

作业一

本次作业一共有3道大题：

- 第1题和第2题是整数与浮点数计算，考察对知识点的掌握；
- 第3题是实践题，通过与实际编程结合，考察对知识点的理解与应用。

Q1. 整数计算

Q1.1. 二进制表示

在所有由 4 个 1 和 4 个 0 组成的 8 位二进制整数（补码）中，最小的带符号数和最大的带符号数分别是多少？答案请以十进制表示。

带符号数的最高位（二进制位"从右到左"对应于"从低到高"）为符号位，0 表示正数，1 表示负数。因此最小的带符号数其符号位为 1；剩下的 0 和 1 按照贪心原则去排列。

最小的带符号数：10000111b = -121

最大的带符号数：01111000b = 120

Q1.2. 补码计算

X、Y 的数据位宽均为 16 位。已知 $[X]_{\text{补码}} = 0x0033$ ， $[Y]_{\text{补码}} = 0xDE5A$ ，则 $[X - Y]_{\text{补码}}$ 和 $[X + Y]_{\text{补码}}$ 的值，计算结果用十六进制补码表示。



快速完成十六进制数取反

给定一个四位二进制数 X ，其取反结果为 $\bar{X}$ ，恒有 $X + \bar{X} = 0xF$
方便记忆的值： $0x0 \xrightarrow{\text{取反}} 0xF$ 、 $0x5 \xrightarrow{\text{取反}} 0xA$ 、 $0x7 \xrightarrow{\text{取反}} 0x8$

于是有 $[-Y]_{\text{补码}} \xrightarrow{\text{取反}+1} 0x21A5 + 0x1 = 0x21A6$

按补码加法直接计算可得：

$$[X - Y]_{\text{补码}} = 0x21D9$$

$$[X + Y]_{\text{补码}} = 0xDE8D$$

Q2. 浮点数计算

Q2.1. 定点数与浮点数哪个多？

给定相同的字长（例如32位），能表示的定点数和浮点数哪个更多？请给出你的理由。

定点数中，不同的 01 串所表示所对应的数完全不同。

浮点数中：

- 1. 规格化数不同的 01 二进制表示所对应的数不同。
- 2. 非规格化数不同的 01 二进制表示所对应的数不同，正负 0 除外（两个不同的 01 串表示同一个数）。
- 3. 存在两个 01 串分别表示正负无穷。
- 4. 存在多个不同的 01 串表示 NaN。

综上，相同字长能表示的定点数较多。

Q2.2. 9位浮点数表示

假设存在一种符合IEEE浮点数标准的9位浮点数，由1个符号位、4位阶码、4位尾数组成，数值表示仍遵循 $V = (-1)^s \cdot M \cdot 2^E$ 。请在下表中填空：

描述	9位二进制表示	M （十进制表示）	E （十进制表示）	V （十进制表示）
3.5				3.5
大于0的最小浮点数				

$Bias = 2^{e-1} - 1 = 2^{4-1} - 1 = 7$

- 3.5（规格化数）：
 $3.5_{10} = 11.1_2 = 1.11_2 \times 2^1$ ，于是 $E = 1$ ， $M = 1.11_2 = 1.75_{10}$
- 大于0的最小浮点数（非规格化数）：
其二进制表示为 0 0000 0001，那么 $M = 0.0001_2 = 2^{-4} = 1/16$ ， $E = 1 - Bias = -6$ ，值等于 $2^{-4} \times 2^{-6} = 2^{-10} = 1/1024$

描述	9位二进制表示	M （十进制表示）	E （十进制表示）	V （十进制表示）
3.5	0 1000 1100	1.75	1	3.5
大于0的最小浮点数	0 0000 0001	1/16	-6	1/1024

Q3. 实践题

Q3.1. 加法溢出检查

带符号数 `si1` 和 `si2` 符号位相同，先检查二者相加后是否会产生加法溢出：若溢出返回 `true`，未溢出返回 `false`，并将结果写进 `*sum` 中。

你需要填写 `__表达式(1)__` 和 `__表达式(2)__`（其中 `__表达式(1)__` 要求使用位运算实现，不能调标准库中 `INT_MIN` 的实现），并解释它的工作原理。

```
bool checkAddOF(int si1, int si2, int* sum) {
    unsigned usum = (unsigned)(si1) + si2;
    const unsigned MY_INT_MIN = __表达式(1__); // 用位运算实现
    if ((__表达式(2)__) & MY_INT_MIN)
        return true;
    else {
        *sum = si1 + si2;
        return false;
    }
}
```

