 28 行 `fork` 以后，下一次被调度的进程，并从这个角度回答本题。同时要求简述解决方案。

(2) Bob 根据 Alice 的反馈，在某两行之间加了若干代码，修复了前述的问题。当 `X` 处代码和 `Y` 处代码都是 `count++;`，`Z` 处为空时，Bob 期望 `file.txt` 中的输出是：

```
+-++--++++--++++--++++--++++--++++------
```

可 Alice 测评 Bob 的程序的时候，却发现有时 Bob 的程序在 `file.txt` 中的输出是：

```
+-+--+---+----+------
```

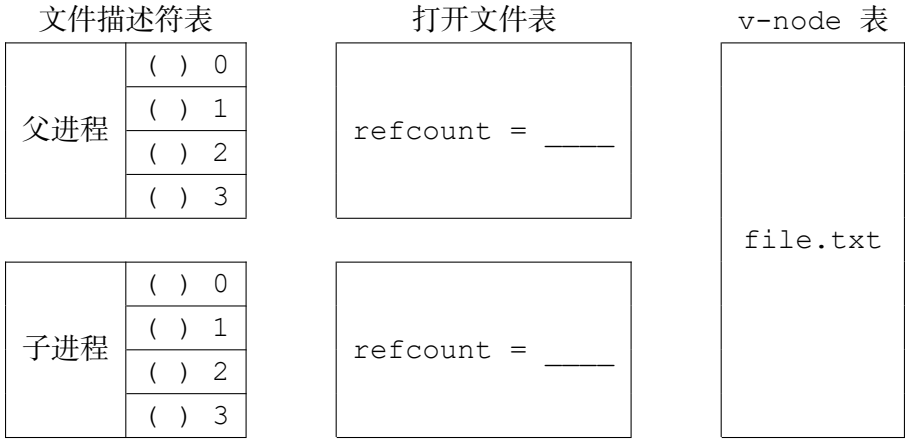
而与此同时，终端上出现了如下的输出：`+`。现在要找出程序的问题，所以要对上面的输出过程进行分析，请按以下要求填写后面的表格：

(i) 在描述符表一栏中，勾选该进程当前 `fd1` 的值。

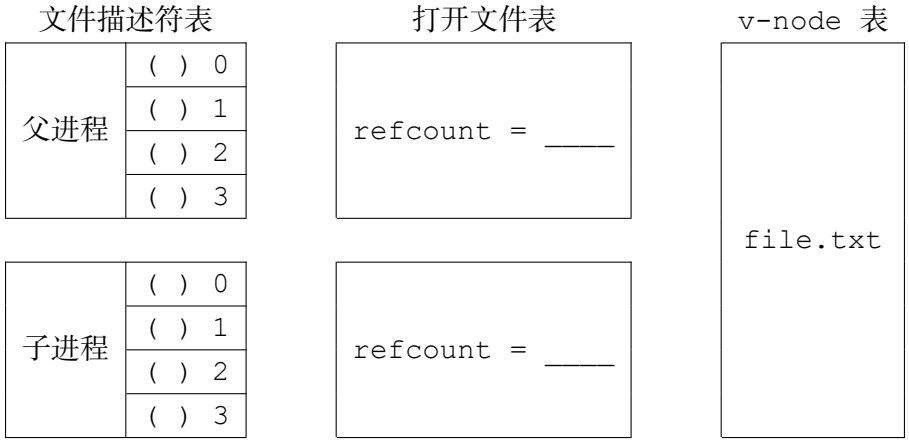
(ii) 在打开文件表一栏中，填写该项的 `refcnt`。如果某一项不存在，请在括号中写 `0`。

(iii) 画出描述符表到打开文件表的表项指向关系。不需要画关于标准输入/标准输出/标准错误的箭头。评分时不对箭头评分，请务必保证 (i)、(ii) 两步的解答与箭头连接的情况相一致。

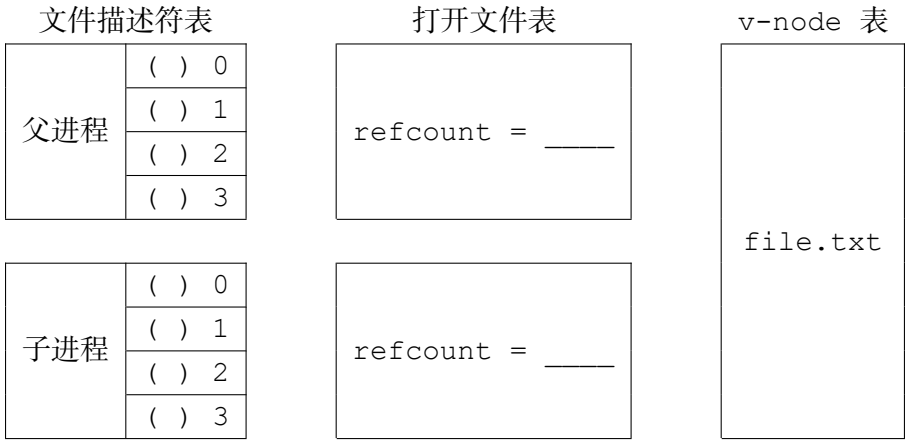
a) 当程序第一次在终端上输出 `+` 的瞬间，Linux 三级表的结构如下，请补全：



b) 当程序第一次在 `file.txt` 中输出 + 的瞬间，Linux 三级表的结构如下，请补全：



c) 如果要产生 Bob 预期的输出，Linux 三级表的关系应当如下，请补全：



(3) 对于上一问的错误代码，如果终端上输出的是 `+++`，那么 `file.txt` 中的内容是什么？
答：_____。

(4) Bob 修复了 (2) 的问题，使得代码能够产生预期的输出。现在，Bob 又希望自己的代码最终输出的是 `+--++++----++++++------`，为此，他对 `X`，`Y`，`Z` 处作了如下的修改。`X`，`Y` 处语句已做如下填写，请补上 `Z` 处语句。

▷ X 处填写为 `count += 2;`。
▷ Y 处填写为 `count += 2;`。
▷ Z 处填写为 _____。

4. (2016) 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

5. (2016) 考虑以下代码，假设 `result.txt` 中的初始内容为 `666666`。

```
char *str1 = "6666";
char *str2 = "2333";
char *str3 = "hhhh";
int fd1, fd2, fd3, i;
fd1 = open("result.txt", O_RDWR);
fd2 = open("result.txt", O_RDWR);
dup2(fd1, fd2);
for (i = 0; i < 5; ++i) {
    fd3 = open("result.txt", O_RDWR);
    write(fd1, str1, 4);
    write(fd2, str2, 4);
    write(fd3, str3, 4);
    close(fd3);
}
close(fd1); close(fd2);
```

假设所有系统调用均成功，则这段代码执行结束后，`result.txt` 的内容中有（ ）个 `6`。

A. 6
B. 16
C. 20
D. 22

6. (2016) 关于 I/O 操作，以下说法中正确的是：

A. 由于 RIO 包的健壮性，所以 RIO 中的函数都可以交叉调用。
B. 成功调用 `open` 函数后，一定返回一个不小于 3 的文件描述符。
C. 调用 Unix I/O 开销较大，标准 I/O 库使用缓冲区来加快 I/O 的速度。
D. 和描述符表一样，每个进程拥有独立的打开文件表。

7. (2016) 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;
```

```
4         pid_1 = fork();
5         if (pid_1 == 0) {
6             pid_2 = fork();
7             if(pid_2 != 0) {
8                 wait(pid_2, NULL, 0);
9                 printf("B");
10            }
11            printf("F");
12            exit(0);
13        } else {
14            __________A__________
15            __________B__________
16            wait(pid_1, NULL, 0);
17            pid_2 = fork();
18            if (pid_2 == 0) {
19                printf("D");
20                cnt -= 1;
21            }
22            if(cnt == 0)
23                printf("E");
24            else
25                printf("G");
26            exit(0);
27        }
28    }
```

(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 位置的代码分别为 `printf("C");`、`exit(0);`，列出所有可能的输出结果：_____。

8. (2016) 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，且每条语句都是原子动作）：

```
1     pid_t pid;
2     int even = 0;
3     int counter1 = 0;
4     int counter2 = 1;
5     void handler1(int sig) {
6         if (even % 2 == 0) {
7             printf("%d\n", counter1);
8             counter1 = _____________________;
9         } else {
10            printf("%d\n", counter2);
11            counter2 = _____________________;
12            even = even + _____________________;
13        }
```

```
14      }
15      void handler2(int sig) {
16          if (_____________________) {
17              counter1 = even * even;
18          } else {
19              counter2 = even * even;
20          }
21      }
22      int main() {
23          signal(SIGUSR1, handler1);
24          signal(SIGUSR2, handler2);
25
26          if ((pid = fork()) == 0) {
27              while (1) {};
28          }
29          while (even < 30) {
30              kill(pid, _____________________);
31              sleep(1);
32              kill(pid, _____________________);
33              sleep(1);
34              even = even + _____________________;
35          }
36          kill(pid, SIGKILL);
37          exit(0);
38      }
```

完成程序，使得程序在输出的数字为以下 Q 队列的前 30 项，Q 队列定义为

$$Q_0 = 0, \quad Q_1 = 1, \quad Q_{n+1} = \begin{cases} Q_n + 1 & (2 \mid n), \\ 2Q_n & (2 \nmid n), \end{cases} \qquad (n \in \mathbb{N}).$$

注意：若某个位置中的程序内容对本次程序执行结果没有影响，请在相应位置填写“无关”。

9. (2015) 设一段程序中阻塞了 SIGCHLD 和 SIGUSR1 信号。接下来，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

10. (2015) 学完本课程后，几位同学聚在一起讨论有关异常的话题，请问下面哪一个说法有误？

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。

11. (2015) 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. (2015) 在系统调用成功的情况下，下面哪个是以下程序可能的输出？

```
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. `AABB`

D. `AA`

13. (2015) 以下关于 Unix I/O 的说法正确的是：

A. 从网络套接字（socket）读取内容时，可以通过反复读的方式处理不足值问题，直到读完所需要的数量或遇到 `EOF` 为止。

B. 以 `O_RDWR` 方式打开文件后，文件会有两个指针，分别记录读文件的当前位置和写文件的当前位置。

C. 用 `read` 函数直接读取控制台输入的文本行，会自动在行末追加 `\0` 字符。

D. 使用 `dup2(4, 1)` 成功进行重定向后执行 `close(4)`，会导致 1 号文件描述符也不可用。

提示 `O_RDWR` 表示文件可读可写。`dup2(oldfd, newfd)` 表示将 `oldfd` 重定向给 `newfd`。

14. (2015) 考虑如下程序代码：

```
#include <stdio.h>
#include "csapp.h"
int main() {
    printf("2");
    if (Fork()) {
        printf("33");
```

```
            Write(STDOUT_FILENO, "lol", 3);
        } else {
            Sleep(1);
            printf("233");
            Write(STDOUT_FILENO, "hhhh", 4);
        }
        fflush(stdout);
        return 0;
    }
```

编译后运行程序，程序正常退出。那么程序的输出最可能是：

A. 233lol233hhhh
B. lol233hhhh2233
C. 233lol2233hhhh
D. 2lol33hhhh233

15. (2015) 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。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);
}

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);
```

```
28              printf("P");
29          }
30          return 0;
31      }
```

16. (2014) 在系统调用成功的情况下，下列代码会输出几个 hello?

```
1   void doit() {
2       if (fork() == 0) {
3           printf("hello\n");
4           fork();
5       }
6       return;
7   }
8   int main() {
9       doit();
10      printf("hello\n");
11      exit(0);
12  }
```

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

17. (2014) 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。

18. (2014) 下列这段代码的输出不可能是：

```
1   void handler() {
2       printf("h");
3   }
4   int main() {
5       signal(SIGCHLD, handler) ;
6       if (fork() == 0) {
7           printf("a") ;
8       } else {
9           printf("b") ;
10      }
11      printf("c") ;
12      exit(0);
13  }
```

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

19. (2014) 设文本文件 ICS.txt 中包含 3000 个字符，考虑如下代码段：

```
int main(int argc, char** argv) {
    int fd = open("ICS.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    write(fd, "ICS", 3);
    char buf[128];
    int i;
    for (i = 0; i < 10; i++) {
        int fd1 = open("ICS.txt", O_RDWR);
        int fd2 = dup(fd1);
        int cnt = read(fd1, buf, 128);
        write(fd2, buf, cnt);
    }
    return 0;
}
```

上述代码执行完后，ICS.txt 中包含多少个字符（假设所有系统调用都成功）?

A. 3
B. 256
C. 3000
D. 3072

20. (2014) 下列系统 I/O 的说法中，正确的是：

A. C 语言中的标准 I/O 函数在不同操作系统中的实现代码一样。
B. 对于同一个文件描述符，混用 RIO 包中的 rio_readnb 和 rio_readn 两个函数不会造成问题。
C. C 语言中的标准 I/O 函数是异步线程安全的。
D. 使用 I/O 缓冲区可以减少系统调用的次数，从而加快 I/O 的速度。

21. (2014) 以下程序运行时系统调用均正确执行，buffer.txt 的初始内容为 pekinguniv。请给出代码运行后打印输出的结果，并给出程序运行结束后 buffer.txt 文件的内容。

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    char c;
    int file1 = open("buffer.txt", O_RDWR);
    int file2;
```

```
10
11          read(file1, &c, 1);
12          file2 = dup(file1);
13          write(file2, &c, 1);
14          printf("1 = %c\n", c);
15
16          int pid = fork() ;
17          if (pid == 0) {
18              read(file1, &c, 1);
19              write(file2, &c, 1);
20              printf("2 = %c\n", c);
21              read(file1, &c, 1);
22              printf("3 = %c\n", c);
23              close(file1);
24              exit(0);
25          } else {
26              waitpid(pid, NULL, 0);
27              close(file2);
28              dup2(file1, file2);
29              read(file2, &c, 1);
30              write(file2, &c, 1);
31              printf("4 = %c\n", c);
32          }
33          return 0;
34      }
```

22. (2014) 某程序员实现了一个自己的 `sleep` 函数，请分析该代码存在哪些问题。

```
1       #include <signal.h>
2       #include <unistd.h>
3       static void sig_alrm(int signo) {
4           /* nothing to do, just return to wake up the pause */
5       }
6
7       unsigned int sleep(unsigned int seconds) {
8           if (signal(SIGALRM, sig_alrm) == SIG_ERR)
9               return seconds;
10
11          alarm(seconds); /* start the timer */
12          pause(); /* next caught signal wakes us up */
13          return alarm(0); /* turn off timer, return unslept time */
14      }
```

23. (2014) 请阅读下面的代码：

```
1       int main(int argc, char** argv) {
```

```
2         int fd1 = open("ICS.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
3
4         write(fd1, "abc", 3);
5
6         int fd2 = fd1;
7         int fd3 = dup(fd2);
8         int fd4 = open("ICS.txt", O_APPEND | O_RDWR);
9         write(fd2, "defghi", 6);
10        write(fd4, "xyz", 3);
11
12        int fd5 = fd4;
13        dup2(fd3, fd5);
14        write(fd4, "pqr", 3);
15
16        close(fd1);
17        return 0;
18    }
```

(1) 设初始时 ICS.txt 文件不存在。程序执行时，所有的系统调用均会成功，所有表项均会从小到大依次分配，描述符表一开始被占用掉前 3 个表项。请填写在第 16 行代码刚刚执行完之后，下面的打开文件表和 v-node 表中表项的部分值，并画出表项之间的指向关系。对于已经释放的打开文件表表项，请填写释放前那一刻的值和指向的 v-node 表表项。可以忽略多余的表项。

描述符表

...
3
4
5
6
7

打开文件表

pos	refcount	是否被释放

v-node 表

文件名

(2) 请填写在第 16 行代码刚刚执行完之后，下列变量的值：

fd1	fd2	fd3	fd4	fd5

(3) 请写出程序执行完之后，ICS.txt 文件中的内容。

24. (2013) 关于信号的叙述，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。

25. (2013) 下面关于非局部跳转的叙述，正确的是哪一个？

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`。

26. (2013) 考虑如下代码，假设 `result.txt` 的初始内容是 `123`。

```
1  int main(int argc, char** argv) {
2      int fd1 = open("result.txt", O_RDWR);
3      char str[] = "abc";
4      char c;
5      write(fd1, str, 1);
6      read(fd1, &c, 1);
7      write(fd1, &c, 1);
8      return 0;
9  }
```

在这段代码执行完毕之后，`result.txt` 的内容是什么？假设所有的系统调用都会成功。

A. `a22`

B. `a21`

C. `a13`

D. `abb`

27. (2013) 已知在适当定义 `fd1`, `fd2`, `str1`, `str2` 后，代码段

```
write(fd1, str1, strlen(str1)); write(fd2, str2, strlen(str2));
```

可以在原本为空的文件 `ICS.txt` 中写下字符串 `I love ICS!`。那么下面确实是“适当定义”的代码有（　　）。

(1)

```
int fd1 = open("ICS.txt", O_RDWR);
int fd2 = open("ICS.txt", O_RDWR);
char *str1 = "I love ";
char *str2 = "ICS!";
```

(2)

```
int fd1 = open("ICS.txt", O_RDWR);
int fd2 = dup(fd1);
char *str1 = "I love ";
char *str2 = "ICS!";
```

(3)

```
int fd1 = open("ICS.txt", O_RDWR);
int fd2 = open("ICS.txt", O_RDWR);
char *str1 = "I love ";
char *str2 = "I love ICS!";
```

(4)

```
int fd1 = open("ICS.txt", O_RDWR);
int fd2 = dup(fd1);
char *str1 = "I love ";
char *str2 = "I love ICS!";
```

A. (1)(4)
B. (2)(3)
C. (1)(2)(3)(4)
D. 以上都不正确

28. (2013) 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。

```
1  int main() {
2      printf("A\n");
3      if (fork() == 0) {
4          printf("B\n");
5      } else {
6          printf("C\n");
7          __________A__________
8      }
9      printf("D\n");
10     exit(0);
11 }
```

(1) 如果程序中的 A 位置的代码为空，列出所有可能的输出。

(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。

(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。

29. (2013) 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，且每条语句都是原子动作。

```
1  pid_t pid;
2  int even = 0; // 根据程序设计要求初始值可能不同
3  int counter1 = 0;
4  int counter2 = 1;
5  void handler1(int sig) {
6      if (even % 2 == 0) {
7          printf("%d\n", counter1);
8          counter1 = __________A__________;
9      } else {
10         printf("%d\n", counter2);
11         counter2 = __________B__________;
12     }
13     even = __________C__________;
14 }
```

```
15    void handler2(int sig) {
16        if (__________D__________) {
17            counter1 = even * even;
18        } else {
19            counter2 = even * even;
20        }
21    }
22    int main() {
23        signal(SIGUSR1, handler1);
24        signal(SIGUSR2, handler2);
25        if ((pid = fork()) == 0) {
26            while (1) {};
27        }
28        while (even < 20) {
29            kill(pid, __________E__________);
30            sleep(1);
31            kill(pid, __________F__________);
32            sleep(1);
33            even += 2;
34        }
35        kill(pid, SIGKILL);
36        exit(0);
37    }
```

(1) 完成程序，使得程序输出斐波那契数列的前 20 项，其中 $F_0 = 0, F_1 = 1, \ldots, F_n = F_{n-1} + F_{n-2}$。如果存在对本次程序执行结果没有影响的语句，请在相应位置填写“无关”。

A: _____

B: _____

C: _____

D: _____

E: _____

F: _____

(2) 完成程序，其中 A，B 处保持不变，使得程序可以分别输出前几个奇数或偶数的平方和。其中若要输出奇数的平方和，even 的初始值为 3；若要输出偶数的平方和，even 的初始值为 2。

A: _____

B: _____

C: _____

D: _____

14 虚拟内存

要点

- ▷ 理解虚拟内存的基本原理（是磁盘的缓存）和由硬件和软件共同实现的基本机制。
- ▷ 熟练掌握分页式虚拟内存中，虚拟地址的结构、页表的结构、页表条目一般要包括的各字段的含义。
- ▷ 结合高速缓存、TLB 熟练掌握地址翻译的完整过程，会根据页表的信息模拟虚拟地址到物理地址的转换。理解多级页表的设计目的，会根据系统信息计算多级页表的详细结构并进行地址翻译。
- ▷ 以 Core i7/Linux 系统为例，了解实际虚拟内存的实现。了解页表自映射的机制。
- ▷ 熟练掌握内存映射的概念，知道共享对象和私有对象的区别，理解 COW 机制并以 `fork` 函数为例掌握其具体过程。会使用 `mmap` 函数进行内存映射工作，知道其各参数和返回值的用法。
- ▷ 综合运用高速缓存、异常控制流、系统级 I/O 和虚拟内存的知识求解综合问题。

1. 在某一 64 位体系结构中，每页的大小为 4 KB，采用的是三级页表，每张页表占据 1 页，页表项长度为 8 字节。则虚拟地址的位数为 _____。如果要映射满 64 位的虚拟地址空间，可通过增加页表级数来解决，那么至少要增加到 _____ 级页表。已知这个体系结构支持多种页大小，最小的三个页大小分别是 4 KB、_____ MB、_____ GB。

2. Intel IA32 体系中，每页的大小为 4 KB，采用的是二级页表，每张页表占据一页，每个页表项（PTE、PDE）的长度均为 4 字节。支持的物理地址空间为 36 位。如果采用二级翻译，那么每个 PDE 条目格式如下（物理地址 35~32 位必定为 0）：

	31~12	11	10	9	8	7	6	5	4	3	2	1	0
PDE	页表的物理地址					0					U/S	R/W	V

每个 PTE 条目格式如下：

	31~12	11	10	9	8	7	6	5	4	3	2	1	0
PTE	虚拟页的物理地址					0					U/S	R/W	V

如果采用一级页表翻译（大页模式），那么每个 PDE 的条目如下：

	31~22	21~17	16~13	12~8	7	6	5	4	3	2	1	0
PDE	虚拟页的物理地址	0	物理地址的35~32 位	~	1					U/S	R/W	V

上表中的最低位均为第 0 位。部分位的意义如下：

▷ V 当前条目是否有效（指向的页在物理内存中）；

▷ R/W 指向的区域是否可写。只有两级页表均为 1 的时候，该虚拟内存地址才可以写。

▷ U/S 指向的区域用户程序是否可访问。只有两级页表均为 1 的时候，该虚拟内存地址才可以被用户程序访问。

某一时刻，一级页表的起始地址为 0x00C188000。部分物理内存中的数据如下：

物理地址	内容	物理地址	内容	物理地址	内容	物理地址	内容
00C188000	63	00C188001	A0	00C188002	67	00C188003	C0
00C188004	0D	00C188005	A0	00C188006	F0	00C188007	A5
00C188008	67	00C188009	A0	00C18800A	32	00C18800B	0D
00C1880C0	67	00C1880C1	30	00C1880C2	88	00C1880C3	C1
00C188300	E7	00C188301	00	00C188302	80	00C188303	9A
00C188C00	**65**	**00C1880C1**	**80**	**00C1880C2**	**18**	**00C1880C3**	**0C**
00D32A294	67	00D32A295	C0	00D32A296	83	00D32A297	67
00D32A298	C0	00D32A299	C0	00D32A29A	BB	00D32A29B	DC
00D32AA5C	67	00D32AA5D	C0	00D32AA5E	83	00D32AA5F	9A
00DA0C294	45	00DA0C295	82	00DA0C296	77	00DA0C297	67
00DA0C298	67	00DA0C299	83	00DA0C29A	29	00DA0C29B	44
00DA0CA5C	0	00DA0CA5D	9A	00DA0CA5E	88	00DA0CA5F	EF

不采用 TLB 加速翻译。

(1) 现在需要访问虚拟内存地址 0x00A97088。

a) 将该地址拆成 VPN1 + VPN2 + VPO 的形式。

b) 对应的 PDE 条目的物理地址是 _____，读出第二级页表的起始地址 _____，PTE 条目的起始地址为 _____，因此翻译得到的物理地址为 _____。

c) 用户模式能否访问该地址？能否写该地址？

(2) 现要访问虚拟内存地址 0x3003C088，则最终翻译得到的物理地址为 _____。

(3) 下列 IA32 汇编代码执行结束以后，%eax 的值是多少？假设一开始 %ebx 的值为 0x00A97088，%edx 的值为 0x3003C088。

```
movl   $0xC, (%ebx)
movl   $0x9, (%edx)
movl   (%ebx), %eax
xorl   (%edx), %eax
```

(4) 下列 IA32 汇编代码执行结束以后，`%eax` 的值是多少？

```
    movl    0xC0002A5C, %eax
```

重点关注加粗的内存。以此为启发，写出读出第一级页表中 `VPN1 = 2` 的条目的代码

```
    movl    _____________________, %eax
```

3. 有下列代码：

```
1   int main() {
2       pid_t pid;
3       int child_status;
4       long* f = mmap(NULL, 8, PROT_READ | PROT_WRITE,
5               _____X_____ | MAP_ANONYMOUS, -1, 0);
6       *f = 0;
7       if ((pid = fork()) > 0) {
8           waitpid(pid, &child_status, 0);
9           *f = *f + 1;
10          printf("Parent: %ld\n", *f);
11      } else {
12          *f = *f + 1;
13          printf("Child: %ld\n",*f);
14      }
15      return 0;
16  }
```

当 X 处为 `MAP_PRIVATE` 时，标准输出上的两个整数是什么？如果是 `MAP_SHARED` 呢？

4. 有下列 C 程序。其中 `sleep(3)` 是为了让 `fork` 以后子进程先运行。`hello.txt` 的初始内容为字符串 `ABCDEFG`，紧接着为 `\0`。Linux 采用写时复制（Copy-on-Write）技术。

```
1   char* f;
2   int count = 0, parent = 0, child = 0, done = 0;
3   void handler1() {
4       if (count >= 4) {
5           done = 1;
6           return;
7       }
8       f[count] = '0' + count;
9       count++;
10      kill(parent,SIGUSR2);
11  }
12  void handler2() {
13      _____Y_____
14      write(STDOUT_FILENO, f, 7);
15      write(STDOUT_FILENO, "\n", 1);
16      kill(child,SIGUSR1);
17  }
```

```
18  int main() {
19      signal(SIGUSR1, handler1);
20      signal(SIGUSR2, handler2);
21      int child_status;
22      parent = getpid();
23      int fd = open("hello.txt", O_RDWR);
24      if ((child = fork()) > 0) { // Parent
25          f = mmap(NULL, 8, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
26          sleep(3);
27          kill(child, SIGUSR1);
28          waitpid(child, &child_status, 0);
29      } else { // Child
30          f = mmap(NULL, 8, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
31          while (done == 0);
32      }
33      return 0;
34  }
```

(1) 若 Y 处为空，程序运行结束以后，标准输出上的内容是什么（四行）？hello.txt 中的内容是什么？

(2) 若 Y 处为 f[6] = 'X';，程序运行结束以后，标准输出上的内容是什么（四行）？hello.txt 中的内容是什么？

5. 有如下 C 程序：

```
1   int main() {
2       char* hello;
3       char* bye;
4       int fd1 = open("hello.txt", O_RDWR);
5       int fd2 = open("bye.txt", O_RDWR);
6       hello = mmap(NULL, 16, PROT_READ, MAP_SHARED, fd1, 0);
7       bye = mmap(NULL, 16, PROT_READ | PROT_WRITE,
8               MAP_SHARED, fd2, 0);
9       for (int i = 0; i < 8; i++)
10          bye[i] = toupper(hello[i]);
11      /************A**************/
12      munmap(hello, 16);
13      munmap(bye, 16);
14      return 0;
15  }
```

某次运行，该程序对应进程的 PID = 2333。此时代码运行到 A 处时，/proc/2333/maps 中的内容如下：

```
ADDRESS                   PERM    PATH
00400000-00401000         (1)     /home/larry/map/pstate
00600000-00601000         r--p    /home/larry/map/pstate
```

```
00601000-00602000          rw-p   /home/larry/map/pstate
7fb596fb5000-7fb59719c000  r-xp   /lib/x86_64-linux-gnu/libc-2.27.so
7fb59719c000-7fb59739c000  ---p   /lib/x86_64-linux-gnu/libc-2.27.so
7fb59739c000-7fb5973a0000  r--p   /lib/x86_64-linux-gnu/libc-2.27.so
7fb5973a0000-7fb5973a2000  rw-p   /lib/x86_64-linux-gnu/libc-2.27.so
7fb5973a2000-7fb5973a6000  rw-p
7fb5973a6000-7fb5973cd000  r-xp   /lib/x86_64-linux-gnu/(4)-2.27.so
7fb5975b1000-7fb5975b3000  rw-p
7fb5975cb000-7fb5975cc000  (2)    /home/larry/map/bye.txt
7fb5975cc000-7fb5975cd000  (3)    /home/larry/map/hello.txt
7fb5975cd000-7fb5975ce000  r--p   /lib/x86_64-linux-gnu/(4)-2.27.so
7fb5975ce000-7fb5975cf000  rw-p   /lib/x86_64-linux-gnu/(4)-2.27.so
7fb5975cf000-7fb5975d0000  rw-p
7ffe671ef000-7ffe67210000  rw-p   [(5)]
7ffe673cc000-7ffe673cf000  r—p    [vvar]
7ffe673cf000-7ffe673d1000  r-xp   [vdso]
ffffffffff600000-ffffffffff601000 r-xp [vsyscall]
```

上面内容中，PERM 下条目有四位。前三位是 r=readable、w=writeable、x=executable，如果是 - 则表示没有这一权限。第四位是 s=shared、p=private，表示内存映射是共享的还是私有的。

(1) 填写上面 (1)~(5) 空的内容。

(2) 7fb5973a0000 对应的页在页表中被标为了只读。对该页进行写操作会发生 SIGSEGV 吗？说明理由。

15 虚拟内存—往年考题

1. (2019) 考虑本课程介绍的 Intel x86-64 存储系统中的某一个进程。它从用户态切换至内核态时，TLB _____，高速缓存 _____；当发生进程上下文切换时，TLB _____，高速缓存 _____。

A. 不必刷新　不必刷新　需要刷新　不必刷新
B. 不必刷新　不必刷新　不必刷新　不必刷新
C. 需要刷新　不必刷新　需要刷新　需要刷新
D. 需要刷新　不必刷新　不必刷新　需要刷新

2. (2019) 已知某系统页面大小为 2 KB，页表项为 8 字节，采用多层分页策略映射 48 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？

3. (2019) 在一个具有 TLB 和高速缓存的系统中，假设地址翻译使用四级页表来进行，且不发生缺页异常，那么在 CPU 访问某个虚拟内存地址的过程中，至少会访问 _____ 次物理内存，至多会访问 _____ 次物理内存。

4. (2019, 有改动) 某 32 位机器有 28 位虚拟地址空间，物理地址为 24 位。采用二级页表，一级页表中有 64 个 PTE，二级页表中有 1024 个 PTE，页表页均按 4 KB 对齐。PTE 最高一位是有效位，页表项中低 12~23 位为物理页号（最低位为第 0 位）。系统中没有 TLB 和高速缓存。某人在该机器上执行了如下代码：

```
1    int *a = calloc(10000, sizeof(int));
2    int *b = a + 5000;
3    fork();
4    *b = 0x80C3F110;
```

设编译后 `b` 的值保存在寄存器中。

(1) 该机器页面大小为 _____ 字节，`VPN1` 长度为 _____，`VPN2` 长度为 _____。

(2) 已知父进程执行第四行代码时进行了若干次物理内存的读写操作，其中第一次读内存操作从地址 `0x67F0E8` 读出 `0x80AA32C4`，另外有两次将 `0x80C3F110` 写入内存的操作，写入的地址分别是 `0xC3F50D` 和 `0xAA3AD0`，但写入顺序并不清楚。据此回答（地址都用 16 进制表示）：程序结束瞬间变量 `b` 中保存的值是_____。在（父进程）解析 `*b` 的过程中，一级页表的起始地址为 _____；二级页表的起始地址为 _____；物理页面的起始地址为 _____。

5. (2018) 下列与虚拟内存有关的说法中哪些是不对的?

A. 操作系统为每个进程提供一个独立的页表，用于将其虚拟地址空间映射到物理地址空间。
B. MMU 使用页表进行地址翻译时，虚拟地址的虚拟页面偏移与物理地址的物理页面偏移是相同的。
C. 若某个进程的工作集大小超出了物理内存的大小，则可能出现抖动现象。
D. 动态内存分配管理中可以采用双向链表组织空闲块，使得首次适配的分配与释放的时间均和空闲块数量成线性关系。

6. (2018) 假定整型变量 `A` 的虚拟地址为 `0x12345cf0`，另一整型变量 `B` 的虚拟地址为 `0x12345d98`，假定一个页的长度为 `0x1000` 字节，那么 `A` 的物理地址数值和 `B` 的物理地址数值之关系应该为：

A. `A` 的物理地址数值始终大于 `B` 的物理地址数值
B. `A` 的物理地址数值始终小于 `B` 的物理地址数值
C. `A` 的物理地址数值和 `B` 的物理地址数值大小取决于动态内存分配策略
D. 无论如何都无法判定两个物理地址数值的大小关系

7. (2018) 虚拟内存中两层页表和单层页表相比，最主要的优势在于：

A. 能够达到更快的地址翻译速度
B. 能够提供更加精细的保护措施
C. 能够充分利用代码的空间局部性
D. 能够充分利用稀疏的内存使用模式

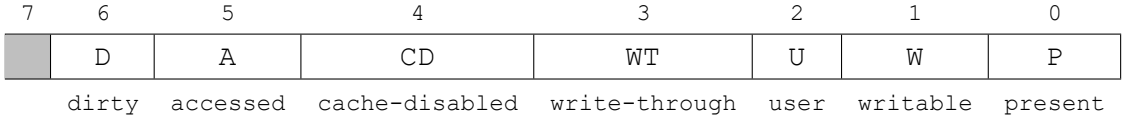
8. (2018) 回答下列问题。

(1) 在进行地址翻译的过程中，操作系统需要借助页表。考虑一个 32 位的系统，页大小是 4 KB，页表项的大小是 4 字节。如果不使用多级页表，则常驻内存的页表一共需要 _____ 页。

再考虑下图示出的物理内存分配情况。假设系统使用二级页表，则已经显示的区域的页表需要占据 _____ 页。

<table>
<tr><td>VP0</td><td rowspan="6">已分配页</td></tr>
<tr><td>...</td></tr>
<tr><td>VP1023</td></tr>
<tr><td>VP1024</td></tr>
<tr><td>...</td></tr>
<tr><td>VP2047</td></tr>
<tr><td>gap</td><td rowspan="2">未分配页</td></tr>
<tr><td>1023 个未分配页</td></tr>
<tr><td>VP10239</td><td rowspan="3">已分配页</td></tr>
<tr><td>...</td></tr>
<tr><td>VP11263</td></tr>
</table>

(2) IA32 体系采用小端法和二级页表。其中两级页表大小相同，页大小均为 4 KB，结构也相同。TLB 采用直接映射。TLB 和页表每一项的前 20 位为物理地址，后 7 位含义如下图所示。为简便起见，假设 TLB 和页表每一项的后 8~12 位都是 0 且不会被改变。例如后 7 位值为 0x27 则表示可读写。



当系统运行到某一时刻时，TLB 内容如下：

索引	TLB 标记	内容	有效位
0	0x04013	0x3312D027	1
1	0x01000	0x24833020	0
2	0x005AE	0x00055004	1
3	0x00402	0x24AEE020	0
4	0x0AA00	0x0005505C	0
5	0x0000A	0x29DEE000	1
6	0x1AE82	0x00A23027	1
7	0x28DFC	0x00023000	0

一级页表的基地址为 0x0C23B00，物理内存中的部分内容如下：

地址	内容	地址	内容	地址	内容	地址	内容
00023000	E0	00023001	BE	00023002	EF	00023003	BE
00023120	83	00023121	C8	00023122	FD	00023123	12
00023200	23	00023201	FD	00023202	BC	00023203	DE
00023320	33	00023321	29	00023322	E5	00023323	D2
0005545C	97	0005545D	C2	0005545E	7B	0005545F	45
00055464	97	00055465	D2	00055466	7B	00055467	45
0C23B020	27	0C23B021	EB	0C23B022	AE	0C23B023	24
0C23B040	27	0C23B041	40	0C23B042	DE	0C23B043	29
0C23B080	05	0C23B081	5D	0C23B082	05	0C23B083	00
2314D200	23	2314D201	12	2314D202	DC	2314D203	0F
2314D220	A9	2314D221	45	2314D222	13	2314D223	D2
29DE404C	27	29DE404D	42	29DE404E	BA	29DE404F	00
29DE4400	D0	29DE4401	5C	29DE4402	B4	29DE4403	2A

此刻，系统先后试图对两个已经缓存在 cache 中的内存地址进行写操作，请分析完成写之后系统的状态（写的地址和上表中的内存地址无交集），完成下面的填空。若不需要某次访问或者缺少所需信息，请填写 X。第一次向地址 0xD7416560 写入内容，TLB 索引为 _____，该项 TLB 内容为 _____，二级页表页表项地址

为 _____，物理地址为 _____。第二次向地址 0x0401369B 写入内容，TLB 索引为 _____，完成写之后该项 TLB 内容为 _____，二级页表页表项地址为 _____，物理地址为 _____。

(3) 本学期的 fork bomb 作业中，大家曾用 `fork()` 逼近系统的进程数量上限。下面有一个类似的程序，请仔细阅读程序并填空。

```
1   #define N 4
2   int main() {
3       volatile int pid, cnt = 1;
4       for (int i = 0; i < N; i++) {
5           if ((pid = fork()) > 0) {
6               cnt++;
7           }
8       }
9       while (wait(NULL) > 0);
10      return 0;
11  }
```

整个过程中，变量 cnt 最大值是 _____。假设所有的数据都已经在内存中，pid 和 cnt 在同一个物理页中。从第一个进程开始执行 for 语句起计算，此过程对 cnt 的操作至少会导致页表中发生 _____ 次如下事件：虚拟页所对应的物理页被修改。

9. (2016) 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
B. D
C. XD
D. U/S

10. (2016) 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

11. (2016) 考虑如下程序，其被编译成可执行文件 i16。

```
1   int main(int argc, char **argv){
2       int fd1, fd2;
3       char *cs;
4       struct stat stat;
5       int sz, i;
6
7       fd1 = Open(argv[1], O_RDWR, S_IRUSR | S_IWUSR);
8       Fstat(fd1, &stat);
9       sz = stat.st_size;
10      fd2 = Open(argv[2], O_RDWR | O_TRUNC | O_CREAT, \
```

```
11                          S_IRUSR | S_IWUSR);
12
13          cs = (char *) Mmap(NULL, sz, PROT_READ | PROT_WRITE, \
14                  __________X__________, fd1, 0);
15          if (Fork()) {
16              Wait(&i);
17              Write(fd2, cs, sz);
18          } else {
19              for (i = 0; i < sz; i++) {
20                  if (islower(cs[i]))
21                      cs[i] = (char) toupper(cs[i]);
22                  else if (isupper(cs[i]))
23              }
24                  cs[i] = (char) tolower(cs[i]);
25                  __________Y__________;
26          }
27          Munmap(cs, sz);
28          Close(fd1);
29          Close(fd2);
30      }
```

(1) 设文件 t1 内容为 ICSexam，根据 X/Y 处的内容，判断执行 ./i16 t1 t2 之后的结果。

a) X 为 MAP_PRIVATE，Y 为空：

A. t1 为 ICSexam，t2 为 ICSexam
B. t1 为 ICSexam，t2 为 icsEXAM
C. t1 为 icsEXAM，t2 为 icsEXAM
D. t1 为 icsEXAM，t2 为 ICSexam

b) X 为 MAP_SHARED，Y 为空：

A. t1 为 ICSexam，t2 为 ICSexam
B. t1 为 ICSexam，t2 为 icsEXAM
C. t1 为 icsEXAM，t2 为 icsEXAM
D. t1 为 icsEXAM，t2 为 ICSexam

c) X 为 MAP_PRIVATE，Y 为 Write(fd2, cs, sz)，t2 的内容：

A. ICSexam
B. icsEXAM
C. icsEXAMICSexam
D. icsEXAMicsEXAM

d) X 为 MAP_SHARED，Y 为 Write(fd2, cs, sz)，t2 的内容：

A. ICSexam
B. icsEXAM
C. icsEXAMICSexam
D. icsEXAMicsEXAM

(2) 设 X 为 MAP_SHARED。当子进程运行到 Y 处时，内存映射的内容如下（只保留了 /proc/pid/maps 中的三部分内容，分别为 address/perm/path）：

```
ADDRESS                          PERM   PATH
00400000-00405000                r-xp   /home/larry/i16/i16
00604000-00605000                r--p   /home/larry/i16/i16
00605000-00606000                __A__  /home/larry/i16/i16
7f5df1e61000-7f5df2020000        r-xp   /lib/x86_64-linux-gnu/libc-2.23.so
7f5df2020000-7f5df2220000        ---p   /lib/x86_64-linux-gnu/libc-2.23.so
7f5df2220000-7f5df2224000        r--p   /lib/x86_64-linux-gnu/libc-2.23.so
7f5df2224000-7f5df2226000        rw-p   /lib/x86_64-linux-gnu/libc-2.23.so
7f5df2226000-7f5df222a000        rw-p
7f5df222a000-7f5df2250000        r-xp   /lib/x86_64-linux-gnu/____B____
7f5df243f000-7f5df2442000        rw-p
7f5df244c000-7f5df244d000        rw-s   __________C__________
7f5df244d000-7f5df244f000        rw-p
7f5df244f000-7f5df2450000        r--p   /lib/x86_64-linux-gnu/____B____
7f5df2450000-7f5df2451000        rw-p   /lib/x86_64-linux-gnu/____B____
7f5df2451000-7f5df2452000        rw-p
7ffd4e35f000-7ffd4e380000        rw-p   [_____D_____]
7ffd4e3b0000-7ffd4e3b2000        r--p   [vvar]
7ffd4e3b2000-7ffd4e3b4000        r-xp   [vdso]
ffffffffff600000-ffffffffff601000 r-xp [vsyscall]
```