要求通过位运算来为实现 `INT_MIN`，并赋值给 `MY_INT_MIN`。而 `INT_MIN` 的特征为除符号位为 `1`，其余位为 `0`，一般实现方式如下：

```
/* 表达式(1) */
// (1) 仅限于32位整数
const unsigned MY_INT_MIN = 1u << 31;
// (2) 通用实现，但涉及一次乘法运算（02能优化掉）
const unsigned MY_INT_MIN = 1u << (sizeof(unsigned) * 8 - 1);
```

加法溢出发生于两个同符号整数相加后，得到的结果与被加数异号，如**负数加负数得到正数**。最直接的做法是判断结果的符号位与被加数 `si1` 或 `si2`（因为它们同号）的符号位是否相同。对此我们可以使用异或运算`来实现：

 异或运算

对于两个位 `x` 和 `y`，如果它们同为 `0` 或同为 `1`，则 `x ^ y = 0`（相互抵消了）；如果不相同，则 `x ^ y = 1`。

<code>X</code>	<code>Y</code>	<code>X ⊕ Y</code> (即 <code>X</code> 异或 <code>Y</code>)
0	0	0
0	1	1
1	0	0
1	1	0

在C语言中，`x ^ y` 表示对 `x` 和 `y` 的每一位进行异或运算，自然包括了作为最高位的符号位：

```
/* 表达式(2) */
// (1) 参考解
unsigned result = usum ^ si1;
// (2) 另一种解答，保险但没必要
unsigned result = (usum ^ si1) & (usum ^ si2);
// (3) 如果被加数是正数，直接看usum符号位；反之是负数，usum是正数，就将其符号位取反
unsigned result = (si1 > 0) ? usum : ~usum; // -usum也行
```

可是我们最终只想看符号位的值：假设加法实际没有溢出，但是除符号位外的位非全 `0`（比如 `0x01234567`），在C语言的 `if ()` 中，所有非零值对应逻辑值为 `true`，则该溢出判断成立——出bug了！

还记得一开始准备的 `MY_INT_MIN` 吗？它只有符号位为 `1`，其余位为 `0`："与"上 `MY_INT_MIN`，保留符号位的同时置其余位为 `0`，这时的结果是否非零完全取决于符号位，符合我们的预期。

```
if (result & MY_INT_MIN) ... // 符号位异号，发生溢出
```

Q3.2. 字节序

在网络传输数据时，发送方需要将本地的数据通过转换为网络字节序后再发送；接收方接收后，需要转换为本地字节序后才能使用。

这个转换可以调用 `htonl()` 和 `ntohl()` 由网络序来实现。



`htonl()` 和 `ntohl()`

`hton` 表示由本地 (`host`) 字节序转为网络 (`network`) 字节序；后缀 `l` 表示 `long`，即32位整数（对应的 `s` 表示 `short`，即16位整数）。

```
// byte_order.c
#include <arpa/inet.h>
#include <stdio.h>

void printMemory(void *p, int size) {
    unsigned char *p1 = (unsigned char *)p;
    for (int i = 0; i < size; i++) {
        printf("%02x ", p1[i]);
    }
    printf("\n");
}

int main(int argc, char const *argv[]) {
    int a = 0x12345678;
    printf("Original: "); printMemory(&a, sizeof(a));

    a = htonl(a); // <1>
    printf("Try htonl(): "); printMemory(&a, sizeof(a));

    a = ntohl(a); // <2>
    printf("Then ntohl(): "); printMemory(&a, sizeof(a));
    return 0;
}
```

上述示例代码的编译与执行结果如下所示：

```
$ gcc byte_order.c -o byte_order && ./byte_order
Original: 78 56 34 12
Try htonl(): 12 34 56 78
Then ntohl(): 78 56 34 12
```

请确保你已按照实验导引在本地配置好Linux环境并测试过 gcc 。

Q3.2.1. 本地序 vs 网络序

根据上述执行结果，请问本地的字节序属于哪一种字节序？网络字节序又属于哪一种？

观察 `printMemory()` 的输出：当地址从低向高递增时，如果是低字节对低位，那么就是小端序 (little endian)；如果是低字节对高位，那么就是大端序 (big endian)。