根据程序执行情况，填充上面的空白。

(3) 设 X 为 MAP_SHARED。当子进程运行到 Y 处时，回答有关问题。

a) 关于页表的描述，正确的是：

A. 由于 i16 本身的代码和数据只占用了低 32 位空间，所以这部分空间只需使用 2 级页表。

B. 所有虚拟地址空间都是 48 位地址空间，都需要使用完整的 4 级页表。

C. 当页表项无效时（P 位为 0），MMU 不会使用到其他位的内容。

D. 除读写权限外，需要在页表项中为 COW 机制提供另外的专门支持

b) 如果子进程在 Y 处访问 7f5df2224 这一页，则可能发生以下异常：

A. Page Fault 或 General Protection Fault

B. General Protection Fault 或 Segmentation Fault

C. Page Fault 或 Segmentation Fault

D. Page Fault 或 General Protection Fault 或 Segmentation Fault

c) 假设 TLB 有 64 个表项，4 路组相联，访问 7f5df2224 这一页时所对应的 TLBT 是：

A. 0111 1111 0101 1101 1111 0010 0010 0010

B. 0111 1111 0101 1101 1111 0010 0010 0010 01

C. 1111 0101 1101 1111 0010 0010 0010 0100

D. 11 1111 0101 1101 1111 0010 0010 0010 0100

d) 发生 TLB 命中时意味着：

A. 相应的页表项在 L1 Cache 里

B. 要访问的数据内容在 L1 Cache 里
C. 如果要访问的数据内容已在 L1 Cache 中，则该 Cache Line 的权限位中应具有相应的读写执行权限
D. 以上都不对

12. (2015) 虚拟内存管理方式可行性的基础是
A. 程序执行的离散性
B. 程序执行的顺序性
C. 程序执行的局部性
D. 程序执行的并发性

13. (2015) Intel 的 IA32 体系结构采用二级页表，称第一级页表为页目录，第二级页表为页表。页面的大小为 4 KB，页表项长 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

14. (2015) 已知某系统页面大小为 8 KB，页表项长 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_____层的分页策略。

15. (2015) 为了提升虚拟内存地址的转换效率，降低遍历两级页表结构所带来的地址转换开销，Intel 处理器针对 32 位模式引入了大页 TLB，即一个 TLB 项可以涵盖整个 4 MB 对齐的地址空间。只要设置页目录项的大页标志位，即可让 MMU 识别这是一个大页 PDE，并加载到大页 TLB 项中。大页 PDE 中记录的物理内存页面号必须是 4 MB 对齐的，而且整个连续的 4 MB 内存均可统一通过该大页 PDE 进行地址转换。

在 32 位的 Linux 系统中，为了方便访问物理内存，内核将地址 0~768 MB 间的物理内存映射到虚拟内存地址 3 G~3 GB + 768 MB 间，并通过大页 PDE 进行进行该区间的地址转换。任何 0~768 MB 的物理内存地址可以直接通过加 3 G（0xC0000000）的方式得到其虚拟内存地址。在内核中，除了该区间的内存外，其他地址的内存通常都通过普通的两级页表结构来进行地址转换。

假设在我们使用的处理器中有 2 个大页 TLB 项，其当前状态如下：

索引号	TLB 标记	页面号	有效位
0	0xC48	0x04800	1
1	0xC9C	0x09C00	1

有 4 个普通 TLB 项，当前的状态如下：

索引号	TLB 标记	页面号	有效位
0	0xF8034	0x04812	1
1	0xF8033	0x09812	1
2	0xF4427	0x12137	1
3	0xF44AE	0x17343	1

当前页活跃的目录页中的部分 PDE 的内容如下：

PDE 索引	页面号	大页位	存在位
786	0x04800	1	1
807	0x09C00	1	1
977	0x09C33	0	1
992	0x09078	0	1

注意，普通页面大小为 4 KB，并且 4 KB 对齐。每个页面的页面号为其页面起始物理地址除以 4096 得到。大页由连续 1024 个 4 KB 小页组成，且 4 MB 对齐。

(1) 分析下面的指令序列，

```
movl   $0xC48012024, %ebx
movl   $128, (%ebx)
movl   $0xF8034000, %ecx
movl   $36(%ecx), %eax
```

执行完上述指令后，%eax 寄存器中的内容是 _____；在执行上述指令过程中，共发生了 _____ 次 TLB miss；同时会发生 _____ 次 page fault。不确定的空填 X。

(2) 请判断下列页面号对应的页面中，哪些一定是页表页，哪些不是，哪些不确定：0x04800、0x09C33、0x09812。

(3) 下列虚拟地址中哪一个对应能将虚拟地址 0xF4427048 映射到物理地址 0x14321048 的页表项？

A. 0x09C33027

B. 0xC9C3309C

C. 0xC9C33027

D. 0x09C3309C

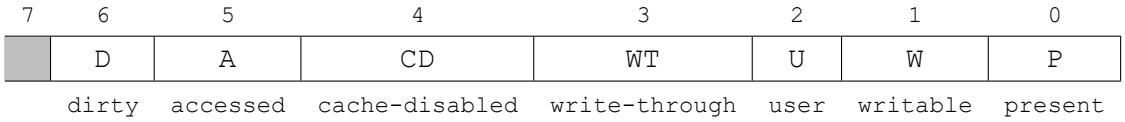
通过上述虚拟地址，利用 movl 指令修改对应的页表项，完成上述映射，在此过程中，是否会产生 TLB miss？

进一步，修改页表项后，是否可以立即直接使用下面的指令序列将物理内存地址 0x14321048 开始的一个 32 位整数清零？说明理由。

```
movl   $0xF4427048, %ebx
movl   $0, (%ebx)
```

16. (2014) IA32 体系采用小端法和二级页表。其中两级页表大小相同，页大小均为 4 KB，结构

也相同。TLB 采用直接映射。TLB 和页表每一项的前 20 位为物理地址，后 7 位含义如下图所示。为简便起见，假设 TLB 和页表每一项的后 8~12 位都是 0 且不会被改变。例如后 7 位值为 0x27 则表示可读写。



已知一级页表的地址为 0x0c23b000，物理内存中的部分内容如下图所示。

地址	内容	地址	内容	地址	内容	地址	内容
00023000	E0	00023001	BE	00023002	EF	00023003	BE
00023120	83	00023121	C8	00023122	FD	00023123	12
00023200	23	00023201	FD	00023202	BC	00023203	DE
00023320	33	00023321	29	00023322	E5	00023323	D2
00023FF8	29	00023FF9	FF	00023FFA	DE	00023FFB	BC
00055004	03	00055005	D0	00055006	74	00055007	89
0005545C	97	0005545D	C2	0005545E	7B	0005545F	45
00055460	97	00055461	D2	00055462	7B	00055463	45
00055464	97	00055465	E2	00055466	7B	00055467	45
0C23B020	55	0C23B021	EB	0C23B022	AE	0C23B023	24
0C23B040	55	0C23B041	AB	0C23B042	2A	0C23B043	01
0C23B080	05	0C23B081	5D	0C23B082	05	0C23B083	00
0C23B09D	05	0C23B09E	D3	0C23B09F	F2	0C23B0A0	0F
0C23B274	05	0C23B275	3D	0C23B276	02	0C23B277	00
0C23B9FC	25	0C23B9FD	D2	0C23B9FE	14	0C23B9FF	23
2314D200	23	2314D201	12	2314D202	DC	2314D203	0F
2314D220	A9	2314D221	45	2314D222	13	2314D223	D2
2314D4A0	BD	2314D4A1	BC	2314D4A2	88	2314D4A3	D3

TLB 的内容如下所示。

索引	TLB 标记	内容	有效位
0	0x08001	2314d220	1
1	0x01000	24aee520	0
2	0x005AE	00055004	0
3	0x016BA	0c23b09d	1
4	0x0AA00	0005545c	1
5	0x0000A	29dee500	0
6	0x5AE82	00023320	1
7	0x28DFC	00023000	1

(1) 某用户态进程试图写入虚拟地址 0x080016ba，该访问的最后结果是：

A. 该进程成功写入，未触发异常

B. 该进程触发了一个缺页异常

C. 该进程触发了一个非法访问异常

(2) 下面描述了上一问的具体访问过程，请填空。如果某个空在访问过程中已不可用，请填入 X。TLB 的索引为 _____，访问 _____（填“命中”或“不命中”）。一级页表表项地址为 _____；二级页表表项地址为 _____；最后得到的物理地址为 _____。

17. (2014) 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

A. TLB 未命中，Cache 未命中，Page 未命中

B. TLB 未命中，Cache 命中，Page 命中

C. TLB 命中，Cache 未命中，Page 命中

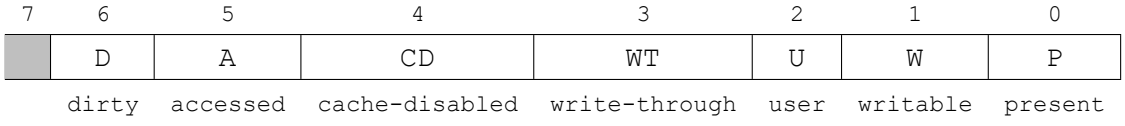
D. TLB 命中，Cache 命中，Page 未命中

18. (2013) 假设有一台 64 位的计算机的物理页块大小是 8 KB，采用三级页表进行虚拟地址寻址，则一般而言它的虚拟地址的 VPN 有 _____ 位。

19. (2013) 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。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 _____。

20. (2013) Intel 的 IA32 体系结构采用二级页表，称第一级页表为页目录，第二级页表为页表。其虚拟地址到物理地址的翻译方式如：先根据 CR3 寄存器找到页目录地址，然后依据偏移找到一个页目录项，页目录项的高 20 位为二级页表地址；在二级页表中根据偏移找到页表项，页表项中的高 20 位即为物理地址的高 20 位，将这 20 位与虚拟地址的低 12 位拼在一起形成完整的物理地址。

页目录和页表均有 1024 项，每一项为 4 字节，31~12 位为物理页号，11~7 为保留和系统使用的位，最后 7 位的含义如下：



页目录和页表由操作系统维护，通常只能在内核态下访问，为了给用户提供一个访问页表项和页目录项内容的接口，假设操作系统中已经执行过如下代码段：

```
1  #define UVPT 0xef400000
2  #define PDX(la) ((((unsigned int) (la)) >> 22) & 0x3FF)
3  ...
4  kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

其中 kern_pgdir 是操作系统维护的页目录数组，共 1024 项，每一项的类型为 unsigned int。

PADDR(kern_pgdir) 用于获得 kern_pgdir 的物理地址，页目录在物理内存中正好占一页，所以 kern_pgdir 的物理地址是 4 KB 对齐的。PTE_U 和 PTE_P 代表了这个页目录项的权限，即用户态可访问（只读）。可以看到，这条语句将页目录的第 PDX(UVPT) 项指向了页目录自身。

利用这一点，对于给定的虚拟地址 va，可以获得 va 对应的页目录项和页表项内容，分别对应于函数 get_pde 和 get_pte，请完成这两个函数：

```
1   #define UVPT 0xef400000
2   // get_pde(va) 获取虚拟地址 va 对应的一级页表 (页目录) 中的页目录项内容
3   unsigned int get_pde(unsigned int va) {
4       unsigned int pdx = (va >> ___________) & ___________;
5       unsigned int addr = UVPT + (___________) + pdx * 4;
6       return *((unsigned int*) addr);
7   }
8   // get_pte(va) 获取虚拟地址 va 对应的二级页表中的页表项内容
9   unsigned int get_pte(unsigned int va) {
10      unsigned int PGNUM = va >> ___________;
11      unsigned int addr = ___________ + PGNUM * 4;
12      return *((unsigned int*) addr);
13  }
```

16 网络和并发编程

要点

- 了解网络的物理层、链路层、网络层、传输层和应用层的基本结构，知道每一层所对应的硬件及其特性（如交换机、路由器）。
- 知道什么是网络协议，熟悉网络包在传输过程中包装和拆解的过程。了解常见的几种应用层网络协议（HTTP、DNS）。知道 TCP 和 UDP 两种协议及其是否有连接、是否可靠等性质。
- 熟悉 IP 地址的结构和用途。知道域名和 IP 地址之间的对应关系。
- 掌握客户端–服务器模型和套接字的概念（地址、端口、协议），熟悉 Linux 中读写套接字的 API 并能编写简单的网络服务器。
- 了解编写并发服务器的三种手段。
- 熟悉线程的概念，知道 POSIX 线程的常见 API。会判断线程程序中，变量是否共享以及共享变量在内存中的性质。
- 熟悉互斥同步问题的概念。会绘制进度图并分析互斥区域和不安全区域。
- 熟练掌握使用信号量解决互斥锁（mutex）、生产者–消费者问题和读者–写者问题的方法，能灵活运用到实际代码中。知道线程提高并行性和同步带来的问题。
- 理解线程安全和可重入的概念，会判断代码是否为线程安全。
- 熟练掌握死锁的概念，会判断程序是否有死锁及如何解决。

1. `volatile` 修饰符保证定义的变量存放在内存中，而不总在寄存器中。请阅读下面的代码，在两个进程的地址空间中标出变量 `gCount`、`vCount` 与 `lCount` 的位置。如果一个量出现多次，那么就标多次。

```c
long gCount = 0;
void *thread(void *vargp) {
    volatile long vCount = *(long *) vargp;
    static long lCount = 0;
    gCount++; vCount++; lCount++;
    printf("%ld\n", gCount + vCount + lCount);
    return NULL;
}
int main() {
    long var; pthread_t tid1, tid2;
    scanf("%ld", &var);
    fork();
    pthread_create(&tid1, NULL, thread, &var);
    pthread_create(&tid2, NULL, thread, &var);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

2. 下面的程序会引发竞争（不考虑原子性问题）。一个可能的输出结果为 `2 1 2 2`。解释输出这一结果的原因。

```c
long foo = 0, bar = 0;
void *thread(void *vargp) {
    foo++; bar++;
    printf("%ld %ld ", foo, bar); fflush(stdout);
    return NULL;
}
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

3. 信号量 `w`, `x`, `y`, `z` 均被初始化为 `1`。下面的两个线程运行时可能会发生死锁。给出发生死锁的执行顺序。

线程 1	`P(w), P(x), P(y), P(z), V(w), V(x), V(y), V(z)`
线程 2	`P(x), P(z), P(y), P(w), V(x), V(y), V(w), V(z)`

4. 某次考试有 30 名学生与 1 名监考老师，该教室的门很狭窄，每次只能通过一人。考试开始

前，老师和学生进入考场（有的学生来得比老师早），当人来齐以后，老师开始发放试卷。拿到试卷后，学生就可以开始答卷。学生可以随时交卷，交卷后就可以离开考场。当所有的学生都上交试卷以后，老师才能离开考场。请用信号量与 PV 操作，解决这个过程中的同步问题。以下所有空缺语句均为 PV 操作。

程序使用的全局变量为 `stu_count: int`，表示考场中的学生数量，初值为 0。信号量有：

- ▷ `mutex_stu_count`，保护全局变量，初值为 1。
- ▷ `mutex_door`，保证门每次通过一人，初值为 _____。
- ▷ `mutex_all_present`，保证学生都到了，初值为 _____。
- ▷ `mutex_all_handin`，保证学生都交了，初值为 _____。
- ▷ `mutex_test[30]`，表示学生拿到了试卷，初值均为 _____。

代码如下：

```
Teacher: // 老师
    _____
    从门进入考场
    _____
    _____ // 等待同学来齐
    for (i = 1; i <= 30; i++)
        _____ // 给 i 号学生发放试卷
    _____ // 等待同学将试卷交齐
    _____
    从门离开考场
    _____

Student(x): // x 号学生
    _____
    从门进入考场
    _____
    P(mutex_stu_count);
    stu_count++;
    if (stu_count == 30)
        _____
    V(mutex_stu_count);
    P(mutex_test[i]) // 等待拿自己的卷子
    学生答卷
    P(mutex_stu_count);
    stu_count--;
    if (stu_count == 0)
        _____
    V(mutex_stu_count);
    _____
    从门离开考场
    _____
```

5. 假设每个资源都有一个互斥锁，以下设定是否可能导致死锁？说明理由。

(1) 一个系统有四个相同类型的资源，它们被三个进程同时共享，每个进程最多需要两个资源一个系统有 m 个相同类型的资源，它们被 n 个进程同时共享。

(2) 在每个时刻，进程只能至多申请或者释放一个资源。假设系统满足以下两个条件：每个进程所需的最大资源数量不超过 m，所有进程的最大资源需求数量之和小于 $m+n$ 个。

17 网络与并发编程—往年考题

1. (2018) 下面有关计算机网络概念的叙述中，正确的是：

A. 大写字母的 Internet 用来描述互联网的一般概念，而小写字母的 internet 用来描述一种具体的实现，也就是全球 IP 互联网。

B. 在一个基于集线器的以太网中，如果往一台主机发送一段数据帧，那么其他主机无法看到这个帧。

C. IP 协议提供基本的命名方法和递送机制，因此我们能够借助 IP 协议，从一台主机往另一台主机发送包，即使两台主机不在同一个 LAN 内。

D. 当一段数据通过路由器，从 LAN1 被发送到 LAN2 时，附加的互联网络包头和局域网帧头保持不变。

2. (2018) 下面有关套接字接口的叙述中，错误的是：

A. 套接字接口常常被用来创建网络应用。

B. Windows 10 系统没有实现套接字接口。

C. `getaddrinfo()` 和 `getnameinfo()` 可以被用于编写独立于特定版本的 IP 协议的程序。

D. `socket()` 函数返回的描述符，可以使用标准 Unix I/O 函数进行读写。

3. (2018) 使用浏览器打开网页 `www.pku.edu.cn` 的过程中，下列网络协议中，可能会被用到的网络协议有 _____ 个：DNS、TCP、IP、HTTP。

4. (2018) 下面关于线程安全和可重入的叙述中，哪一个是正确的？

A. 如果一个函数的所有参数都是值传递的且无返回值，该函数一定是可重入的。

B. 函数的可重入版本一定比不可重入版本高效。

C. 可重入函数一定是线程安全的。

D. 以上说法都不正确。

5. (2018) 下列关于进程与线程的描述中，哪一个是不正确的？

A. 一个进程可以包含多个线程。

B. 进程中的各个线程共享进程的代码、数据、堆和栈。

C. 进程中的各个线程拥有自己的线程上下文。

D. 线程的上下文切换比进程的上下文切换快。

6. (2018) 给定下列代码片段：

```
1    char **ptr; /* global var */
```

```
    int main(int main, char *argv[]) {
        long i; pthread_t tid;
        char *msgs[2] = {" Hello from foo", " Hello from bar" };
        ptr = msgs;
        for (i = 0; i < 2; i++)
            Pthread_create(&tid, NULL, thread, (void *) i);
        Pthread_exit(NULL);
    }
    void *thread(void *vargp) {
        long myid = (long) vargp;
        static int cnt = 0;
        printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
        return NULL;
    }
```

下列哪一组变量集合是对等线程 1 引用的?

A. ptr, cnt, i.m, msgs.m, myid.p0, myid.p1
B. ptr, cnt, msgs.m, myid.p0
C. ptr, cnt, msgs.m, myid.p1
D. ptr, cnt, i.m, msgs.m, myid.p1

7. (2018) 给定如下程序:

```
    #include <stdio.h>
    #include <pthread.h>
    int i = 0;
    int j = 0;
    void *do_stuff1(void *arg __attribute__((unused))) {
        int a;
        for (a = 0; a < 1000; a++) { i++; j++; }
        return NULL;
    }
    void *do_stuff2(void *arg __attribute__((unused))) {
        int a;
        for (a = 0; a < 1000; a++) { j++; i++; }
        return NULL;
    }
    int main() {
        pthread_t tid1, tid2;
        pthread_create(&tid1, NULL, do_stuff1, NULL);
        pthread_create(&tid2, NULL, do_stuff2, NULL);
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
        printf("%d, %d\n", i, j);
        return 0;
    }
```

(1) 用以下元素的编号写出 `i++` 编译后最可能的汇编代码。(a) `mov` (b) `add` (c) `0x601040` (d) `%eax` (e) `$0x1`

(2) 请分别回答该程序是否有可能输出如下结果，并简述原因。(a) `2000, 2000` (b) `1500, 1500` (c) `1000, 1000` (d) `2, 2`

(3) 某同学在学习了信号量之后，决定要让程序能稳定输出 `2000, 2000`，于是对程序进行了如下改写。

```c
#include <stdio.h>
#include <pthread.h>
int i = 0;
int j = 0;
sem_t si;
sem_t sj;
void *do_stuff1(void *arg __attribute__((unused))) {
    int a;
    for (a = 0; a < 1000; a++) {
        P(&si);
        P(&sj);
        i++;
        j++;
        V(&si);
        V(&sj);
    }
    return NULL;
}
void *do_stuff2(void *arg __attribute__((unused))) {
    int a;
    for (a = 0; a < 1000; a++) {
        P(&sj);
        P(&si);
        j++;
        i++;
        V(&si);
        V(&sj);
    }
    return NULL;
}
int main() {
    pthread_t tid1, tid2;
    sem_init(&si, 0, 1);
    sem_init(&sj, 0, 1);
    pthread_create(&tid1, NULL, do_stuff1, NULL);
    pthread_create(&tid2, NULL, do_stuff2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("%d\n", i);
```

```
40          return 0;
41      }
```

请问该同学的程序有什么潜在问题？为什么？如果对该同学的程序改动一处数字来消除上述问题，同时仍然保证输出结果稳定为 2000，2000，应该如何改动？回答：将 _____（填行号）行的 _____（填数字）改为 _____（填数字）。

8. (2016) 关于 IP，以下说法正确的是：

A. IP 协议提供了可信赖的主机与主机之间的数据包传输能力。
B. IP 地址的长度固定为 32 位。
C. 一个域名可以映射到多个 IP，但一个 IP 不能被多个域名映射。
D. 一个域名可以不映射到任何 IP。

9. (2016) 以下说法正确的是：

A. HTTP 协议规定服务器端使用 80 端口提供服务。
B. 使用 TCP 来实现数据传输一定是可靠的。
C. Internet 上的两台的主机要通信必须先建立端到端连接。
D. 在 Linux 中只能通过 Socket 接口进行网络编程。

10. (2016) 假设有一个 HTTPS（基于 HTTP 的一种安全的应用层协议）客户端程序想要通过一个 URL 连接一个电子商务网络服务器获取一个文件，并且这个服务器 URL 的 IP 地址是已知的，以下哪种协议是一定不需要的？

A. HTTP
B. TCP
C. DNS
D. SSL/TLS

11. (2016) 请阅读如下代码：

```
1   sem_t s;
2   int main() {
3       int i;
4       pthread_t tids[3];
5       sem_init(&s, 0, 1);
6       for (i = 0; i < 3; i++)
7           pthread_create(&tids[i], NULL, justdoit, NULL);
8       for (i = 0; i < 3; i++)
9           pthread_join(&tids[i], NULL);
10      return 0;
11  }
12  int j = 0;
13  void *justdoit(void *arg) {
14      P(&s);
15      j = j + 1;
```

```
16          V(&s);
17          printf("%d\n", j);
18      }
```

下列哪一个输出结果是不可能的?

A. (1, 3, 2)
B. (2, 3, 2)
C. (3, 3, 2)
D. (2, 1, 2)

12. (2016) 请阅读下列代码 `badcnt.c`:

```
1   /* Global shared variable */
2   volatile long cnt = 0;
3   /* Counter */
4   int main(int argc, char **argv) {
5       long niters;
6       pthread_t tid1, tid2;
7       niters = atoi(argv[1]);
8       Pthread_create(&tid1, NULL, thread, &niters);
9       Pthread_create(&tid2, NULL, thread, &niters);
10      Pthread_join(tid1, NULL);
11      Pthread_join(tid2, NULL);
12      /* Check result */
13      if (cnt != (2 * niters))
14          printf("BOOM! cnt=%ld\n", cnt);
15      else
16          printf("OK cnt=%ld\n", cnt);
17      exit(0);
18  }
19  /* Thread routine */
20  void *thread(void *vargp) {
21      long i, niters = *((long *) vargp);
22      for (i = 0; i < niters; i++)
23      cnt++;
24      return NULL;
25  }
```

运行后可能产生如下结果:

```
$ ./badcnt 10000
OK cnt=20000
```

或者:

```
$ ./badcnt 10000
BOOM! cnt=13051
```

为什么会产生不同的结果？请分析产生不同结果的原因。

13. (2016) 某辆公交车的司机和售票员为保证乘客的安全，需要密切配合、协调工作。司机和售票员的工作流程如下所示。请编写程序，用 PV 操作来实现司机与售票员之间的同步。

司机进程循环执行以下命令	售票员进程循环指令以下命令
① 启动车辆 ② 正常行驶 ③ 到站停车 ④	⑤ 关门 ⑥ 报站名或维持秩序 ⑦ 到站开门 ⑧

(1) 请设计若干信号量，给出每一个信号量的作用和初值。

(2) 请将信号量对应的 PV 操作填写在代码中适当位置。注：每一标号处可以不填入语句（请标记为 X）或多条语句。

14. (2015) HTTP 协议中，哪个命令可以用来获取动态内容？

A. `HEAD`

B. `GET`

C. `POST`

D. `PUT`

15. (2015) 下列关于计算机网络概念的说法中，哪一项是正确的？

A. HUB 会把它任意端口上接收到的帧只转发到它的目的地去。

B. 当在不同的 LAN 中的主机 A 和主机 B 通信的过程中，他们的数据包中的 LAN frame header 不会变化。

C. `162.105.0.0` 是一个 B 类地址。

D. 同一台主机每次进入相同的网络，通过动态地址分配的到的 IP 地址总是相同的。

16. (2015) 有如下代码：

```
1   int counter = 0;
2   void *thread(void *vargp) {
3       int thread_var = ((int *) vargp);
4       static int thread_counter = 0;
5       thread_internal(thread_var);
6       thread_counter++;
7       return NULL;
8   }
9
10  int main(int argc, const char **argv) {
11      int tid1, tid2;
```

```
12          int var = atoi(argv[1]);
13          Pthread_create(&tid1, NULL, thread, (void *) var);
14          Pthread_create(&tid2, NULL, thread, (void *) var);
15          Pthread_join(tid1, NULL);
16          Pthread_join(tid2, NULL);
17          return 0;
18      }
```

那么线程 tid1 与线程 tid2 可以共享的变量是

A. counter, var
B. counter, thread_counter
C. var, thread_counter
D. thread_var, thread_counter

17. (2015) 有四个信号量，初值分别为：a = 1, b = 1, c = 1, d = 1。

进程 1	进程 2	进程 3
P(a)	P(d)	P(d)
P(d)	P(a)	P(c)
P(c)	P(c)	P(b)
P(b)	P(b)	P(a)
V(c)	V(d)	V(c)
V(b)	P(d)	V(b)
V(d)	V(a)	V(a)
V(a)	V(b)	V(d)
	V(c)	
	V(d)	

下列哪两个线程并发执行时，一定不会发生死锁？

A. 1, 2
B. 1, 3
C. 2, 3
D. 都可能发生死锁

18. (2015) 以下关于网络的说法哪些是正确的？

A. 在 client-server 模型中，server 通常使用监听套接字 listenfd 和多个 client 同时通信。
B. 在 client-server 模型中，套接字是一种文件标识符。
C. 准确地说，IP 地址是用于标识主机的 adapter（network interface card），并非用于标识主机。
D. Web 是一种互联网协议。
E. 域名和 IP 地址是一一对应的。
F. Internet 是一种 internet。

19. (2015) 有三个线程 PA、PB、PC 协作工作以解决文件打印问题。PA 将记录从磁盘读入内存缓冲区 Buff1，每执行一次读一个记录；PB 将缓冲区 Buff1 的内容复制到缓冲区 Buff2，每执行一次复制一个记录；PC 将缓冲区 Buff2 的内容打印出来，每执行一次打印一个记录。缓冲区 Buff1 可以放 4 个记录；缓冲区 Buff2 可以放 8 个记录。请用信号量及 PV 操作实现上述三个线程以保证文件的正确打印。

PA 循环执行以下命令	PB 循环执行以下命令	PC 循环执行以下命令
① 从磁盘读入一个记录 ② 将记录放入 Buff1 ③	④ 从 Buff1 中取出一个记录 ⑤ 将记录放入 Buff2 ⑥	⑦ 从 Buff2 中取出一个记录 ⑧ 打印

(1) 请设计若干信号量，给出每一个信号量的作用和初值。

(2) 请将信号量对应的 PV 操作填写在代码中适当位置。注：每一标号处可以不填入语句（请标记为 X）或多条语句。

20. (2014) 如果两个局域网高层分别采用 TCP/IP 协议和 SPX/IPX 协议，那么可以选择的互连设备应是

A. 网桥
B. 集线器
C. 路由器
D. 交换机

21. (2014) 下面说法正确的是：

A. TCP 是一种可靠的无连接协议。
B. UDP 是一种不可靠的无连接协议。
C. Web 浏览器与 Web 服务器通信采用的协议是 HTML。
D. 数字数据只能通过数字信号传输。

22. (2014) 对于如下 C 语言程序：

```
1   #include "csapp.h"
2   void *thread (void *arg) {
3       printf("Hello World");
4       Pthread_detach(pthread_self());
5   }
6   int main(void) {
7       pthread_t tid;
8       int sta;
9       sta = Pthread_create(&tid, NULL, thread, NULL);
10      if (sta == 0)
11          printf("Oops, I can not create thread\n");
12      exit(NULL);
```

```
13      }
```

在上述程序中，Pthread_detach 函数的作用是

A. 使主线程阻塞以等待线程 thread 结束。
B. 线程 thread 运行结束后会自动释放所有资源。
C. 线程 thread 运行后主动释放 CPU 给其他线程。
D. 线程 thread 运行后成为僵尸线程。

23. (2014) 两个线程中共享如下一段 C 代码：

```
1     for(j = 0; j < N; j++)
2         count += 2;
```

假设其对应的汇编代码如下：

```
        movq    (%rdi), %rcx            }
        testq   %rcx, %rcx              } Hi
        jle     .L2                     }
        movl    $0, %eax                }
.L3:
        movq    count(%rip), %rdx       Li
        addq    $2, %rdx                Ui
        movq    %rdx, count(%rip)       Si
        addq    $1, %rax                }
        cmpq    %rcx, %rax              } Ti
        jne     .L3                     }
.L2:
```

上面用 $H_i, L_i, U_i, S_i, T_i\ (i = 1, 2)$ 划分了指令的几个部分，下标代表执行的线程。在下列指令顺序对应的轨迹线中，哪一个是安全轨迹线？

A. $H_1 \to H_2 \to L_2 \to L_1 \to U_2 \to U_1 \to S_1 \to S_2 \to T_1 \to T_2$
B. $H_1 \to L_1 \to U_1 \to H_2 \to L_2 \to S_1 \to T_1 \to U_2 \to S_2 \to T_2$
C. $H_2 \to L_2 \to U_2 \to H_1 \to S_2 \to L_1 \to T_2 \to U_1 \to S_1 \to T_1$
D. $H_2 \to L_2 \to H_1 \to L_1 \to U_1 \to U_2 \to S_2 \to T_2 \to S_1 \to T_1$

24. (2014) 以下问题默认为 IPv4 协议。

(1) 一个服务器拥有四个独立的固定 IP 地址，那么它在 web 应用端口 80，理论上可以最多再监听多少个来自一个客户端独立的 socket 连接（设每个客户端只有一个固定 IP 地址）?

(2) 设客户端 IP 为 162.105.192.178，内网 IP 为 192.168.100.121。HTTP 服务器端 IP 为 208.216.181.15。服务器使用的是默认监听端口号。指出下面这个浏览器应用的 connection socket pair 有何错误，并简要说明原因：客户端 192.168.100.121:15321，服务器 208.216.181.15:25。

25. (2014) 桌子上有一个水果盘，恰能容纳一个水果。一家四口人中，爸爸专门往盘子里放苹果，妈妈专门往盘子里放橘子；儿子专等盘子里的苹果吃，女儿专等盘子里的橘子吃。请用信号量及 PV 操作实现上述四个过程以保证家庭和睦。

爸爸循环执行	妈妈循环执行	儿子循环执行	女儿循环执行
准备一个苹果 ① 往盘子里放一个苹果 ②	准备一个橘子 ③ 往盘子里放一个橘子 ④	⑤ 从果盘中拿出苹果 ⑥ 吃苹果	⑦ 从果盘中拿出橘子 ⑧ 吃橘子

(1) 请设计若干信号量，给出每一个信号量的作用和初值。

(2) 请将信号量对应的 PV 操作填写在代码中适当位置。注：每一标号处可以不填入语句（请标记为 X）或多条语句。

26. (2013) 下列关于计算机网络概念的说法中，哪一项是正确的？

A. 全球最大的计算机网络是互联网 Internet，所以计算机网络协议是 Internet Protocol 即 IP 协议。

B. 计算机之间的网络通信是一个机器上的一个 process（如 client process）与另一个机器上的 process（如 server process）之间的通信。

C. 网络应用程序有默认的端口号，大部分应用的端口号可以修改，而少部分知名应用如 Web 服务程序的端口号 80 是无法修改的。

D. 一个域名只能对应一个 IP 地址，而一个 IP 地址可以对应多个域名。

27. (2013) 在 client-server 模型中，一个连接可以由 IP 地址和端口号的组合来表示。假设一个访问网页服务器的应用，客户端 IP 地址为 `128.2.194.24`，目标服务器端 IP 地址为 `208.216.181.15`，用户设置的代理服务器 IP 地址为 `155.232.108.39`。目标服务器同时提供网页服务（默认端口 80）和邮件服务（默认端口 25）。当客户端向目标服务器发送访问网页的请求时，下面 connection socket pairs 正确的一组是

A. 客户端请求 `(128.2.194.242:25, 155.232.108.39:80)`
代理请求 `(128.2.194.242:51213, 208.216.181.15:80)`

B. 客户端请求 `(128.2.194.242:51213, 155.232.108.39:80)`
代理请求 `(128.2.194.242:12306, 208.216.181.15:80)`

C. 客户端请求 `(128.2.194.242:25, 208.216.181.15:80)`
代理请求 `(155.232.108.39:51213, 208.216.181.15:80)`

D. 客户端请求 `(128.2.194.242:51213, 208.216.181.15:80)`
代理请求 `(155.232.108.39:12306, 208.216.181.15:80)`

28. (2013) 下列代码输出正确的是：

```
1  void *th_f(void *arg) {
2      printf("Hello World");
3      pthread_exit(0);
4  }
```

```
5   int main(void) {
6       pthread_t tid;
7       int st;
8       st = pthread_create(&tid, NULL, th_f, NULL);
9       if (st < 0) {
10          printf("Oops, I can not create thread\n");
11          exit(-1);
12      }
13      sleep(1);
14      exit(0);
15  }
```

A. Oops, I can not create thread

B. Hello World
   Oops, I can not create thread

C. Hello World

D. 不输出任何信息

29. (2013) 以下问题默认为 IPv4 协议。

(1) 一个服务器拥有两个独立的固定 IP 地址，那么它在 web 应用端口 80 上最多可以监听多少个独立的 socket 连接？

(2) 该服务器在所有有 web 应用端口上最多可以监听多少个独立的 socket 连接？

30. (2013) 某个城市为了解决市中心交通拥堵的问题，决定出台一项交通管制措施，对进入市中心区的机动车辆实行单双日限制行驶。具体要求是，逢单日，只允许车辆牌号号码为单数的机动车进入市中心区；逢双日，只允许车辆牌号号码为双数的机动车进入市内中心区。

有一个进入市中心区的交通路口，进入该路口的道路有一条，离开该路口的道路有两条，其中一条是通往市中心区的道路，而另一条是绕过市中心区的环路，在进入路口处设置了自动识别车辆牌号的识别设备与放行栅栏控制设备。

在单日，遇到单号车辆进入路口车辆号码识别区，号码识别设备打开通往市中心区道路的放行栅栏；而遇到双号车辆，则打开绕过市中心区环路的放行栅栏。反之亦然。显然，只有在该路口车辆号码识别区中无车时，才允许一辆车进入车辆号码识别区。同时为了防止有车辆混过路口，两个放行栅栏平时处于关闭状态，只有在车辆号码识别区中的车辆已被识别出单双号之后，放行栅栏才会在识别设备的控制下，打开对应的放行栅栏，在车辆通过之后，该放行栅栏自行关闭。

请编写程序，用 PV 操作来实现车牌号检查和两个放行栅栏之间的同步。

检查车辆牌号线程循环执行	市区放行栅栏线程循环执行	环路放行栅栏线程循环执行
车辆到达识别路口 ① 车辆进入号码识别区 如果号码为奇数 ② 否则 ③	④ 允许车辆进入市中心区 ⑤	⑥ 允许车辆绕行环路 ⑦

(1) 请设计若干信号量，给出每一个信号量的作用和初值。

(2) 请将信号量对应的 PV 操作填写在代码中适当位置。注：每一标号处可以不填入语句（请标记为 X）或多条语句。

18 部分参考答案

参考答案一

1. 根据小端法可知分别为 `0x56 0x34 0x12 0x00`、`0x34 0x12 0x78 0x56`。

2. 根据小端法，`A` 在内存中从高地址到低地址分别是 `11 11 22 22`，得到 `P = 0x2222`，同理 `Q = 0x3333`，结果为 `0x5555`。

3. 答案为 `0x0303`，分析方法和前一题一样。

4. 1 取 `x = 1, y = -1` 即不正确；2 取 `x = -1` 即不正确；3 正确，利用异或的交换律、结合律，以及 `x ^ x == 0`（视为模 2 加法即可）；4 正确，即使是对负数；5 不正确，负奇数该运算向 0 舍入；6 正确，`(~x)  y` 也就是 `(~x) + (~y) + 1`，注意运算优先级；7 不正确，`!!ux` 是有符号数。

5. `sizeof` 返回的结果是无符号数，因此循环条件恒不成立，不会执行循环体。

6. 这是基本练习，填完的表格如下：

描述	二进制表示	M（写成分数）	E	f
负零	10000000	/	/	-0.0
/	01000101	21/16	1	1/8
最小的非规格化负数	10001111	15/16	-2	-15/64
最大的规格化正数	01101111	31/16	3	31/2
一	00110000	1	0	1.0
/	01010110	11/8	2	5.5
+∞	01110000	/	/	/

7. 根据无穷大的定义，类 IEEE 754 的浮点数都只有 1 个无穷大。一般阶码越多，则最大绝对值越大，所以 B 大；A 大，理由同前；小数长度越多，则 `NaN` 越多，而总的可能表示的模式数一样，所以 B 更多。

8. 1 正确；2 取 `a = INF, b = FLT_MAX`；3 取 `a = 16777220, b = 16777218`；4 取最大的非规格化负数；5 正确；6 `nan != nan`；7 正确。

9. $16777217 = 2^{24} + 1$，这时写成科学计数法小数点后面要 24 位，超出了单精度浮点的表示范围，会发生舍入。

10. 写出 $2^{25} + 34$ 的科学计数法表示后按要求舍入即可，这些数舍入到单精度浮点数的结果计算如下。我们用下划线表示单精度浮点数小数表示的极限位置，后面的需要进行舍入，根据四舍六入五成双规则即可得到答案（注意最后一位相当于 1，倒数第二位相当于 2）。

1. `1.00000000000000000001000_10 * 2^25`，差 2。
2. `1.00000000000000000001000_11 * 2^25`，差 −1。
3. `1.00000000000000000001001_00 * 2^25`，差 0。
4. `1.00000000000000000001001_01 * 2^25`，差 1。
5. `1.00000000000000000001001_10 * 2^25`，差 −2
6. `1.00000000000000000001011_11 * 2^25`，差 −1。
7. `1.00000000000000000001010_00 * 2^25`，差 0。
8. `1.00000000000000000001010_01 * 2^25`，差 1。

参考答案二

1. B。

5. C，连接词的完备集。

10. A。计算过程：浮点数阶码 8 位，即最大指数为 127，尾数为 23 位，即最低位表示的数为 $2^{-23}2^{127} = 2^{104}$。那么误差在刚好落在最低位的一半的时候最大，即 2^{103}。

11. D。计算 $0.5 \le \left|\frac{5}{2^e} - 1\right| < 1 \implies e = 4, \text{EXP} = 7 = 111_2$，注意此时 M 为负数，$|M| = 0.01010_2$，用补码表示得 1.10101_2。

13. B。此题和 C90 标准密切相关，分析参看 https://stackoverflow.com/questions/9941261/warning-this-decimal-constant-is-unsigned-only-in-iso-c90。

15. D，注意 C 可能是有符号整数。

18. B，可以直接计算，注意不同运算的舍入规则。

23. 分别为 D、D、BD、E、AD、E、C、C。

参考答案三

1. 1 不正确；2 不正确，`$` 只用来表示立即数；3 正确，是内存地址 `0x30`；4 正确；5 不正确，缩放比例只能是 1、2、4、8；6 正确；7 不正确，x86-64 不允许将除了 64 位寄存器以外的寄存器作为寻址模式基地址；8 不正确，`%rsp`不能作为操作数！

2. 注意 `movl` 会清零高 32 位。如果 `movabsq` 的立即数是负数，主要要取绝对值（本题中不需要）。答案为：

1. `0x0123456789ABCDEF, 0x0000000000000000`
2. `0x0123456789ABCDEF, 0x000000000000CDEF`
3. `0x0123456789ABCDEF, 0xFFFFFFFFFFFFCDEF`
4. `0x00000000FFFFCDEF, 0xFFFFFFFFFFFFCDEF`
5. `0x0123456789ABCDEF, 0xFFFFFFFFFFFFCDEF`
6. `0xFFFFFFFF89ABCDEF, 0xFFFFFFFFFFFFCDEF`

3. C。A、B 中，`movzbl/movzwl` 都生成了四字节，把高位设为 0。D 中 `cltq` 是对 `%eax`的符号拓展。而 C 中 `movl` 和 `movzlq` 等价。

4. 1 不正确，目的不能是立即数；2 正确；3 不正确，两个操作数不能同时是内存地址；4 正确；5 不正确，要用 `movabsq`；6 正确；7 不正确，`movabsq` 的目标地址必须是整数寄存器；8 不正确，不能用 `mov` 向 `%rip` 中传入数据，否则读程序计数器产生破坏。

5. C，注意前两个选项只是计算了地址，而 `int` 每个元素 4 字节，D 错误。

6. 如下所示:

```
long func(long a, long b) {
    return a * 15 + b * 7;
}
```

7. A。如果 `a > b`，那么 `a - b` 或者为正，或者发生上溢变负；同时 `a` 不等于 `b`。

8. 如下所示（快速幂）:

```
long func(long a, long b) {
    long ans = 1;
    while (b > 0) {
        if (b & 1)
            ans = ans * a;
        b = b >> 1;
        a = a * a;
    }
    return ans;
}
```

9. 1、4 会被编译成条件传送。1 由于比较前计算出的 a 与 b 就是传送的目标，因此会被编译成条件传送；2 由于比较结果会导致 a 与 b 指向的元素发生不同的改变，因此会被编译成条件跳转；3 由于指针 a 可能无效，因此会被编译为条件跳转；4 会被编译成条件传送，因为 a 和 b 都是局部变量，返回的时候对 a 和 b 的操作都是无影响的。

10. 03; f8。第五行跳转位置为 0xf8(-8) + 0x4004dd = 0x4004d5，注意进行跳转之前，PC 指向该指令的下一条指令。

11. 做法是首先根据跳转表, 确定各标号的起始地址, 然后再作汇编代码的对应. 答案如下所示：

```
    case 0:
    case 1:
        c = c - 5;
    case 2:
        res = 4 * c; // or res *= c
        break;
    case 5:
        res = 86547; // or 0x15213
        break;
    case 3:
        c = 2;
    case 7:
        b = b & c;
    default:
        res += b; // or res = b + 4
```

13. 阅读 callee 汇编代码可知，首先 *b 进入 %rax，随后 %rax 变为 *a ^ *b。根据 C 代码确定下一步要将结果转入 *a 中，所以汇编语言代码的空应填写 %rax, %(rdi)。caller 汇编代码的阅读不需要太仔细 (就做题而言)，极易填写。注意这里体现的典型的 for 循环结构，快速识别 %ebx 为循环变量，.L4 后为循环条件检查。后者需要计算 n / 2，要留意编译器选择用移位作除法，但是对于负数需要作修正 (舍入方式不同)。sarq 等移位操作若只有一个操作数，则移位量默认为 1。

此处栈空间的填写难度不大。首先，main 即将发生调用时，%rsp 值为 0xf...f80，接下来 call 后需要在栈中设置返回地址，这个返回地址为 0x4000ac + 0x5 = 0x4000b1。所以栈的第三个空填此。注意栈是向下增长的，所以前两个空不确定。接下来 caller 要保存 %r12, %rbp, %rbx 三个寄存器，因此先后填入 0x213, 0x18, 0x15。最后，callee 被调用，需要再次填入返回地址 0x400088 + 0x5 = 0x40008d。最后一个空显然不确定。

14. 解题时不妨用框图的形式把整个结构的内存布局画出来并标好偏移量。CC1 从位置 0 开始；整数 4 字节对齐，所以 II1从位置 8 开始；长整数 8 字节对齐，所以 LL1 必须从位置 16 开始；而后的字符数组 CC2 延展到位置 34，仍由长整数 8 字节对齐，LL2 只好从位置 40 开始；最后的整数从位置 48 开始，现在总大小为 52 字节。在结构体后面再放一个相同的 A，发现需要再补 4 字节才能使得第二个 A 的 LL1 对齐，所以第一问为 56 字节。第二问：可以重排顺序为 LL1 LL2 II1 II2 CC1 CC2，刚好没有空白空间，得到的大小为 40 字节。显然这个可以用

贪心法来做（先放对齐要求高的元素）。

15. 本题的主要难点是确定 union，struct 本身的对齐，方法一般是在结构或联合后面再放一个相同的对象，构成数组，看需要补多少字节才能使得第二个元素对齐（对齐要求的是每个基本数据类型对齐！）。经验规则是按其中对齐要求最严格的元素进行这一工作。

根据对齐规则，题给 union 只需要 7 字节就能使得内部对齐，但是后面接一个同样的对象时 short 需要 2 字节对齐，所以它实际上为 8 字节，且要求 2 字节对齐。这样我们就知道 struct 的大小为 $3+\underline{1}+8+4=16$ 字节。由此，前 4 空答案为 4、8、12、16。同理可以得到后面几问的答案，为 14、8、8、16、(3、7、12、16)。

16. 这是典型的汇编综合题，需要同时结合 C 代码和汇编代码来观察。

第一步先通读 C 代码，确定代码大意，便于在汇编中找一些对应的片段。然后来填写汇编代码。主要难点在厘清 C 语言变量，内存和寄存器的映射关系，确定代码的用途。先看主函数，前三行为金丝雀。接下来，0x69，0xfc 分别进入 %rsp，%rsp + 8 对应的内存位置中。根据主函数唯一的 &np 取局部变量地址，以及紧接着 %rsp 被作为参数传入 func，知道这就是 np 结构体的初始化工作。所以 C 代码的第 10、11 空分别填写 105、252。

接着，func 的返回值被存入 %rsi，一个立即数（看上去像地址）被存入 %rdi，然后调用 printf 函数。结合 C 代码知道 %rdi 是字符串 "%ld"（属于 char* 类型）的地址。所以由给出的 ASCII 码表计算得到问答题 (1) 应填写 0x25，0x6c，0x64，0x00。特别留意字符串的最后有一个 0x00 表示结尾！主函数的最后几个空是检查金丝雀是否被破坏，易知金丝雀的原值在 %fs:0x28，而在栈上对应于地址 %rsp + 0x18（这个之后还要用到）。检查金丝雀是否被破坏是容易的，40063b 先把后者放到 %rcx 中，所以要比较 %fs:0x28，%rdx，这就是空 (4) 的答案。接下来要条件跳转，因为如果条件成立会正常返回，所以空 (5) 写 je。空 (6) 是函数返回释放栈空间，因为一开始分配了 0x28 个字节，所以这个空也要填 0x28。

现在分析 func 函数。前三行是金丝雀以及寄存器清零。从地址 4005aa 开始分析，首先 p->a，p->b 分别进入 %rax，%rdx 中。接下来比较 p->a >= p->b。结合 C 代码，如果这成立，就不需要经过交换 p->a，p->b 的过程。立刻看出，4005b6，4005b9 两行完成了交换工作（没有出现 temp 的对应寄存器映射）。由于不确定交换之后 %rax 到底对应 p->a，p->b 中的哪个，所以其要重新移入 p->b 到 p->a 中。所以 4005bd 就是空 (1) 的跳转目标。接下来用 test 指令判断 %rax 是否为 0。注意 C 代码，如果某个条件成立，就要 return p->a。因为 4005cb 把 p->a 放到 %rax 中，所以空 (2) 会跳到函数结尾返回，也就是说跳转目标是 4005e2（返回之前要检查金丝雀是否被破坏）。由此也可知只有 p->b 为零时才返回，空 (7) 填写 p->b == 0。

紧接着处理 func 的递归调用。结合 C 代码知道要生成新的 np，而且 np.a，np.b 分别在栈地址的 %rsp，%rsp + 8 处。因为这时 p->a 在 %rdx 中，p->b 在 %rax 中，所以由 4005ce，4005d1 两行立刻知道 (8)、(9) 两空分别写 p->a - p->b，p->b。

补全 C 代码后，可以看出这是辗转相除法，输入是 252、105，它们的最大公约数是 21，故问答题的 (4) 回答 21。

最后填写栈。对于递归调用，我们应该先正确划分各次调用的栈的边界。第一步，由于主函数和 func 一开始都 subq 0x28，%rsp，所以栈空间都有 40 字节或 5 个 64 位地址。注意 6 号位置是 0x400629，恰好是主函数调用 func 的返回地址，所以我们知道 1–5 号位置是主

函数的栈帧，6 是从 func 返回主函数的返回地址；7–11 是第一次调用 func 的栈帧，12 是从第二次递归调用返回第一次调用 func 的返回地址，13–17 是第二次调用 func 的栈帧，以此类推。所以 12、18 都填 0x4005e2（func 中递归调用的下一条指令的位置）。

第二步，根据前面的分析，每个栈的栈帧的底部向上数第一个和第二个数都是存储局部变量 np（无论是主函数还是 func），所以位置 4、5 似乎分别应该写上 0xfc, 0x69。但是，注意后面的递归调用中，这两个位置的数会被交换以保证 p->a >= p->b！所以这两个空要倒过来，4、5 分别写上 0x69, 0xfc。第三步，由前面的分析，金丝雀值一直存在 %rsp + 0x18，也就是栈帧自底向上第四个位置，又题目条件说 %fs 寄存器没有改变，所以所有的金丝雀值都是 0xc76d5add7bbeaa00，这样 8、14、20 都是这个值。

对于栈的 10、11 号位置，前面的分析知道它们应该是 252 - 105, 105 = 0x93, 0x69（顺序未定），而交换后保证上一个要小于下一个，所以 10、11 给出的情况印证我们的分析。同理，16、17 空是 0x93 - 0x69 = 0x2a, 0x69，上一个要小于下一个。虽然 22、23 号位置不用填写，但是我们务必注意，栈在这个状态时，刚刚进行下一次递归调用，这两个数还没有被下一次递归调用修改顺序（尽管实际上不需要修改顺序）。

栈的剩下位置，7、21 都是不确定的，因为只是分配了没有修改过。（注意栈帧开头应该不是旧的帧指针，因为如果要这样需要显式 push。）

参考答案四

3. 此题有错，A、B 都不准确，有一些 SIMD 指令并不会改变条件码；C 明显是错的；D 基本正确，确实有指令可以直接写条件码。

4. 此题有错，B、D 都不对，其中 B 不能用 32 位寄存器寻址，D 中少了一个逗号。

9. B，这个条件相当于 (a == 0 || a == 1 || a == -1)。

14. C，从 subl 可看出待比较的数是字符型。

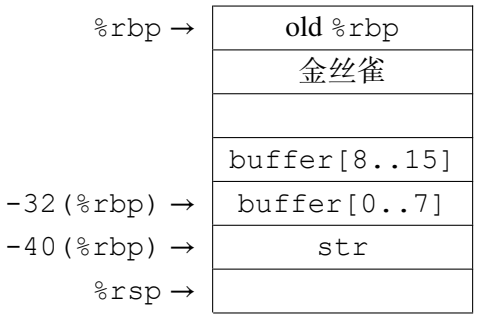
17. D。题目没有说明大小端，如果是大端法则为 A。若为小端法，注意 char 在不同的编译器上可能是有符号型或者是无符号型，所以计算知 –16 和 240 都有可能。

19. 此题有错，A、D 都可能产生（和优化选项有关）。

26. 此题的难点不多。其一是标号 .LC2 处的字符串中，形如 \252 的代码表示的是 8 进制的一个字节（注意表示字符串时总是可以用一个一个字节表示），main 函数中的字符串则是 16 进制的。

下面分析 myprint 汇编代码。这里前三步指令是标准的，分配了 48 字节的栈空间（所以填 subq $48, %rsp），并使得 %rbp 指向其开头，%rsp 指向其结尾。结合 C 语言的声明知 %rdi 是字符串 str 的开头，现在被存到 -40(%rbp) 的位置。接下来的三句汇编也是标准的，是在栈的头部放置金丝雀，因此填 movq %fs:40, %rax。

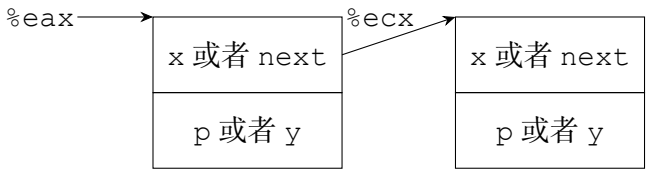
接下来的空需要倒推。注意到 strcpy 返回后，需要将 buffer 作为 printf 的第二个参数，由此推断出 buffer 在栈的 -32(%rbp) 的位置。栈空间的布局如下所示：



从而，xor 后的第一条指令对应于 str 参数，即 `movq -40(%rbp), %rdx`，第三条指令为 `movq %rdx, %rsi`，第四条指令为 `movq -32(%rbp), %rdi`。call 后仍为 leaq。myprint 函数的结尾是检查金丝雀，是标准的。填写 `movq %fs:40, %rax`、`xorq -8(%rbp), %rax` 或者 `movq -8(%rbp), %rax`、`xorq %fs:40, %rax` 都是可以的。

27. 本题我们主要分析的是 (2)、(4) 两问。

对于第 (2) 问，我们首先注意这是 IA32 汇编，第一个参数位置是 `8(%ebp)`，所以第一行结束后 `%eax` 是 up 的地址。我们画出联合体的图示（以及链表第二个元素）如下：

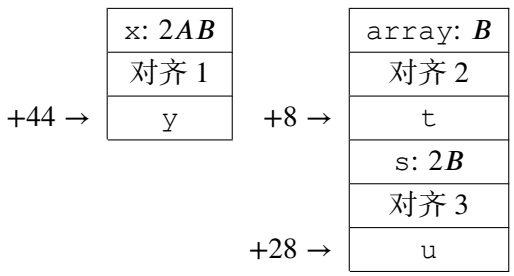


在第二行，我们对 `%eax` 进行了一次解引用，得到了 `up->e1.x` 或者 `up->e2.next`，结果到 `%ecx` 中。注意到第三行再次对结果进行了一次解引用，所以 `%ecx = up->e2.next`。随后解引用 `4(%ecx)`，可能得到 `up->e2.next->e1.p` 或者 `up->e2.next->e2.y`，结果在 `%edx` 中。注意到下一行再次对其进行了一次解引用，所以 `%edx = up->e2.next->e1.p`，进一步下一行中 `%edx = *(up->e2.next->e1.p)`。

第五和第六行类似分析，因为要从 `%edx` 中减去 `4(%eax)`，所以它必须是整数，因此是 `up->e2.y`，减法得到的 `*(up->e2.next->e1.p) - up->e2.y` 是个整数，存入 `%ecx` 指向的位置，所以是 `up->next->e1.x`，故最终答案是

`up->next->e1.x = *(up->e2.next->e1.p) - up->e2.y`。

第 (4) 问也要画图分析，首先给出两个结构体的粗略布局（考虑对齐，注意 str2 中的变量 s 是自然对齐的），并结合汇编代码决定变量的偏移量。注意 `8(%ebp)` 是第一个参数，`12(%ebp)` 是第二个参数。



根据对齐要求我们可列出方程

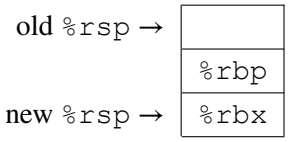
$$\begin{cases} 41 \le 2AB \le 44, \\ 5 \le B \le 8, \\ 25 \le 8 + 4 + 2B \le 28, \\ A, B \in \mathbb{N}, \end{cases}$$

解得 $A = 3, B = 7$。

28. 这是典型的汇编综合题，需要同时结合 C 代码和汇编代码来观察。

第一步先通读 C 代码，确定代码大意，便于在汇编中找一些对应的片段。然后来填写汇编代码。主要难点在厘清 C 语言变量，内存和寄存器的映射关系，确定代码的用途。

本题的 C 代码基本没什么信息，先看汇编。我们可以发现汇编代码首先访问了栈指针以下的内容，再修改栈指针的，栈空间为 16 字节。此时我们暂时不知道 `f+5` 位置应该做什么。接下来，我们知道 `%edi` 中存储的是 `n`，`%esi` 中存储的是 `m`。所以 `f+16` 对应于 C 代码的 `if (m > 0)`。之后两行使用了 `%ebp`，`%ebx` 两个寄存器，注意这是被调用者保存寄存器，所以我们现在意识到 `f+5` 的位置是保存 `%rbp`。这样，保存寄存器和弹出寄存器的 `f+51`，`f+55` 就能对应起来。不过我们需要注意的是，这两部分代码所对应的 `%rsp` 值是不一样的！如下图的栈空间分析所示：



所以我们知道第一个空填 `mov %rbp, -0x8(%rsp)`，第二个空填 `mov (%rsp), %rbx`。

下面的分析主要是为了补全 C 语言代码。紧接着确定 `m > 0` 的测试之后，`n`，`m` 分别转入了 `%ebp`，`%ebx` 中。若 `m > 0` 不满足，会直接跳到 `f+51` 并返回，注意 `%eax` 一直是 0，所以这确实对应于 C 代码的最后一句。接下来的 `f+24` 和 `f+27` 又比较是否 `n > 1`（所以 C 代码的一个空填此）。如果不满足，则跳转到 `f+65`。这里 `sete` 指令若 `n == 1` 则会将 `%eax` 的低字节 `%al` 设为 1，后面则进行零扩展，再跳到 `f+51` 返回。这说明 `else if` 句后面是 `n == 1` 的判断。

最后确定中间的步骤。`f+29` 处计算了 `n - 1` 到 `%rdi` 中并递归调用，说明返回了 `f(n - 1, m)`。这个值接着在 `f+37` 处和 `%rbx` 存储的 `m` 进行了 `f(n - 1, m) + m - 1` 的算术运算，结果在 `%ebx` 中。注意接下来复制 `%ebx` 到 `%eax` 并把 `%ebx` 算术右移 31 位，相当于填满了 `%ebx` 的符号位。我们知道指令 `idiv` 的行为是这样：以 `%ebx:%eax` 连接作为被除数，操作数作为除数，计算除法，商在 `%eax` 中，余数在 `%ebx` 中（这样的行为和计算机中除法器的实现方法有关，以后会学到）。于是这个填满符号位的操作相当于给 `%eax` 作符号扩展，并作为被除数。`idiv` 除以 `%rbp` 即 `n` 之后的余数加 1 作为返回值返回了，因此递归主体的两空填写 `int r = f(n - 1, m);`、`return (r + m - 1) % n + 1`。

填写栈的分布依然是难点。首先我们知道，第一次 hit `retq` 是在第一次遇到递归终止条件，也就是 `f(4, 3) -> f(3, 3) -> f(2, 3) -> f(1, 3)` 的时候。其次划分栈帧，我们注意题目中给出的栈的增长间隔是 4（而不是 8），因此每次递归调用会产生高度 6 个空的栈帧

（2 个是返回地址，都是 0x400505，2 个保存 %rbp，2 个保存 %rbx）。再次注意题目中给出的栈指针是 ..340，而在返回前栈指针已经被加回去了，所以最后一个递归调用过程的栈帧在 ..340 的下面！根据汇编代码可以知道每次在栈上保存的就是上次调用的 n，m 值。综上所述可很快填出栈的分布如下（注意小端法，低位填在低地址）。

- ▷ 0x7fffffffe38c X
- ▷ 0x7fffffffe388 X
- ▷ 0x7fffffffe384 X
- ▷ 0x7fffffffe380 X
- ▷ 0x7fffffffe37c X
- ▷ 0x7fffffffe378 X
- ▷ 0x7fffffffe374 0x0
- ▷ 0x7fffffffe370 0x400505
- ▷ 0x7fffffffe36c 0x0
- ▷ 0x7fffffffe368 0x4
- ▷ 0x7fffffffe364 0x0
- ▷ 0x7fffffffe360 0x3
- ▷ 0x7fffffffe35c 0x0
- ▷ 0x7fffffffe358 0x400505
- ▷ 0x7fffffffe354 0x0
- ▷ 0x7fffffffe350 0x3
- ▷ 0x7fffffffe34c 0x0
- ▷ 0x7fffffffe348 0x3
- ▷ 0x7fffffffe344 0x0
- ▷ 0x7fffffffe340 0x400505
- ▷ 0x7fffffffe33c 0x0
- ▷ 0x7fffffffe338 0x2
- ▷ 0x7fffffffe334 0x0
- ▷ 0x7fffffffe330 0x3
- ▷ 0x7fffffffe32c X
- ▷ 0x7fffffffe328 X
- ▷ 0x7fffffffe324 X
- ▷ 0x7fffffffe320 X

参考答案五

1. 依次为 RISC、CISC、RISC、RISC、CISC、CISC、RISC、CISC、RISC、RISC、RISC、CISC。这里对于 1，固定长的指令译码电路简单；通常 CISC 有变长指令，例如 x86 中清零用 xor，因为这样可以节省指令长度。对于 10，嵌入式系统是指可穿戴设备、物联网之类的小设备，功能

简单且需要省电，因此处理器电路不用太复杂。

2. `f(A, B) = (!A && B) || (A && !B)`。

3. 首先确定 `call V` 和 `jXX V` 两个指令都长 9 字节，然后按运算逻辑进行填写即可。没有填写的空格表示不需要填。

		call	jXX
Fetch	icode:ifun	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]
	rA, rB		
	valC	valC <- M8[PC+1]	valC <- M8[PC+1]
	valP	valP <- PC+9	valP <- PC+9
Decode	valA		
	valB	valB <- R[%rsp]	
Execute	valE	valE <- valB+(-8)	
	Cond Code		Cnd <- Cond(CC, ifun)
Memory (M)	valM	M8[valE] <- valP	
Writeback	dstE	R[%rsp] <- valE	
	dstM		
PC	PC	PC <- valC	PC <- Cnd?valC:valP

4. 此题顺着上一题的填写就很容易写出了。