- 本地：小端序
- 网络：大端序

Q3.2.2. 大小端序转换

试补齐下述宏定义中缺失的数值，使得 `htonl()` 能够正确工作，并解释该宏定义的功能。

```
// glibc/bits/byteswap.h

/* Swap bytes in 32-bit value. */
#define __bswap_constant_32(x)
    (((x) & __空格(1)__) >> 24) | (((x) & __空格(2)__) >> __空格(3)__) | \
    (((x) & __空格(4)__) << 8) | (((x) & 0x000000ffu) << __空格(5)__))
```

补充：在C语言中，`0x000000ffu` 的意思是将 `0x000000ff` 转换为无符号整数。注意观察，位运算的格式为 `((x) & 无符号十六进制数) >> 十进制常值`，按照这个格式补齐 `__空格(n)` 的值。

出于编程习惯，涉及位运算的操作都倾向于按无符号数处理；实际上不添加后缀 `u` 也能正常工作。

`htonl` 和 `ntohl` 的核心在于将小（大）端序转换为大（小）端序，主要实现在 `__bswap_constant_32` 宏中。

从 `(x) & 0x000000ffu` 可知取出最低8位，然后移到最高8位，也就是左移 $32 - 8 = 24$ 位，因此 `__空格(5)` 的值为 `24`

此时分析 `__空格(1)`，它应将最高8位移到最低8位上，这个可以通过 `& 0xff000000u` 完成，因此 `__空格(1)` 为 `0xff000000u`。

同理可以推断出 `__空格(3)` 是将次高8位移到次低8位，`__空格(3)` 填 `8`，而 `__空格(2)` 为 `0x00ff0000u`，`__空格(4)` 可知为 `0x0000ff00u`

最终实现如下：

```
// glibc/bits/byteswap.h

/* Swap bytes in 32-bit value. */
#define __bswap_constant_32(x)
    (((x) & 0xff000000u) >> 24) | (((x) & 0x00ff0000u) >> 8) | \
    (((x) & 0x0000ff00u) << 8) | (((x) & 0x000000ffu) << 24))
```

Q3.2.3. 思考题

如果调换 `a = htonl(a);`（注释 <1> 处）和 `ntohl(a)`（注释 <2> 处），程序的输出结果会有变化吗？请解释原因。

由于底层二进制数据的存储方式是一样的，因此在字节序翻转上并无本质区别。查看glibc实现可知其实二者通过 `weak_alias` 宏关联在一起：

```
// glibc/inet/htonl.c

uint32_t
htonl (uint32_t x)
{
    #if BYTE_ORDER == BIG_ENDIAN
        return x;
    #elif BYTE_ORDER == LITTLE_ENDIAN
        return __bswap_32 (x);
    #else
        # error "What kind of system is this?"
    #endif
}

weak_alias (htonl, ntohl)
```

? 什么是 weak_alias 宏

其目的是为函数添加一个"弱"别名，与"强"符号函数名区分。

```
//macro weak_alias
#define weak_alias(name, aliasname) _mweak_alias(name, aliasname)
#define _weak_alias(name, aliasname) \
extern __typeof(name) aliasname __attribute__((weak, alias(#name)));
//macro weak_alias end
```

在后续课程中会着重讲解，有兴趣的同学可以先阅读CSAPP第7.6.1节。

具体而言，如果调用"被 weak_alias "的函数（上例中为 `ntohl`）时，无同名的"强"符号函数实现，则会调用该别名对应的函数（即 `htonl`）。

作业二

本次作业一共有2道大题：

- 第1题是基础题，通过对比阅读源代码与汇编代码，巩固大家对课上介绍的汇编知识的掌握；
- 第2题是分析题，需要根据给定的情况进行分析，对课上所学进行补充。

Q1 基础

Q1.1 匹配 (3分; 1空1分)

```
int choice1(int x) { return (x < 0); }

int choice2(int x) { return (x << 31) & 1; }

int choice3(int x) { return 15 * x; }

int choice4(int x) { return (x + 15) / 4; }

int choice5(int x) { return x / 16; }

int choice6(int x) { return (x >> 31); }
```

对于下列的汇编代码段，从上述C代码中找出其对应的函数实现（填写函数名即可）。

为方便讲解，我们以（旧）、（新）表示 caller 和 callee 下的寄存器值。

观察汇编代码可知这是X86-32的汇编代码，是通过运行栈来传递参数的。在 caller 栈帧中保存了返回地址（在 [esp] 处）、若干参数（以 [esp + 4] 为始的连续空间）；调用函数后进入 callee，先保存当前 %ebp，并将 callee 的 %ebp(新) 置为 caller 的 %esp(旧) 值，从而创建新的栈帧。

分配 callee 栈帧后，此时参数 x 所在的地址为 %ebp(新) + 4 + 4 = %ebp(新) + 8。结束若干操作后，在返回前将返回值保存在 %eax 中，并还原 %esp(旧) 和 %ebp(旧)。

汇编代码	对应函数 (填写函数名即可)	说明
foo1: pushl %ebp movl %esp, %ebp movl 8(%ebp), %eax sall \$4, %eax subl 8(%ebp), %eax movl %ebp, %esp popl %ebp ret	choice3	关键操作简化为 $(x \ll 4) - x$ ，即 $x * (16 - 1) = x * 15$

汇编代码	对应函数 (填写函数名即可)	说明
<pre>foo2: pushl %ebp movl %esp, %ebp movl 8(%ebp), %eax testl %eax, %eax jge .L2 addl \$15, %eax .L2: sarl \$4, %eax movl %ebp, %esp popl %ebp ret</pre>	choice5	通过 testl 指令设置 ZF 和 SF（并清空 CF, OF），若 SF 为 0（表示正数），则直接跳转到 .L2 处执行 >> 4，否则先 + 15 再继续 这等价于向零舍入的带符号整数除法（见课件《2.1整数》P.28）
<pre>foo3: pushl %ebp movl %esp, %ebp movl 8(%ebp), %eax shr \$31, %eax movl %ebp, %esp popl %ebp ret</pre>	choice1	关键的地方是逻辑右移指令 shr，在C语言中对于 x >> 4，若 x 是 int 则进行算术右移（见 课件《2.1整数》P.27），若 x 是 unsigned 才进行逻辑右移：这里的参数是 int x。因此最合适的是 choice1 返回符号位（最高位）的值（0 或 1）。

✗ C语言中 >> 对应的汇编代码

带符号整数右移为算术右移

```
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.c
int foo3(int x) { return x >> 31; }

int main(int argc, char const *argv[]) {
    foo3(-123);
    return 0;
}
zyx@p100-2:/extradisk/zyx/intro2cs$ gcc -S -fno-stack-protector test.c -o test.S
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.S
.file "test.c"
.text
.globl foo3
.type foo3, @function

foo3:
.LFB0:
.cfi_startproc
endbr64
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl %edi, -4(%rbp)
movl -4(%rbp), %eax
sarl $31, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc

.LFE0:
.size foo3, .-foo3
.globl main
.type main, @function
```

无符号整数右移为逻辑右移

```
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.c
int foo3(unsigned int x) { return x >> 31; }

int main(int argc, char const *argv[]) {
    foo3(-123);
    return 0;
}
zyx@p100-2:/extradisk/zyx/intro2cs$ gcc -S -fno-stack-protector test.c -o test.S
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.S
.file "test.c"
.text
.globl foo3
.type foo3, @function

foo3:
.LFB0:
.cfi_startproc
endbr64
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl %edi, -4(%rbp)
movl -4(%rbp), %eax
shrl $31, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc

.LFE0:
.size foo3, .-foo3
.globl main
.type main, @function
```

Q1.2 for 循环 (4分; 1空1分)

我们实现了如下的C语言函数：

```
int foo(int* a, int n, int val) {
    int i;
    for (i = n - 1; i >= 0 && val != a[i]; --i);
    return i;
}
```

根据上述函数，填写下面汇编代码中缺失的部分：

```

foo:
    leal    -1( __空格1__ ), %eax
.L2:
    testl   __空格2__, %eax
    js      .L1
    movslq   %eax, %rcx
    cmpl     %edx, ( __空格3__, %rcx, 4)
    __空格4__ .L1
    subl     $1, %eax
    jmp      .L2
.L1:
    ret

```

FIXME： cmpl 指令的寄存器应为 %edx 而非 %rdx，在参考解答中已修正。

观察可知变量 `i` 对应寄存器 `%rax`，由此推断 `leal` 指令实际完成 `i = n - 1;`，而参数 `n` 对应 `%rsi`。

`testl` 指令则用于检查 `i >= 0`：通过按位与判断符号位并设置 `SF`。若 `SF=1` 则在 `js` 指令处发生跳转，直接结束。一般检查某个值是否为负数的表示为 `testl %eax, %eax`，即自身"与"上自身；不过 `testl` 允许 `Src operand` 为立即数，因此 `testl $0x80000000, %eax` 也是可以的，但 `Dst operand` 不能是立即数，因此 `testl %eax, $0x80000000` 是非法的。