```
int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Cnd: valC;
    icode == IRET : valM;
    1: valP;
];
```

5. 分别是错误（深度会影响冒险问题和电路复杂性，以及吞吐量有上界）、正确、正确、错误（load-use 冒险）、正确。

6. 总延迟为 $100 + 100 + 100 + 3 \times 10 = 330$ ps，吞吐率为 1/330 = 9.09 GIPS。

7. 这里有多条数据通路，其总延迟由最高延迟的通路 A → F → G → H 决定，延迟是 $40 + 60 + 40 + 30 + 10 = 180$ ps。插入寄存器使得其成为二级流水线，要求每一个通路都成为二级流水线，相当于多条单通路流水线求 max min。这里，通路 A~D 应当在 BC 之间插入寄存器；通路 E~H 应当在 FG 之间插入寄存器，此时各条数据通路都已经是二级的，最高单级延迟为 $40 + 60 + 10 = 110$ ps，吞吐量 9.09 GIPS。对于三级流水线也是同样的道理，先考虑最慢通路 AFGH，它的两个寄存器应该放到 AF 和 FG 之间，此时 EFGH 的寄存器只能插在 EF 间（否则 AFGH 成为四级）。ABCD 通路的寄存器应插在 AB、BC 之间。综上，插入位置是 AB、AF、EF、BC、FG 之间，最高单级延迟 80 ps，吞吐量 12.5 GIPS。

8. (1) 为 2 个。第 5 个时钟周期结束以后，第 3 行代码才能开始译码。而原来第 3 行在第 4 个时钟周期开始译码，因此需要 2 周期停顿。(2) 为 3 个。1、3 两行，2、3 两行均有数据相关。为了解决 1、3 两行的数据相关，需要额外的 2 个停顿；为了解决 2、3 两行的数据相关，需要额外 3 个停顿，因此需要将第 3 行的指令停顿 3 个周期。(3) 为 6 个。1、2 两行的数据相关，需要额外 3 个停顿才能解决；2、3 两行的数据相关，需要额外 3 个停顿才能解决，因此需要将 2、3 行的指令各停顿 3 个周期，共 6 个周期。

9. 此题帮助你理解并记忆 `pipe.hcl` 的实现。

第一问。`ret` 指令在通过 D、E、M 三个阶段时，流水线都有问题（都在取错指令）：其中 D 阶段开始取错指令，M 阶段行将结束时才能拿到返回地址，进行转发。所以应该填写 `IRET in {D_icode, E_icode, M_icode}`。顺便一提，`load-use` 的访存指令通过 E 时，前一阶段的 D 才出现问题，而访存通过 M 之后才能转发信号。`mispredicted jXX` 影响最短，E 时可能发现预测错误，E 结束之后也就能确定下一条指令位置了。所以 `load-use` 的检测条件是 `E_icode in { IMRMOVQ, IPOPQ }`，等等。

第二问为 `ret` 的处理方法。它在 D 时，前面处于 F 阶段的指令已经出现了问题。那 F 阶段是 stall 还是 bubble 呢？我们指出 F 阶段是不可以 bubble 的（Y86-64 实现中的 bubble 是将寄存器设为 reset 状态；F 仅仅是预测的 PC，无法模拟出 `nop` 的状态）。这样我们令 F 阶段 stall（反复地尝试取指），根据暂停规律[1]，后一阶段应该是 bubble，其余 normal。这样一来 `ret` 到 E、M 两个阶段时，都是让 F stall，D bubble，实际上连续产生了三个气泡。等 M 结束后，下一次取指才能正常，冒险结束。所以答案为（注意对于 `ret`，这个表实际上是三个一样的表）

F	D	E	M	W
stall	bubble	normal	normal	normal

根据第一问的补充分析，第三问应该填 `E_icode == IJXX && !e_Cnd`。第四问的处理：`mispredicted jXX` 在 E 时，可能发现前两条指令都取错了。这时，我们需要让这两条指令消失，这两条指令即将（“下个瞬间”）分别进入 D、E 阶段，因此 D、E 设为 bubble，其他均为 normal。接下来正确指令的地址就知道了，冒险结束，F 可以正常进行。

F	D	E	M	W
normal	bubble	bubble	normal	normal

最后一问基本由上面的分析可得（冒险最早什么时候结束）。对于 `mispredicted jXX`，在 D 阶段，`valP` 进入 `d_valA`，在 E 结束以后就可以将正确的 PC（自增）转发，此时它在 `M_valA` 中。对于 `ret`，当 M 阶段结束以后，`ret` 才能从内存中取出正确的地址，因此所需要的数据在 `W_valM` 中。答案为

```
int f_pc = [
    M_icode == IJXX && !M_Cnd : M_valA;
    W_icode == IRET : W_valM;
```

[1]若某阶段是 stall，则下一阶段如无暂停必要就是 bubble，最后一个暂停后一定是 bubble；stall 的前一阶段一定是 stall。

```
        1 : F_predPC;
    ];
```

10. (2)、(3)、(4) 都不会，理由分别是浮点数加法运算没有结合律、foo 函数可能有额外副作用和别名问题（例如若 p，q，r 都指向同一个元素，那么两段代码行为不等价）。

11. 循环中产生数据相关的寄存器为累加器 %rax 和计数器 %rdx。由于每次 %rax 更新都需要等待一次较慢的加法或乘法，其所在的数据相关路径为关键路径，需要 k 个周期。因为循环每次处理 2 个元素，从而数据相关给出的 CPE 下界分别是 4 和 0.5。

参考答案六

7. 取指部分的填法是很基本的，需要获得 rA，rB，D，所以我们写 icode:ifun <- M1[PC]、rA:rB <- M1[PC+1]、valC <- M8[PC+2] 和 vavlP <- PC+10。译码部分也很简单，为 valA <- R[rA]、valB <- R[rB]。

在执行部分，rmmovq 都要计算地址偏移，因此 valE <- valB+valC，但是 cmrmovqXX 还需要处理条件码，所以还要填 Cnd <- Cond(CC，ifun)。最后 rmmovq 还需要访存，即 M8[valE] <- valA；而 cmrmovqXX 除了访存 valM <- M8[valE] 之外，还需要条件写回 if (Cnd) R[rA] <- valM。

10. 首先，每级流水线都是 3 ns，然后画出时序图就可以了。一般地，需要时间是一级流水线的总时间，补上一头一尾。这里需要的时间为 $10 \times 3 + 3 + 3 = 36$ ns。

12. 本质还是 ret，只是需要稍微添加一下条件判断。取指是 icode:ifun <- M1[PC]、valP <- PC+1。译码需要注意，在 ret 中两个值都会被取用，故填写 valB <- R[%rsp]、valA <- R[%rsp]。

执行阶段是要恢复栈空间，因此为 valE <- valB+4 和 Cnd <- Cond(CC，ifun)。这里用 valA 也可以。访存阶段需要读出返回地址，请注意，这里必须用 valA（参看 SEQ 实现图中的数据通路），填写 valM <- M4[valA]。随后的写回和更新程序计数器的工作主要是关心条件（是否真的要返回了），所以分别是 if (Cnd) R[%rsp] <- valE 和 PC <- Cnd? valM:valP。

参考答案七

1. SRAM 较快，复杂且贵，而 DRAM 相对反之。所以 1 属于 SRAM 而 2、3、4、5 都是 DRAM。6、7 都是 RAM 的属性，而 SDRAM 是一种 DRAM 的变种（SDRAM 是同步动态随机存取存储器的缩写）。

2. AB，属于基本常识题。

3. 因为是双面磁盘，所以计算 $2\times2\times10000\times400\times512=8\,192\,000\,000\,\text{Byte}=8.192\,\text{GB}$，注意一般较慢的存储介质（例如磁盘容量、网速等）都是用十进制的数前缀。

4. 平均地看，需要旋转半周。又磁盘主要的延迟来源于寻道和旋转，所以结果为 $6+0.5\times(60/7500\times1000)=10\,\text{ms}$。

5. 答案为 60/6000（转一圈的时间）× 1/400（转过一个磁道的时间）× 1000 = 0.025 ms。

6. 数组 `A`，`B` 都存在逐元素访问的过程，因此都体现了空间局部性。但 `B` 存在同一个元素反复访问而 `A` 没有。因此 1 错误而 2、3、4 均正确。顺便一提：任何一个有意义的程序的指令都体现时间局部性（程序计数器）和空间局部性（顺序执行）。

7. 总容量等于组数乘以路数（行数）乘以块大小，其中直接映射高速缓存路数为 1，而全相联高速缓存的组数为 1。因此第一问为 $2^3\times2^10/2^5=256$ 组，1 行；第二问为 1 组，$2^3\times2^10/32=256$ 行。对于第三问，组数为 $2^4\times2^10/(2^6\times4)=64$ 组，所以组索引占据 6 位，偏移量占据 6。所给出的地址为 `1100 1010 1111 1110`，即组索引是 `101011`，即 43。

8. 1 正确，相当于在 cache 中直接增加一组空间。2、3 都错误，由于总容量不变，所以组数都会降低，命中率都可能降低。实际上实验表明，增大路数一般会导致命中率降低，而增大块大小对于不同的访存步长可能造成命中率升高或降低。4 正确，对于某些特定的程序，LRU 的期望性能可能不如随机替换（参看 2018 年期中第 13 题的构造），实验表明在较小的 cache 中随机策略更优。

9. 处理 cache 命中率的题，首先确定 tag, index 和 offset 的分布，然后先尝试通过组索引简化分析，**获得各地址放置位置的规律**，最后再直接模拟。对于本题，首先知道每次访问 4 个整数，cache 的每行（路）都能容纳 1 个元素或者 2 个元素（视小问而不同）。一共访存 100 次，只需要计算命中的次数。我们下面用下标 0、1、2、3... 来代指数组中的元素。注意缓存是写分配的，因此即便是写内存也有访问高速缓存的问题（一般我们假设缓存是写分配且写回的）。

第一问。计算得一共 2 组，每组 1 行，所以最低三位中，高 1 位是组索引，剩下 2 位是偏移。因为是直接映射，且每行只能容纳一个整数，所以组索引其实也代表了元素在数组的下标，因此实际上两行分别存储下标为奇数和偶数的元素。第一次 0、1、2、3 均不命中，此时 2、3 留在缓存中。接着访问 1、2、3、4，但 3 会被 1 替换，只有 2 能命中，结束时 3、4 留在缓存中。再次访问 2、3、4、5，但 4 会被 2 替换，只有 3 能命中。由此发现后 24 组访问各恰有一次命中，命中率为 24%。

第二问。计算得只有 1 组，组内有 2 行，因此只有最低 2 位代表组内偏移，每行只能容纳一个整数，因此是依次填满并 LRU 替换。第一次 0、1、2、3 均不命中，此时 2、3 留在缓存中。接着访问 1、2、3、4 时，首先 1 不命中，替换 2，然后 2 不命中，替换 3，最后 4 不命中，替换 2，以此类推。由此发现这个访存序列发生了抖动，每次都不命中，命中率为 0%。

第三问需要一些细致的模拟。计算得有 2 组，每组 2 行，每行都能容纳两个连续的整数。在这里，组索引表示的是下标模 4 的结果，也就是说 0、1、4、5... 会进入一组，而 2、3、6、7... 进入另一组。我们预期访存序列模 4 会有某种规律。模拟过程如下（你可能没必要模拟这么长）：

(i) 0、1、2、3，其中 0、2 不命中，结束时第一组有 0、1，第二组有 2、3。
(ii) 1、2、3、4，其中 4 不命中，结束时第一组有 0、1、4、5，第二组有 2、3。
(iii) 2、3、4、5，全部命中。
(iv) 3、4、5、6，只有 6 不命中，结束时第一组有 0、1、4、5，第二组有 2、3、6、7。
(v) 4、5、6、7，全部命中。
(vi) 5、6、7、8，只有 8 不命中，此时开始替换，第一组有 8、9、4、5，第二组有 2、3、6、7。
(vii) 6、7、8、9，全部命中。
(viii) 7、8、9、10，只有 10 不命中，再次替换，第一组有 8、9、4、5，第二组有 10、11、6、7。
(ix) 8、9、10、11，全部命中。
(x) 9、10、11、12，只有 12 不命中，替换得到第一组有 8、9、12、13，第二组有 10、11、6、7。这样就大致找到规律了，不命中只可能是在 $0, 2, \ldots, 24, 26$ 时发生，这里一共有 14 次，因此命中率为 86%。最后一次访存是 24、25、26、27，是全部命中的，根据 (iv)、(vi)、(ix) 的规律，其中剩下 cache 的部分应该对应前四个元素，即 20~23 都在缓存中。综上所述，地址 `80` 起的 8 个字节都在缓存中，所以缓存可以画为

	有效位	内容	有效位	内容
组 0	1	M[96-103]	1	M[80-87]
组 1	1	M[88-95]	1	M[104-111]

第四问考察冷不命中。当缓存充分大时，至少第一次访问该元素时是要发生不命中的，而这里一共有 28 个元素要访问，所以最高命中率是 72%。

参考答案八

9. CD。对于全相联缓存，操作主要和替换策略有关。LRU 和 FIFO 的行为是一样的，并且发生抖动（第 n 次访问会替换第 $n+1$ 次访问需要的块），因此不会发生任何命中。随机替换策略中，每次新来一个块，随机换出的块会导致之后的四次访存有一次 miss，且是等概率地有一次。因而平均 2.5 次访存会发生一次 miss，所以命中率是 60%。LIFO 策略中，前 3 个内存块的访问一直会命中，而最后两个块交替发生 miss，所以命中率也是 60%。

10. C。计算得到每个缓存块可以存放 $256/(4SE) = 8$ 个整数 $(N, M) = (64, 4)$ 时，访问数组第一行元素会将第一、二行元素都放入缓存，因此在访问数组第二行元素 `A[1][0]` 时不会 miss；而 $(N, M) = (2, 128)$ 时，同理可得访问 `A[0][1]` 时不会 miss。利用组索引简化分析，可以验证 $(N, M) = (32, 8), (16, 16)$ 时都会发生全部 miss。

参考答案九

1. C。课本中的 `HCL` 不是硬件描述语言而是硬件控制语言，它没有描述寄存器等模块的机制。

2. C，属于基本常识。

3. B。模拟退火属于很简单的通用启发式算法，搜索效率一般比较低，而划分法则是对问题简单的启发式近似，结果不会太好。

4. A。问题的模型并不是最短路，还需要决定线路的摆放位置，可能有约束：不能增加面积。其简化模型确实有一个子问题是最短路。

5. 此题具体写起来非常麻烦，具体思路简述如下，仅供参考。首先将宿舍楼和教学楼各分为 10 组，对应 10 个食堂。优化目标是最小化所有多边形的中心的 Euclid 距离，约束包括：不能重叠，共 $\Theta(n^2)$ 个约束，每个矩形的四个顶点都不能在剩下 89 个矩形的边界和内部；在校园内部，共 $\Theta(n)$ 个约束，每个矩形的四个顶点都必须在多边形曲线内部；靠近不同的食堂，各组矩形的中心距离 10 个食堂矩形的距离应各有 9 个不等式约束，表示其距离该组对应食堂的距离最小。

参考答案十

1. 根据课本描述，从上到下分别是 `cpp`，`cc1`，`as`，分别得到的后缀名为 `.i`，`.s`，`.o`。

2. (1) 错误，处理静态库只会拷贝被用到的模块，特别地，只会解析当前用到的符号，因此其顺序很重要。(2) 错误，对于模块文件，会把所有函数都拷贝，因此其顺序一般不重要。(3) 正确，参见 (2) 的分析。(4) 错误，例子可以参看书中练习 7.3 的 C。事实上，如果引用成环的话，至少有一个静态库要写两次。

3. 第一问。对于 `x`，它是全局符号，在 `main.c` 中初始化了，为强符号，而在 `count.c` 中则为弱符号。`bar` 是强符号，但在 `main.c` 中只是声明而不是定义，因此严格来说不能谈强弱，但一定要谈则认为是弱符号。`ans` 在两个模块中均为过程中的静态变量，因此都要用随机数字进行区分；因为它是局部符号，所以无强弱符号的区分。答案为：

文件名	变量名	在符号表中的名字	是局部符号吗?	是强符号吗?
`main.c`	`x`	`x`	全局	强符号
	`bar`	`bar`	全局	弱符号
	`ans`	`ans.114`	局部	不填
`count.c`	`x`	`x`	全局	弱符号
	`bar`	`bar`	全局	强符号
	`ans`	`ans.514`	局部	不填

阅读程序可知，代码的工作是维护两个累加器，一个是 `foo` 中的，一个是 `bar` 中的，每次 `x` 都会加一。注意链接时选用强符号，所以两个模块共用同一个初始化为 1 的 `x`，从而两个 `a1`，`a2` 输出相同，答案为 `1`，`1`，`3`，`3`，`6`，`6`。

第二问类似，但是 `x` 都是各模块中过程外的静态变量，对于编译器来说，过程外静态变量

不需要区分，因此此时 x 无需加随机数字，其他都和上一问一样。

文件名	变量名	在符号表中的名字	是局部符号吗?	是强符号吗?
main.c	x	x	局部	不填
	bar	bar	全局	弱符号
	ans	ans.114	局部	不填
count.c	x	x	局部	不填
	bar	bar	全局	强符号
	ans	ans.514	局部	不填

对于这一情形，两个模块使用的 x 是独立的了，所以唯一的变化时 x 的自增 op 只作用于 main.c 中的 x，所以答案为 1, 1, 3, 2, 6, 3。第三问显然会发生链接错误，因为有两个强符号。

4. C，即局部变量。请注意这里说的是一定不需要，有一些非局部变量或者指令是可能需要重定位也可能不需要重定位的。当然，此题的说法不妥，重定位都是针对一条引用，而不是针对符号本身。说一个符号不需要重定位，应当是指其任何引用都不可能要重定位。

5. D。需要理解的是，全局符号本质上是一个内存地址处的数据，访问时会在相应位置读取内容（例如 mov 0xabc(%rip), %rax），因此 char main 这个符号对应 .text 节 main 开头的那个字节。虽然如此，如果在 f1.c 中打印 main，得到的会是其地址。

6. 第一问。k 是宏定义，在预处理过后就会被展开，不会出现；ans 是一个过程内静态变量，因此符号表中出现的是形如 ans.???? 的符号，ans 不会出现，类似可知 y.???? 不会出现，因为 y 是过程外静态变量；局部变量 acc, x, n 都不会出现；foo 是全局符号，出现。答案：

名称	k	ans	acc	foo	y.????	x	n
是否出现	否	否	否	是	否	否	否

第二问。注意节头表中，offset 指示节在目标文件中的偏移，而 address 是加载时应当放置的相对位置，二者不一样。Size 栏就是填数据大小，是容易的。对于 ans 和 y，它们都是静态变量，因此都是局部符号，进入 .bss 节（即 4）。对于 z，它是一个外部变量，只有声明而没有定义，所以是全局符号且归入 UND 伪节。对于 t，它是一个未初始化的全局变量，只有这类符号会进入 COM 伪节，它当然是全局符号。bar, main 都是全局函数，是全局符号，属于代码，所以进入 .text 节（即 1）。答案如下：

Num	Value	Size	Type	Bind	Vis	Ndx	Name
5	0000000000000000	8	OBJECT	LOCAL	DEFAULT	4	ans.1797
7	0000000000000008	8	OBJECT	LOCAL	DEFAULT	4	y
11	0000000000000000	52	FUNC	GLOBAL	DEFAULT	1	bar
12	0000000000000034	139	FUNC	GLOBAL	DEFAULT	1	main
13	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	z
15	0000000000000008	8	OBJECT	GLOBAL	DEFAULT	COM	t

第三问和第四问都是简单概念。诸如 printf 中的常量字符串进入 .rodata 节，而 .bss 不占空间，因此为 0 字节。

第五问为重定位，通用方法为：

▷ **相对** 计算当前引用的绝对地址（基址加上 offset），知道目标位置的绝对地址，后者减前者计算相对地址，然后补上 addend 得到答案。

▷ **绝对** 直接用目标的绝对地址并补上 addend 即可。

根据题给信息，该重定位条目为相对引用类型，offset 为 6c，addend 为 -0x4。又代码段起始地址为 0x400517，所以当前位置是 0x400517 + 0x6c，目标地址刚好是 .text 开头的 bar，所以结果为 0x400517 - (0x400517 + 0x6c) + (-0x4)，其绝对值是 0x70，转换为负数得到 1001 0000，所以符号扩展得到 0xffffff90。而此时 callq 的地址是 0x400517 + 6b = 0x400582，所以答案填