`cmpl` 指令则对应 `val != a[i]`，考察的是 `a[i]` 的汇编表示：参数 `*a` 对应寄存器 `%rdi`，而在上一条的 `movslq` 指令中，`%rcx = (符号扩展)(%eax)`，又因 `sizeof(int) == 4`，因此 `(%rdi, %rcx, 4)` 等价于 `a[i]`。`cmpl` 比较是 `val` 和 `a[i]` 的值，如果等于则立即返回，对应的下一条汇编为 `je`。

空格	值
__空格1__	%rsi
__空格2__	%eax 或 \$0x80000000
__空格3__	%rdi
__空格4__	je

Q1.3 switch 语句 (5分, 1 case 1分)

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $3, %rdi
    je      .L8
    jg      .L3
    cmpq    $1, %rdi
    je      .L4
    cmpq    $2, %rdi
    jne     .L11
    movq    %rsi, %rax
    cqto
    idivq   %rcx
    addq    %rcx, %rax
    ret
.L3:
    movl    $1, %eax
    subq    $5, %rdi
    subq    %rdx, %rax
    cmpq    $2, %rdi
    movq    %rax, %rcx
    movl    $2, %eax
    cmovb   %rcx, %rax
    ret
.L8:
    movl    $1, %eax
    addq    %rcx, %rax
    ret
.L4:
    movq    %rdx, %rax
    imulq   %rsi, %rax
    ret
.L11:
    movl    $2, %eax
    ret
```

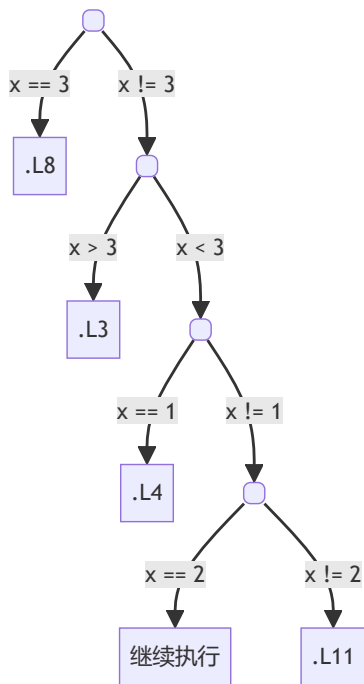
根据上述汇编代码，实现下方缺失的 switch 语句。

```
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch (x) {
        // TODO: 实现switch
    }
    return w;
}
```

具体来说，你需要将 switch 语句中各分支的 case 和执行语句填写在下表中。题目中填写了 default 及对应的语句作为示例。注意：视情况需要包含 break; ；若无则填 - 。

变量 x 在 %rdi，y 在 %rsi，z 在 %rdx，w 作为返回值在 %rax 中。

从汇编代码可得到 switch 语句的实际分支跳转：



需要注意在 switch 之前已经执行了 `int w = 1;`。这时分情况讨论：

- `x == 3`：跳转到 `.L8`
 - 先执行 `w = 1;`，再执行 `w = w + z;` 后就返回了；
 - `w += z; break;`
- `x > 3`：跳转到 `.L3`
 - 先执行 `x -= 5;`（实际没有修改 `x` 的值），在赋值 `%rax` 前令 `%rcx = w - z`，`%rax = 2`；执行 `cmpq $2, %rdi`，来判断 $(x - 5) < 2$ 是否为真（看 $(x - 5) - 2$ 是否需要借位，若需要则置 `CF = 1`，反之为 `0`）；若成立（对应于 `CF = 1`）则执行 `cmovb` 指令，令 `%rax = %rcx`，否则保持 `%rax`；
 - 注意，在判断时不会考虑是否为有符号数：`x = 4` 时，`x - 5 = 0xffffffff > 2`，`x > 6` 时，`x - 5 >= 2`，它们对应的赋值为 `w = 2;`，**注意只有 default 分支有立即数赋值**，因此对应于 `default` 分支；`x - 5 < 2`，只有可能是 `x = 5` 和 `x = 6`，对应的赋值为 `w = w - z;`，执行完毕后返回；

```
case 5:
case 6:
    w -= z;
    break;
// 其他情况与default一致
```

- `x == 1`：跳转到 `.L4`
 - 先执行 `w = z`，再执行 `w = w * y`，之后返回；
 - `w = z * y; break;`
- `x == 2`：继续执行
 - 先执行 `w = y`，再执行 `w = w / z`（对应于 `cqto` 和 `idivq %rcx` 指令），最后执行 `w = w + z` 后返回；
 - `w = y / z; w += z; break;`
 - 考虑到 `w += z; break;` 其实对应于 `x == 3`，可以按Fall through来写：

```
case 2:
    w = y / z;
case 3:
    w += z;
    break;
```

最终整合如下：

case	执行语句（若需要应包含 <code>break;</code> ；若无则填 -）
default	<code>w = 2; break;</code>
1	<code>w = y * z; break;</code>
2	<code>w = y / z;</code>

case	执行语句（若需要应包含 break; ；若无则填 - ）
3	w += z; break;
5	-
6	w -= z; break;

对应的 switch 语句：

```
switch (x) {
  case 1:
    w = y * z; break;
  case 2:
    w = y / z;
  case 3:
    w += z; break;
  case 5:
  case 6:
    w -= z; break;
  default:
    w = 2; break;
}
```

另外一种（冗余但直观的）写法：

case	执行语句（若需要应包含 break; ；若无则填 - ）
default	w = 2; break;
1	w = y * z; break;
2	w = y / z; w += z; break;
3	w += z; break;
5	w -= z; break;
6	w -= z; break;

```
switch (x) {
  case 1:
    w = y * z; break;
  case 2:
    w = y / z; w += z; break;
  case 3:
    w += z; break;
  case 5:
    w -= z; break;
  case 6:
    w -= z; break;
  default:
    w = 2; break;
}
```

i 观察什么时候 switch 才会真的使用跳转表

通过不停增加 含实际、独立的执行体的分支 数量，观察 switch 使用跳转表的时机。从中判断什么条件下 gcc 会倾向于将 switch 优化为分支，而不是创建跳转表来实现。

Q2 分析

Q2.1 整数 & 浮点数回顾（5分；1表达式1分，答案和解释各0.5分）

假设定义了以下变量：

```
int x, int y;
unsigned ux = (unsigned)(x);
unsigned uy = (unsigned)(y);

float f;
double d;
assert(!isnan(f) && !isnan(d)); // 保证 f 和 d 都不是 NaN
```

判断下面的表达式是否为真，若为假请解释或给出反例。

表达式	T/F	解释
<code>x == (int)(float)x</code>	F	<code>float</code> 尾数为23位，不足以完整表示 <code>int</code> ，因此会出现精度损失
<code>(d + f) - d == f</code>	F	反例， <code>d</code> 取一个 <code>double</code> 才能表示的大数， <code>f</code> 取一个 <code>float</code> 极小数，相加时大数吃小数，相减后近似0
<code>if (x <= 0) then -x >= 0</code>	F	$T_{\min}$ （即 <code>INT_MAX = 0x80000000</code> ）的负值还等于 $T_{\min}$
<code>x >> 4 == x / 16</code>	F	移位运算不会使结果向0舍入
<code>(x -x) >> 31 == -1</code>	F	反例， <code>x = 0</code>

Q2.2 条件跳转指令（3分；结果1分，解释2分）

X86-64指令集架构中的条件跳转指令 `jg` 是用于带符号数比较还是无符号数比较的？（1分）
其产生跳转的成立条件是 `~(SF^OF)&~ZF` 为真，请解释为何是这一条件。（2分）

有符号数比较；`jg` 是 greater (signed) 跳转，即 `if(a - b) > 0`；

两种情况：

- 当 `OF == 0` 时，则代表没有溢出，此时 `SF` 必须为 `1`，`SF` 为 `1` 代表结果为负，即 `a - b < 0`，代表 `a < b`；
- 当 `OF == 1` 时，则代表有溢出，而此时 `SF` 必须为 `0`，即结果最后为正数，那么此时则是负溢出，也可以得到 `a - b < 0`，即 `a < b`。

综上，`SF ^ OF` 则代表小于的意思，`ZF` 代表等于，所以 `~(SF ^ OF) & ~ZF` 代表大于。

Q2.3 分支跳转指示（5分；答对+解释 4分，优化1分）

在课上我们介绍了 `likely()` 和 `unlikely()` 指示宏，它们可以用于告诉编译器某个条件的成立概率，以便编译器进行优化。早期这两个宏是作为编译器专有扩展提供的，如今已被纳入C++20标准中，使用方式如下：

```
// branch_pred.cpp
#include <cstdlib>

int main(int argc, char *argv[]) {
    int a, b;
    /* Get the value from somewhere GCC can't optimize */
    a = atoi(argv[1]); /*string to int*/
    b = a * a;
    if (a == 2) [[likely]] { // likely
        a++;
        b++;
    } else {
        a--;
        b--;
    }
    return a + b;
}
```

使用支持C++20标准的编译器（如 g++-11 ）编译：