```
    0x400582: e8 90 ff ff ff          callq 400517 <bar>
```

第六问绝对重定位比较简单，由题意得到 .rodata.str.1.1+0xa 的地址是 0x40069e，所以其起始地址是 0x400694。

第七问考察程序加载，先执行的是 _start 函数，所以地址为 0x400430。

7. (1) 正确，这是动态链接引入的一个重要动机。早期有很多操作系统是基于静态链接的，但是慢慢被淘汰了。(2) 错误，必须编译成位置无关代码，因为加载时重定位放置的位置是不能保证的（相反，可执行文件可以不编译成 PIC，因为通常 Linux 系统给进程代码分配的起始位置是 0x400000）。(3) 错误，PLT 中本质是一些可执行代码，因此必须在代码段，而 GOT 必须能写，所以在数据段。(4) 正确，位置无关代码需要用到大量相对偏移，不能让这一点出现错误。

8. 题目第一问的填写方法和之前问题是类似的，惟 buf 的地址需要注意。因为 .data 节中只有 bufp1, buf，而 bufp1 从 0x4010 开始占据了 8 字节，所以计算立刻得到 buf 的相应字段填写为 0x4018。顺便指出，我们再一次看到，外部定义的内容在符号表中会出现在 UND 节中。

Num	Value	Size	Type	Bind	Ndx	Name
35	00...004024	4	OBJECT	LOCAL	24	count.1797
54	00...004010	8	OBJECT	GLOBAL	23	bufp0
59	00...00115a	78	FUNC	GLOBAL	14	foo
62	00...004018	8	OBJECT	GLOBAL	23	buf
64	00...0011a8	54	FUNC	GLOBAL	14	main
68	00...004028	8	OBJECT	GLOBAL	24	bufp1
51	00...000000	0	FUNC	GLOBAL	UND	printf@@GLIBC_2.2.5

我们知道 .interp 节存储的是动态链接器的绝对地址，而节头说明 .interp 有 28 个字符，所以还要填 4 个，即 /lib64/ld-linux-x86-64.so.2。注意 .bss 节声明为 NOBITS（根据我们的认识也应该这样声明），所以它存储时永远不占据空间，但运行时要分配空间，这由其 size 字段决定，即表中的 16 字节。

(2) 问考察静态链接重定位。重定位条目的 offset 为 0x12，addend 为零；目标地址是 4018，所以结果为 4018 - (11a8 + 12) = 0x2e5e。结合 main 重定位到 11a8 知道指令

地址是 11b8，根据小端法填写 11b8: 8b 15 5e 2e 00 00 mov 0x2e5e(%rip), %edx。从这里（结合 C 代码）我们也能发现 addend 的实际作用是修正真正的跳转地址到 buf[1]。下一小问是也类似，比较简单的做法是发现下一条指令的地址，即 %rip 为 11c6 + 7 = 11cd，所以目标是 11cd + e37 = 0x2004。直接用重定位方法也可以，即 addend 为 0x4，offset 为 0x21，得 x - (11a8 + 21) - 4 = 0xe37，故 x = 0x2004。

(3) 问为本题重点要练习的内容，即动态链接的过程。第一个重定位和上一问方法一样，答案为 11d2: e8 59 fe ff ff callq 1030 <printf@plt>。

从代码中我们可发现，PLT 表在代码段中，开头是一段和动态链接器有关的代码，然后顺次是各需要动态链接的函数的 PLT 条目，例如这里是 printf@plt。因为 PLT 条目一个 16 字节，printf@plt 相对于开头的偏移量恰为 16 字节，所以其条目为 PLT[1]（个人认为这里给出代码的真实性存疑，因为一般 PLT 表的前三条都是要被动态链接器占用的）。再来看 printf@plt，我们知道，第一条指令代表从 printf 的 GOT 条目中查出其地址，然后跳转过去，因此其 GOT 条目的地址是 0x2f9a(%rip)，即 3fd0。又前面的注释中写 # 3fc8 <_GLOBAL_OFFSET_TABLE_+0x10>，所以 3fd0 是 GOT + 0x18，因为 GOT 条目一个 8 字节，所以 printf 的 GOT 条目是 GOT[3]。

根据延迟绑定的过程，第一次跳入 printf@plt 时，其对应的 GOT 条目会被初始化为其 PLT 条目的第二条指令的地址，也就是 pushq $0x0 的地址 0x1036（注意这是可执行文件内部的相对地址，还没被装载）。根据 main 被重定位 11a8 得 GOT 条目是 55...551a8 + (1036 - 11a8) = 0x55...55036。跳回 PLT 的第二条指令后，后续的工作是准备跳到 PLT 开头的动态链接器入口，去真正地把 printf 真实位置填写到 GOT 中，这里 pushq $0x0 是把 .rel.plt 中 printf 的下标传给动态链接器，以便于知道是在重定位何函数。所以真正绑定后，*0x2f9a(%rip) 就是 printf 的地址，而 GOT 表的位置是 55...551a8 + (3fd0 - 11a8) = 0x55...57fd0，所以最后一空填写 *(long *) 0x555555557fd0。

参考答案十一

1. A，可能不报告，例如多个弱符号任选一个，于是有造成非预期行为或者运行时错误的风险。

2. 第一问比较简单，按照基本的理解进行就可以，不要忘了未初始化的全局变量进入 COM。注意此题后两栏的填写要求的是“定义的模块”及其所在的节，因此 foo.o 中的 buf 虽然是 extern 变量，但它在 m.o 中定义，所在的位置是 .data 节。如果要问 buf 在 foo.o 中的位置，则答案为 UND。

第二问有两种做法。一种是根据重定位条目计算，一种是直接根据链接完毕的程序中的信息。我们先讲后者。注意到 %rip 是程序计数器，因此题中访问全局变量或者静态变量时所使用的偏移本质上就是目标地址减下一条指令的地址（看汇编）。注意最后一个空我们可以结合汇编代码和 C 代码判断出重定位的是 count 的引用，从而各空依次答案为 1000 - ff6 = 0xa（注意这个填写时要用小端法，且补足 4 字节 0a 00 00 00），2330 + 4 = 0x2334，3028 - 100f = 0x2019，2338 - 1022 = 0x1316 和 3024 - 1048 = 0x1fdc。

符号	是否为.symtab 条目	符号类型	定义符号的模块	节
bufp0	是	全局	foo.o	.data
buf	是	外部	m.o	.data
bufp1	是	全局	foo.o	COMMON
foo	是	全局	foo.o	.text
temp	否	/	/	/
count	是	局部	foo.o	.bss

传统的做法首先要确定 addr(s)。注意这里需要重定位的引用都是指令，所以 m.o 中的 .text 绝对地址是 0xfe8，而 foo.o 中的 .text 的绝对地址是 0x1000。对于第 ① 个，目标为 0x1000，偏移量 0xa，addend 为 -0x4，所以结果是 1000 - (fe8 + a) - 4 = 0xa（填写方法同上）。第 ② 个是绝对引用，2330 + 4 = 0x2334 即可。第 ③ 个的目标为 0x3028，偏移量 0x7，addend 为 -0x8，所以结果是 3028 - (1000 + 7) - 8 = 0x2019。剩下的皆同理，不再赘述。

9. B，显然初始化的静态变量要出现在 .data 中，符号表自然也要修改。但注意不要误解了 .rel.data 的含义，这里 .rel.data 是不会发生变化的，因为 count 初始化时，不需要引用别的非局部变量的地址，换言之它作为一个数据，初始化时不涉及重定位问题。别的位置引用它可能涉及重定位问题，但那是 .rel.text 的事情。另一方面，如果题目改为 static int *count = &x;，其中 x 为一个全局变量，那么 .rel.data 是会发生改变的。或者，再加一条 int y = count;，那么 .rel.data 也会改变，但这个条目对应的是 y。.rel.text 也不会发生变化，除非增加一条引用 count 的指令，例如 count = 3;，那么这条指令需要重定位，会在 .rel.text 中对应重定位条目。原则上 .strtab 也会发生改变（但我们一般不考虑这个）。

参考答案十二

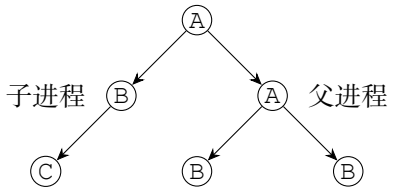
1. 属于基本概念题，注意故障可能返回也可能不返回，而且返回一般是返回到当前指令（例如缺页故障会返回再次执行，但除零等故障则终止程序）。中断的异步是指它一般是其他设备触发的，CPU 执行完当前指令之后才会去响应中断。

异常的种类	是否同步	可能的返回行为		
		重复当前指令	执行下一条指令	结束进程运行
中断（interrupt）	否		✓	
陷阱（trap）	是		✓	
故障（fault）	是	✓		✓
终止（abort）	是			✓

2. 分别为：陷阱（系统调用）；终止；故障（称为一般保护故障，segmentation fault 也属于此类）；中断（I/O 设备发起读写）；中断（I/O 设备发起读写，即使是 DMA 方式也是要中断的）；陷阱（系统调用）；故障。

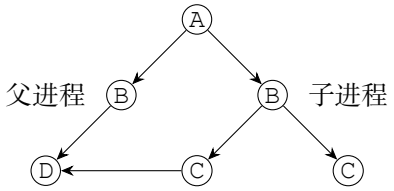
3. 父进程和子进程的 `a` 相互独立，注意自增自减运算符的位置，所以父进程输出 `p2: a=9\n`，而子进程输出 `p1: a=9\np2: a=8\n`。

4. 做此类题，画出简要的进程图（结点只需要标明输出）并进行拓扑排序即可。进程图如下：



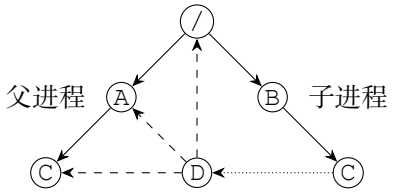
所以 `AABBBC`、`ABCABB`、`ABABCB` 是可能的，其他都不可能。

5. 遇到 `wait` 时，进程图中子进程结束前的最后一个输出向父进程 `wait` 后的第一个输出连一条有向边，然后再拓扑排序。进程图如下：



所以 `ABBCCD`、`ABBCDC` 是可能的，其他都不可能。

6. 信号被发送之后，接收的时间是不确定的（也可能丢失，不接收），某种意义上是一个悬空的结点，有多种连边的可能，讨论信号接收时间然后在进程图中连接边即可。这里 `SIGCHLD` 是子进程结束的信号，进程图如下：



所以 `ACBC`（父进程抢先结束）、`ABCCD`（父进程即将退出的时候收到信号）、`BCDAC`、`ABCC` 是可能的，其他都不可能。

7. (1) 注意 `fork()` 可能失败，返回 `-1` 表示之，所以必须将 `(pid = fork()) != 0` 改为 `(pid = fork()) > 0`，否则不能及时停下来。

(2) 因为代码中没有回收工作，所以子进程结束后会变成僵死进程，继续占用系统资源，所以 Bob 说错了。(3a) 则同理，父进程结束以后没有回收子进程，子进程一直在运行，占用系统资源，导致有第二次执行无法再创建更多子进程。对于 (3b)，由于父进程结束了，因此 `22725`

是第一个子进程的 pid，于是 22724 是第一个父进程的 pid。由于要 kill 整个进程组，子进程的进程组号均为 22724，所以填写 -22724。

最后一问和输入输出的缓冲区有关，printf 一般是行缓冲的，进程被 kill 为非正常退出，缓冲区的内容还没来得及写到标准输出，程序就已经结束了；所以需要在 printf 的字符串后面加换行符，令结果得到输出。

8. 本题帮助你理解输入输出缓冲的机制。

printf 是行缓冲的，每个程序都有一个全局的 buffer。因此第一个程序中，只有程序结束后才会输出全部字符串。因为 fork() 时整个进程上下文会被复制，所以 4 个进程的缓冲区最后都是 abc，故最后的结果是 abcabcabcabc。由于进程结束的先后不影响结果，故这是一个唯一的输出。

write 是无缓冲的，只要调用就会直接输出，所以 a 只会输出一次，b 两次，c 四次。第一个输出的字符当然是 a。第三问则和前两问同理，printf 的内容每个进程都会输出，而且最后输出，所以有 4 个 a、2 个 b 和 4 个 c，第一个输出的字符是 b。

9. fd1 和 fd2 对应独立的打开文件表条目，所以指针相互独立，前者写上 123 而后者从头写上 45，所以文件中得到 453，标准输出显然是 3 4（顺序分配文件描述符）。三级表图略，文件描述符分别指向两个条目，refcnt 均为 1。

10. 由于 fd2 被复制给 fd1，所以现在两个打开文件表条目是同一个，本质上是同一个文件读写指针，所以写出的结果是 12345。注意复制的是打开文件表条目，所以标准输出上还是 3 4。三级表图略，文件描述符指向同一个条目，refcnt 为 2。

11. 因为 fork 时文件描述符也被复制，因此父子进程用的描述符都是 3 号描述符，且二者共用同一个读写指针，只不过读写的顺序不确定，所以文件中可能是 45123 或者 12345。不过标准输出中的内容则顺序确定：因为 printf 是行缓冲，而这里一定是子进程先结束，所以输出为 C:3 P:3。

参考答案十三

1. C。A 显然是错误的，特权指令必须在内核模式下进行。B 中 option 的位应该用 | 运算符来连结，而不是用且，否则一般会得到 0。D 不是唯一的例外，SIGKILL 和 SIGSTOP 都是例外（不要把后者和 SIGTSTP 混淆了）。

2. A。注意 API 的参数顺序 dup2(old, new)，所以 old 的指针会复制给 new，new 的指向的打开文件表条目会变成指向 old 的条目，最后 0、1、4、10 四个文件描述符会指向同一个打开文件表条目。这里调用序列会向 10 号文件描述符复制，这是允许的，但是如果从没有打开的文件描述符复制，则确实会发生错误（但不会导致程序终止，只是 dup2 返回值表示出错，并且文件描述符的状态没有发生改变）。

3. 此题很有意思。

首先阅读代码可知，Bob 想要完成的是建立两个进程，然后父子进程互发信号，交替地在

文件中写 + 和 -（一共写 6 次）。对于第一问，根据提示我们知道 `fork` 之后可能调度子进程充分长时间，导致其未等到父进程的信号就结束了，因而不会有任何输出。如果需要等待信号，则要调用 `sigsuspend` 来安全地等待——这就是解决方案。

(2) 问初看非常神秘，为什么会在终端中打印字符？这说明必定有进程把字符向 1 号文件描述符写了。阅读代码可知，`fd` 是被初始化为 1 的！由此可见，子进程可能 `fork` 之后还没来得及打开 `file.txt` 就被其他进程抢占，然后接收到父进程的信号，结果 `fd` 还是个错误值，就在标准输出中打印了，其后它才打开 `file.txt`，使得 `fd` 变为正确值。至于为什么 `file.txt` 中的 + 号数量严重不足则很容易想到，因为父子进程的文件是 `fork` 之后才打开的，这样它们的读写指针相互独立，写的时候会发生相互覆盖，出现问题。

由此可见，当程序第一次在终端上输出 + 的瞬间，父进程的 `fd` 为 3，而子进程的 `fd` 则为 1，父进程的该文件描述符打开一个 `refcnt = 1` 的文件表条目，而子进程尚未有之。而当程序第一次在 `file.txt` 中输出 + 的瞬间，父进程和子进程的 `fd` 都为 3，而二者的描述符分别指向打开文件表中的不同条目，`refcnt` 皆为 1。Bob 期望的是二者的描述符表都指向打开文件表中的一个条目，`refcunt = 2`，这样才能达到互不覆盖的目的。

分析 `file.txt` 中的输出时，切忌在一行中模拟，这样很容易搞不清楚读写指针的位置。对于独立的读写指针，我们应该先写出它们分别试图写的内容，确定每个位置哪个指针最后写，然后用最后一次写的字符来确定最终的内容。我们先复现题干给出的 `file.txt` 的输出。父进程和子进程的写序列分别是

读写顺序	1		3			5				7					9					11					
父进程	-		-	-		-	-		-	-		-	-	-	-	-		-	-	-	-	-	-	-	
子进程		+			+			+			+						+								+
读写顺序		2			4			6			8					10									

每个字符我们都选择后写的那个，在上面用阴影标出，可以看到确实得到了题干中的序列。同样的道理，对于 (3) 问，我们知道在子进程正确打开 `fd` 之前它输出了两次，所以序列可以画成：

读写顺序	1		2		4			6				8				10						
父进程	-		-	-	-	-	-	-				-	-			-	-	-	-	-	-	-
子进程		+		+			+			+				+				+				
读写顺序		3			5			7				9										

我们便得到 (3) 问的解答为

```
++++++-+++--++++------
```

如果本题直接在一行上模拟，效率是非常低的（因为要多次修改指针位置和写入的内容，容易粗心犯错）。

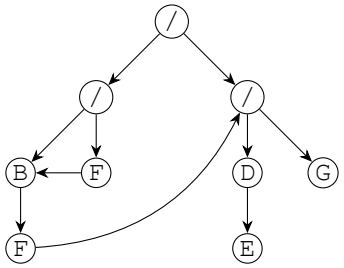
最后一问是容易的，因为父进程第一次需要打印 2 个 - 而子进程需要打印 1 个 +，所以 `Z` 处填写 `count++`，让父进程第一次打印两个。

5. B。`fd1` 和 `fd2` 的指针是一回事，因此每次在文件中写入 `66662333`，一共 5 次。而 `fd3` 每次循环都重新打开，指针都定位到开头，所以结束时前 4 个字符会被覆盖为 `hhhh`，于是最后有

$5 \times 4 - 4 = 16$ 个 `6`。

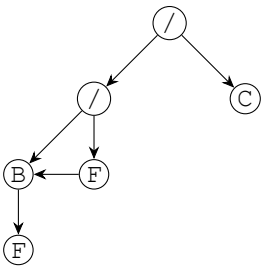
6. C。有缓冲和无缓冲的输入输出函数不能混用，否则最后输出的顺序是不对的。对于 B，注意并没有说未关闭 `stdin`，`stdout`，`stderr`，如果提前关闭了，则可能返回较小的文件描述符。D 显然是错的，进程的描述符表独立，但打开文件表系统只有一张。

7. 第 (1) 问是代码相对比较复杂的 fork puzzle，但是进程图仍然很简单，仔细分析 `wait` 的关系即可，如下：



因此全部可能的输出为 `FBFDEG`，`FBFGDE`，`FBFDGE`。

(2) 问更加简单一点，进程图如下：



所以 `CFBF`，`FCBF`，`FBCF`，`FBFC` 都是可能的。

13. A。B 从来没有两个指针一说。`read` 作为一个比较底层的函数，需要人工添加字符串结尾，C 错误。D 需要注意执行重定向后，1、4 对应的打开文件表条目的 `refcnt` 是 2，所以关闭 4 只会导致 `refcnt` 下降，文件仍然可用（直到 `refcnt` 降为 0，系统才会释放相关资源）。

14. B。因为 `printf` 是行缓冲，子进程睡眠 1 秒，所以最先输出的一定是父进程的 `lol`，这样便可选出答案，但后面的分析也是很平凡的。

15. `fork() : fork() ? fork()` 看上去复杂，实际上非常简单，就是产生了 4 个进程，进程树两边是完全对称的（无论如何都要再 `fork` 一次）。而且我们注意到，每边打印的字符顺序是严格的（`sigprocmask` 中有系统调用，结束后是从内核模式回到用户模式，立刻处理信号，不会错过）：父进程需要等子进程，所以先出现 `S`，父进程再进入信号处理程序，打印 `2`，最后打印 `P`。所以所有可能的结果是穿插两个 `S2P` 得到的。枚举知道至少有 5 个答案：`SS22PP`、`S2PS2P`、`SS2P2P`、`S2S2PP`、`S2SP2P`。

参考答案十四

1. 每张页表占据一页，所以每个页表有 $2^{12}/8 = 512$ 个条目。所以每一级 VPN 为 9 位，去掉 VPO 的 12 位，至少需要六级页表。页大小的更改则一般是基于页表级数的更改（和硬件有关），所以增加页的大小就是每次给 VPO 添加 9 位，所以剩下的两个页大小为 $2^{12} \times 2^9 = 2^{21} = 2\,\text{MB}$ 和 $2^{12} \times 2^{18} = 2^{30} = 1\,\text{GB}$。

2. 首先根据开头指明的每张页表占一页知道两级 VPN 都是 10 位，VPO 为 10 位。

第一问。拆分时注意，尽量不要增加二进制和十六进制转换的工作，因此只需要拆开 A，也就是说，马上可看出 VPO = 0x088 和 VPN2 = 0x297。由于这里高位都是 0，所以 VPN1 立刻看出是 0x2。所以 PDE 的地址是 0x00C188000 + 4 * 0x2 = 0x00C188008，该位置的内存内容是（注意用小端法）0D32A067。根据题给信息，最低的 8 位中，大页位为零，所以是两级翻译。故二级页表的起始地址是 0x0D32A000（留意每个页都是 4 KB 对齐的，页表项中就只需要记录物理页号，这是我们希望页表都恰占一页的原因）。结合 VPN2，二级页表项的地址为 0x0D32A000 + 4 * 0x297 = 0x0D32AA5C，读出此处的内存为 9A83C067，所以最终的物理页号是 9A83C，和 VPO 拼出物理地址为 0x9A83C088。鉴于其低 8 位是 0x67 = 0110 0111，所以不发生缺页，用户态程序可读也可写。

第二问。拆分得到的 VPN1 = 0xC0，VPN2 = 0x03C，VPO = 0x088。PDE 的地址是 0x00C188000 + 4 * 0xC0 = 0xC188300，读出一级页表项的内容是 9A8000E7。注意大页位是 1，所以是大页翻译，物理地址的 31~22 位是 1001 1010 10，其最高 4 位是 0000，所以其物理地址的 35~22 位是 0x9A83，和剩下 22 位拼在一起的到 0x9A83C088。

由前两问立刻得到第三问结果为 0，因为这两个虚拟地址对应的物理内存为同一块内存。

第四问涉及页表自映射，请回忆课上讲的内容。利用地址 C0002A5C算出 VPN1 = 0x300 以及 VPN2 = 0x2，因此 PDE 条目的物理地址为 0x00C188C00，在物理内存中读出第二级页表的起始地址为 0x00C188000（这里就发生了一次自映射，指向页表开头），于是计算出 PTE 条目地址为 0x00C188008，读出物理页号为 0x00D32A000，物理地址为 0x00D32AA5C，所以 %eax 的值为 0x9A83C067。为了指向第一级页表的第三个条目（VPN1 = 0x2），也就希望二级页表映射以后映射到的物理页就是第一级页表（即自映射两次）。因此前两次映射都应当映射到页表自己。注意到如果 VPN1 = 0x300 的话，那么第一级映射就映射到自己开头；所以如果还有 VPN2 = 0x300 的话，第二级映射也还是映射到自己开头。又对应的 VPO = 0x8，所以拼接得到虚拟地址 0xC0300008。

3. 若映射为私有的，则写时复制，因此答案相同均为 1；若映射为共享的，则两个进程共享更新，所以第一次子进程打印 1，然后父进程打印 2。

4. 首先阅读程序。父子进程都进行了内存映射。由于父子进程 fork() 结束之后状态是一样的，因此 mmap 映射到的虚拟地址空间相同；又它们的页表内容一样，所以 f 对应同一块物理内存（内容是 hello.txt 的前 8 个字符）。除非发生写时复制，否则二者将共享这块内存。其中，父进程中，hello.txt 映射为私有的可读写；子进程中，hello.txt 映射为共享的可读写。

完成 fork() 之后，子进程收到信号，进入信号处理程序 1，然后父子进程轮流执行信号

处理程序。子进程会依次在 f 对应的内存写 0123；每写一次，都会让父进程执行信号处理程序 2，父进程打印出 f 的内存映射中的内容。

当 Y 为空时，映射为私有的父进程从来不写内存，因此它会打印出子进程修改的内容。所以每次的标准输出分别是 0BCDEFG，01CDEFG，012DEFG，0123EFG，文件中的内容是 0123EFG。

而当父进程尝试在标记为私有的内存进行写时（Y 为 f[6] = 'X'）时，会触发 COW 机制，内容 0BCDEFG 会被复制，然后被修改为 0BCDEFX 打印出来（mmap(...MAP_PRIVATE...) will protect you from changes made to the file on disk）。由于后来两个进程关于 f 的物理内存是独立的，所以每行打印出的都是 0BCDEFX。留意到这段内存映射被复制之后，父进程对其的更改是不会反映到文件中的（否则导致一对多），所以文件内容还是反映为子进程的修改，即 0123EFG。

5. Linux 加载可执行文件时代码段一般都放在 0x400000 所以 (1) 应填写 r-xp。(2)、(3) 是程序创建的内存映射，其中 bye.txt 是共享可读写，所以 (2) 是 rw-s，而 hello.txt 是共享只读，所以 (3) 是 r--s。(4) 是动态链接器即 ld.so。结合加载可执行文件的内存布局知道 (5) 应该是栈，即写 stack。

注意到题目中给出的内核内存映射中，7fb5973a0000 处是标记为可写的。这说明该段发生写会触发 COW 机制：页表项表明不可写，触发故障，异常处理程序完成复制，然后再写。

参考答案十五

1. A。高速缓存是物理寻址的，所以一定不用刷新。用户态和内核态的转换不会改变内存映射，而上下文切换会改变虚拟内存到物理地址的对应关系，所以 TLB 在上下文切换时要刷新。

4. 解答之前先指出，此题原题条件不足，数据错误，叙述令人十分摸不着头脑，属于质量非常糟糕的题目。

易做出 (1)，VPN1 和 VPN2 分别占 6、10 位，所以 VPO 为 12 位，页面大小为 4096 字节。

现在考虑父进程执行 *b 的写时的读写问题。在解析 *b 时，第一步当然是找出一级页表项，因此从地址 0x67F0E8 读出 0x80AA32C4 就是从一级页表中，物理地址 0x67F0E8 的位置，读出一级页表项的内容 0x80AA32C4。根据题目对 PTE 结构的描述，我们知道 0xAA3 是二级页表页的页号，即 0xAA3000 是二级页表的起始物理地址。由于一级页表是 4 KB 对齐的，所以一级页表项的物理地址就是 0x67F000。

为什么会写两次？显然，我们知道 COW 机制在起作用，这段内存首先要被复制。0x80C3F110 这个数字当然是精心设计的，必然暗藏玄机。首先，一次写肯定是写 *b，那么另一次写呢？注意这次写也是父进程，因此和子进程无关。由于 COW 后内存映射需要修改，结合后面也有类似于 0xC3F... 的物理地址，我们可以推定这个实际上就是 COW 后，b 对应的二级页表项。这样我们马上知道，其二级页表项的物理地址在 0xAA3AD0，而 b 指向的真实物理地址是 0xC3F50D（页号 0xC3F）。这时我们就可以计算出 b 的值（即虚拟地址）了。一级页表项的偏移是 (0x67F0E8 - 0x67F000) / 4 = 0x3A，二级页表项的偏移是 (0xAA3AD0 - 0xAA3000) / 4 = 0x2B4，VPO = 0x50D，所以虚拟地址是它们的拼

接。注意不是直接将 16 进制拼接，虚拟页号为 `11 1010 10 1011 0100 = 0xEAB4`，所以 `b = 0xEAB450D`。

此题页表项中的最低 12 位是何含义，属于未解之谜。

6. B，因为两个变量的虚拟页号相同，所以一定在同一个物理页中，先后顺序取决于 `PPO = VPO`。

8. (1) 问计算页表大小是容易的。一共有 2^{20} 个页面，每个页面都需要一个页表项，所以答案为 $2^{20} \times 2^2 = 2^{22} = 4\,\text{MB}$。

在二级页表下，可以算出 `VPN1`，`VPN2` 都有 10 位，每个页表恰好 4 KB，有 1024 个页表项（即对应 1024 个页面）。所以图中 `VP0..VP2047` 需要两张二级页表，`VP10239` 需要独占一张二级页表（注意对齐），`VP10240..11263` 需要一张二级页表，加上一级页表一直是需要的，所以一共需要 5 页。

(2) 问需要结合 TLB 解决问题，留意 TLB 中存储的都是最后一级页表项，通过 `VPN` 就可以查出。由于 TLB 有 8 个项，且是直接映射，所以索引占 3 位，剩下 17 位都是 TLB 标记。对于第一次访问，其 `VPN` 为 `0xD7416 = 1101 0111 0100 0001 0110`，于是 `TLBI = 6`，`TLBT = 1 1010 1110 1000 0010 = 0x1AE82`，恰好命中 TLB，所以页表项是 `0x00A32027`，物理页号是 `0x00A32`，所以物理地址是 `0x00A32560`。这一过程中不需要访问二级页表项所在的内存位置。

下一个内存访问的 `VPN` 为，同理 `TLBI = 3`，`TLBT = 0x00802`，没有命中 TLB，所以需要经过两级翻译。`VPN1 = 0x10`，`VPN2 = 0x13`，`VPO = 69B`。一级页表项的地址为 `0x0C23B00 + 4 * 0x10 = 0x0C23B40`，页表项内容为 `29DE4027`，用户态程序可以读写。进一步得到二级页表项的地址是 `0x29DE4000 + 4 * 0x13 = 0x29DE404C`，页表项内容为 `00BA4227`，即物理页号是 `0x00BA4`，物理地址是 `0x00BA469B`。需要注意的是，完成写之后，TLB 会被替换，而且因为写过，dirty 位会变为 1，所以 TLB 条目的内容是 `0x00BA4267`。

(3) 也是简单的，`cnt` 代表了进程树中进程所处的深度，所以最大值是 5。每次产生的子进程时，相对于产生进程的父进程都要修改 `cnt`，会触发 COW，也就是虚拟页对应的物理页（内存映射）被修改。因为一共新产生了 15 个进程，所以答案为 15。

9. AB，因为只有最后一级页表项有 dirty 的问题。当然，就事实而言，A 也是正确选项。

20. 此题的背景是页表自映射问题。32 位系统下，所有的 `PTE` 所占的空间刚好是 4 MB。如果将这些 `PTE` 连续地放在内存中，那么这 4 MB 内存空间对应的 `PTE`（“`PTE` 的 `PTE`”）刚好在一个 4 KB 页中，而这个 4 KB 在页目录表中刚好也占一项。如果能再合理地设置二级页表项的地址，那么就可使 `PTE` 的 `PTE` 所占的 4 KB 的内容与 `PDE` 所占的 4 KB 的内容完全相同。换句话说，二级页表每个表都是 1 个页面，这个页面当然也需要页表项，我们不难发现它的页表项其实就是 `PDE`，所以 `PDE` 可以“一人分饰两角”。页目录除了作为页目录之外，还作为整个大页表在内存中的二级页表。

题目具体分析如下（以下由张茂森提供）：首先明确用户进程要访问内存区域，用的只能是虚拟地址。由于

```
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P
```

这一操作使得一级页表的第 `PDX(UVPT)` 项指向了了一级页表（页目录）本身的物理地址，而这一项的内容本应该是第 `PDX(UVPT)` 个二级页表（也就是虚拟地址 `UVPT` 对应的二级页表）

的起始地址，因此第 PDX(UVPT) 个二级页表就是一级页表（页目录）。

同时，第 PDX(UVPT) 个二级页表的第 PDX(UVPT) 项　指向了了一级页表本身，而这一项的内容本应该是虚拟地址 UVPT + (UVPT >> 10) 对应的物理地址（注意 UVPT 低 22 位都是零，所以才可以这样写）。因此虚拟地址 UVPT + (UVPT >> 10) 对应的物理地址就是一级页表（页目录）的起始物理地址，也就是我们可以认为一级页表的起始虚拟地址就是 UVPT + (UVPT >> 10)。

要获取 va 对应的一级页表中的页目录项内容，注意到 va 对应的页目录项在页目录中的位置已经知道了（pdx = (va >> 22) & 0x3ff，也就是 va 的高 10 位），只要将一级页表的起始地址（虚拟地址）加上 pdx * 4 即可。这样得到前三空的答案为：

```
1    // get_pde(va) 获取虚拟地址 va 对应的一级页表 (页目录) 中的页目录项内容
2    unsigned int get_pde(unsigned int va) {
3        unsigned int pdx = (va >> 22) & 0x3ff;
4        unsigned int addr = UVPT + (UVPT >> 10) + pdx * 4;
5        return *((unsigned int*) addr);
6    }
```

要获取 va 对应的二级页表中的页表项内容，va 对应的二级页表项在该页表中的位置是 va 的中间 10 位，而该二级页表的起始物理地址是一级页表第 pdx（va 的高 10 位）项的内容。由于一级页表等价于第 PDX(UVPT) 个二级页表，因此一级页表第 pdx 项的内容就是第 PDX(UVPT) 个二级页表第 pdx 项的内容，也就是虚拟地址 UVPT + (pdx << 12) 对应的物理地址。因此可以认为 va 对应的二级页表的起始虚拟地址就是 UVPT + (pdx << 12)。综上所述，va 对应的二级页表项的虚拟地址为 UVPT + (pdx << 12) + (index << 2)，也就是 UVPT + ((va >> 12) << 2)。后两空的答案为：

```
1    // get_pte(va) 获取虚拟地址 va 对应的二级页表中的页表项内容
2    unsigned int get_pte(unsigned int va) {
3        unsigned int PGNUM = va >> 12;
4        unsigned int addr = UVPT + PGNUM * 4;
5        return *((unsigned int*) addr);
6    }
```

需要注意，此处不能直接调用上一问 get_pde 函数的返回值作为页表起始地址，因为其返回值是物理地址，用户进程是无法访问物理地址的。另注：题干有改动，注意 UVPT + ((va >> 12) << 2) 并不一定等价于 UVPT + (va >> 10)。

参考答案十六

1. gCount 是全局变量，lCount 是静态变量，都会存储在数据段，因此两个进程各一个，线程共用。vCount 是局部变量，但要求存在内存中，所以是在栈中。答案如下：

2. 线程 1 将 foo、bar 改为 1 以后被线程 2 打断，线程 2 将 foo 改为 2 以后被线程 1 打断，线程 1 再输出 2 1，线程 2 将 bar 改为 2，然后输出了 2 2。

3. 可看出有 w, x 的潜在反序。线程 1 先执行完 P(w) 后线程 2 连续执行 P(x), P(z), P(y)，随后二者就发生了相互等待。

4. 此题属于一种读者–写者问题的变种。因为已经设计了信号量，就比较容易。答案如下：

- ▷ mutex_stu_count 初值为 1。
- ▷ mutex_door 初值为 1。
- ▷ mutex_all_present 初值为 0。
- ▷ mutex_all_handin 初值为 0。
- ▷ mutex_test[30] 初值均为 0。

代码：

```
1   Teacher: // 老师
2       P(mutex_door);
3       从门进入考场
4       V(mutex_door);
5       P(mutex_all_present); // 等待同学来齐
6       for (i = 1; i <= 30; i++)
7           V(mutex_test[i]); // 给 i 号学生发放试卷
8       P(mutex_all_handin); // 等待同学将试卷交齐
9       P(mutex_door);
10      从门离开考场
11      V(mutex_door);
12
13  Student(x): // x 号学生
14      P(mutex_door);
15      从门进入考场
16      V(mutex_door);
17      P(mutex_stu_count);
18      stu_count++;
```

```
        if (stu_count == 30)
            V(mutex_all_present);
        V(mutex_stu_count);
        P(mutex_test[i]) // 等待拿自己的卷子
        学生答卷
        P(mutex_stu_count);
        stu_count--;
        if (stu_count == 0)
            V(mutex_all_handin);
        V(mutex_stu_count);
        P(mutex_door);
        从门离开考场
        V(mutex_door);
```

5. 都不会。假设 1 出现死锁，那么 4 个资源耗尽，而且每个进程都想要请求资源，但至少有一个进程有 2 个资源，矛盾。类似，假设 2 出现死锁，则全部资源都被消耗，而且每个进程都在请求资源，这样所需求的资源不少于 $m+n$ 个，矛盾。

参考答案十七

1. C。A 反了，Internet 是指全球 IP 互联网，是一个具体的实现。B 不对，集线器是广播的，大家都能看到。D 也不对，会发生反复的打包和拆包过程，因此帧头会有变化。

2. B，系统一般都实现了套接字接口。其他都是正确的。注意 D 不够准确，其返回的描述符是半打开的，并不是能马上读写。

4. C。A 不对，因为可以访问共享变量。B 不对，如果不需要同步则不一定。C 是对的，可重入函数真包含于线程安全的函数（举一个不可重入但线程安全的函数的例子？）。

6. C。注意 `i` 是传值的，因此没有引用；`msg` 则是通过指针 `ptr` 引用的。

7. (1) 问考察对课本内容的理解，根据题目的意思我们知道自增应该分 load, inc, store 三步（当然，本题考虑不周，实验表明不一定这样翻译！）。三句汇编为 (a)(c)(d)、(b)(e)(d)、(a)(d)(c)。

(2) 问比较有意思（但是只是理论上的，由于硬件和体系结构的缘故，实际情况和理论分析不会一致，所以本题是个错题）。首先，`2000, 2000` 当然是可能的（就是没有原子性问题的情况）。两个线程实际上执行了如下的

```
i++, j++, i++, j++, ...
j++, i++, j++, i++, ...
```

的序列。为什么会出现不足值的情况？就是说某个进程在执行 load, inc, store 的间隙中，另一个线程对 `i` 的操作都是全部无效的（会被写回覆盖）。我们可以构造 `1500, 1500` 的情形。比如线程 1 读出 `i` 后被 2 打断，2 进行了 500 次 `j++` 和 `i++`，此时`i = 500, j = 500`。然后 1 写回，得到 `i = 1, j = 500`。此时 1 再读 `j`，又被打断，2 进行了 500 次的 `j++` 和 `i++`，得到

`i = 501, j = 1000`。然后 1 写回，得到 `i = 501, j = 501`。这时 2 已经结束，1 还有 `i++` 和 `j++` 各 999 个，所以得到 `1500, 1500`。

做到这里我们可以发现两个执行序列可以通过一种“break window”的方式来构造可能的输出。这样可以构造出 `1001, 1001` 到 `2000, 2000` 的所有可能，于是猜测 (3)(4) 不可能。

让我们来考虑 1 的某个 `i++` 被打断的情形，我们知道最坏情况是 2 在这个窗口中完成了 n 次 `i++`，然后都因为写回失效了，由于自增交替进行，所以这中间至少会有 $n-1$ 次的 `j++` 被完成，而补上 1 被打断的那个 `i++`，我们发现损失的自增最多不超过有效的自增数目。所以 `2, 2` 绝对不可能。另外我们注意到开头的 `i++` 和 `j++` 至少有一个完全有效（也不会在被打断的过程中被用来抵消别的损失），结尾的两个也是这样（反证即可），所以 `i + j` 至少是 $2000-1+2=2001$。因此 `1000, 1000` 不可能。

但是 `1001, 1000` 是可能的，表明我们的估计是紧的。事实上我们可以考虑如下序列：

```
i++, j++, i++, j++, ..., i++, j++
     j++, i++, j++, ..., i++, j++, i++
```

首先让 1 完成一次 `i++`，剩下的 999 个配对的 `i++` 和 1000 对配对的 `j++` 均可用以下方法使得每对只造成一次自增：1 load 之后被打断，2 再 load 之后又被打断，然后二者分别写回同一个自增 1 的值即可。结尾的 `i++` 正常完成，所以 `i = 999 + 2 = 1001, j = 1000`。

(3) 问很显然，出现了反序的加锁，所以有潜在的死锁的问题。解决方案是将任何一个信号量的初始值改为 ≥ 2 的值。

9. D。IP 属于网络层协议，其传输是 best effort，不保证传输可靠。IPv6 协议的地址长度为 128 位，B 错误。C 则是明显的，IP 地址和域名之间属于“关系”但不保证是“映射”，更不保证是单射或者满射。

11. D。信号量的操作导致 `j` 的增加是原子的，也就是说实际上是 3 个 `j++` 和 3 个 `printf`（分两步）拓扑排序。这里我们需要假设 `printf` 是先读取 `j`，然后再输出，所以结果不一定是单调递增的。但无论如何，最后的输出结果一定会有一个是 3，因此 D 不可能。

13. 此题没有互斥关系，有两对同步关系，即停车之后才能开门以及关门之后才能开车，所以要设计两个信号量：开车信号量（初始化为 0）和开门信号量（初始化为 0）。这样即可填写代码：

司机进程循环执行以下命令	售票员进程循环指令以下命令
`P`(开车信号量)	`X`
启动车辆	关门
`X`	`V`(开车信号量)
正常行驶	报站名或维持秩序
`X`	`P`(开门信号量)
到站停车	到站开门
`V`(开门信号量)	`X`

17. D。1, 2 和 1, 3 都存在 `a, d` 的反序，会出现死锁。2, 3 的死锁出现在 `c, d` 的反序，比如 2 第二次要 `P(d)` 时还持有 `c` 的锁，但是 3 在获得 `c` 的锁之前不会释放 `d`。所以都是可能发生死锁的。

19. 此题是二重生产者–消费者问题。有两个互斥关系，即两个缓冲区的访问；有两对同步关系，即两个缓冲区的空和满。这样，一共需要六个信号量，两个互斥，两组同步（空/满）。空信号量初始化为容量（4、8），满信号量初始化为 0。这样即可填写代码：

PA 循环执行	PB 循环执行	PC 循环执行
X 从磁盘读入一个记录 P(b1_empty) P(b1_mutex) 从 Buff1 中取出一个记录 V(b1_mutex) V(b1_full)	P(b1_full) P(b1_mutex) 从 Buff1 中取出一个记录 V(b1_mutex) V(b1_empty) P(b2_empty) P(b2_mutex) 将记录放入 Buff2 V(b2_mutex) V(b2_full)	P(b2_full) P(b2_mutex) 从 Buff2 中取出一个记录 V(b2_mutex) V(b2_empty) 打印

20. C，因为协议不同，所以需要路由器来完成工作。另一方面，只有路由器属于网络层，能够处理 IP 协议。

23. C。这里的不安全问题就是 LUS 必须一步执行到位，否则出错（一个线程的 S 必须在另一个线程 L 之前）。所以如果存在 L_i, U_i 被打断给另一边的 L_i, U_i，就不是安全的。A、B、D 都出现了这样的问题。