```
$ g++-11 -O2 branch_pred.cpp -o branch_pred
```

尝试执行该程序：

```
$ ./branch_pred 5
$ echo $?
28
```

echo \$? 是什么意思？

以往编程中，我们约定 `return 0`；表示正常退出，非零值都是有问题的！这些值其实表示当前程序在执行结束后的退出状态 (exit status)，称为退出码 (exit code)。

与HTTP状态码 (HTTP Status Code) (比如常说的404了) 一样，通过观察退出码，用户可以判断程序的执行是否正常，以及产生的异常类型。在早期高级语言没有提供异常机制时，这能帮助大家锁定问题所在。通过 `echo $?` 可以查看上一次执行的退出码。

一个常见的使用场景是在环境配置脚本中，检查当前虚拟环境下Python的某个package是否导入成功，若不成功则进行安装：

```
#!/bin/bash
# ./exec_app.sh

PYPI_MIRROR=https://pypi.tuna.tsinghua.edu.cn/simple

# * 启用已存在的虚拟环境（若无则先创建）
if [[ ! -d ".venv" ]]; then
    python -m venv .venv
fi
source .venv/bin/activate

# ^ 检查 numpy 是否成功导入，若否则重新安装
python -c "import numpy as np"
if [[ $? != 0 ]]; then
    pip install --force-reinstall --no-cache-dir -i ${PYPI_MIRROR} numpy
fi
```

PS：如果你使用 Oh My Zsh（并搭配合适的主题，如PowerLevel10k），zsh 会显示上一次执行的退出码，比 bash 直观。

观察 `branch_pred.cpp`，我们最终通过 `main()` 的 `return` 语句将最终结果作为退出状态值返回，通过 `echo $?` 来查看结果。

对程序进行反汇编：

```
$ objdump -d test > test.asm
```

该命令会将反汇编的结果从标准输出流 `stdout` 重定向输出 到 `test.asm` 文件中保存。你可以使用编辑器（如VSCode）打开文件，或者用 `cat` 命令：

```
$ cat test.asm
```

试判断下面两段反汇编代码分别对应 `[[likely]]` 还是 `[[unlikely]]`（需要加上 `[[ ]]`），并说明理由。（4分）

`[[likely]]` 和 `[[unlikely]]` 会把更可能执行的代码顺序排放在分支之后。

反汇编代码	对应的指示宏	理由
<pre>0000000000001060 <main>: 1060: f3 0f 1e fa endbr64 1064: 48 83 ec 08 sub \$0x8,%rsp 1068: 48 8b 7e 08 mov 0x8(%rsi),%rdi 106c: ba 0a 00 00 00 mov \$0xa,%edx 1071: 31 f6 xor %esi,%esi 1073: e8 d8 ff ff ff callq 1050 <strtol@plt> 1078: 89 c2 mov %eax,%edx 107a: 0f af d0 imul %eax,%edx 107d: 83 f8 02 cmp \$0x2,%eax 1080: 74 0d je 108f <main+0x2f> 1082: 83 e8 01 sub \$0x1,%eax 1085: 83 ea 01 sub \$0x1,%edx 1088: 01 d0 add %edx,%eax 108a: 48 83 c4 08 add \$0x8,%rsp 108e: c3 retq 108f: ba 05 00 00 00 mov \$0x5,%edx 1094: b8 03 00 00 00 mov \$0x3,%eax 1099: eb ed jmp 1088 <main+0x28> 109b: 0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)</pre>	<code>[[unlikely]]</code>	可以发现 <code>0x1080</code> 处的指令判断 如果 <code>a == 2</code> ，那么 <code>jmp</code> 到 <code>0x108f</code> ，如果不相等，执行后续的减法操作。减法是更可能被执行的一个分支，所以是 <code>[[unlikely]]</code>
<pre>0000000000001060 <main>: 1060: f3 0f 1e fa endbr64 1064: 48 83 ec 08 sub \$0x8,%rsp 1068: 48 8b 7e 08 mov 0x8(%rsi),%rdi 106c: ba 0a 00 00 00 mov \$0xa,%edx 1071: 31 f6 xor %esi,%esi 1073: e8 d8 ff ff ff callq 1050 <strtol@plt> 1078: 41 b8 08 00 00 00 mov \$0x8,%r8d 107e: 83 f8 02 cmp \$0x2,%eax 1081: 75 08 jne 108b <main+0x2b> 1083: 44 89 c0 mov %r8d,%eax 1086: 48 83 c4 08 add \$0x8,%rsp 108a: c3 retq 108b: 89 c2 mov %eax,%edx 108d: 0f af d0 imul %eax,%edx 1090: 44 8d 44 02 fe lea -0x2(%rdx,%rax,1),%r8d 1095: eb ec jmp 1083 <main+0x23> 1097: 66 0f 1f 84 00 00 00 nopw 0x0(%rax,%rax,1) 109e: 00 00</pre>	<code>[[likely]]</code>	在分支 <code>0x1081</code> 处判断是否 <code>a !=2</code> ，分支后的指令将直接返回 <code>0x8</code> 也就是 <code>a == 2</code> 的结果， <code>a == 2</code> 是更可能发生的分支，所以是 <code>[[likely]]</code>

试说明 `[[likely]]` 和 `[[unlikely]]` 为什么在一定程度上能在优化程序在流水线处理器上的执行？（1分）

只要说明 "实际跳转会停止已进入流水线的指令的处理，造成性能损失，（默认大概率不会跳转时）因此尽量避免跳转" 即可得分。

以下解释为什么处理器执行时更倾向于 "默认预测分支不会发生跳转"，属于课外内容，供大家参考。

因为在存储层次结构中，靠近 CPU 的 Cache（后续课程会讲解，对应CSAPP 第）是以 cacheline 为单位（通常为 64 Bytes）搬运数据的，一个 cacheline中包含多条指令，将更可能跳转的指令和分支指令排在一起，可以把该跳转方向的指令和分支指令一起搬运进Cache，从而提高指令Cache 的命中率。

另外现代 CPU 中集成了Next-Line等指令预取器，会将更靠近分支的指令提前预取进 Cache。

作业三

本次作业一共分为4道大题，**总分30分**：

- Q1 是综合题：考察对课上内容的掌握；
- Q2 是分析题：具体来说你需要分析汇编代码的执行过程及其用途；
- Q3 为实践题：通过实际的X86汇编编程，让大家能更好的理解X86汇编；
- Q4 为多选题：与课调委沟通后，我们希望收集大家对作业的反馈，一同改进课程质量。

⚠ 打包提交本次作业

具体来说你需要将 书面作业的答案 + 经 check.py 测试的 fib.s 两个文件**打包上传**到网络学堂。

假如你的学号为 2022011451，则可以通过 tar 或 zip 命令打包：

```
$ ls . # 显示当前作业目录
fib.s 作业三_答案.pdf/.docx/.md
$ tar -czvf 2022011451.tar.gz *
$ zip 2022011451.zip *
```

将打包好的 2022011451.tar.gz 或 2022011451.zip 上传到网络学堂。

Q1. 综合 (7分)

Q1.1. 填充 (5分)

编程解决猴子吃桃问题：每天吃一半再多吃一个，第十天想吃时候只剩一个，问总共有多少。
该程序的 C 语言程序如下：

```
int eat_peaches(int i) {
    if (i == 10) return 1;
    else return (eat_peaches(i + 1) + 1) * 2;
}
```

填充汇编代码（Linux X86-64）中缺失的内容：

```
eat_peaches:
.LFB0:
    cmpl     __空格1__, %edi
    __空格2__ .L8
    movl     $1, %eax
    ret
.L8:
    __空格3__ $8, %rsp
    addl     $1, %edi
    call     eat_peaches
    leal     __空格4__(%rax,%rax), %eax
    addq     $8, __空格5__
    ret
```

本题每一空格1分。

.LFB0 部分判断 `i == 10`，如果一致则执行 `if` 分支，直接返回 1，否则跳转至 .L8 执行 `else` 分支；因此可以确定 __空格(1)__ 为 `$10`，__空格(2)__ 为 `jne`

注意到 `eat_peaches` 其实是一个递归函数，因此在 .L8 中需要分配与回收栈帧，结合上下文可知 __空格(3)__ 为 `subq`，__空格(5)__ 为 `%rsp`

实际的返回值为 `eat_peaches(i + 1)` 的返回值加 1 再乘 2，即 $2 * eat_peaches(i + 1) + 2$ ，在 `leal` 指令中写法应为 `2(%rax, %rax)`，即 `%rax + %rax + 2`，因此 __空格(4)__ 为 2

空格	值
__空格(1)__	\$10
__空格(2)__	jne
__空格(3)__	subq
__空格(4)__	2
__空格(5)__	%rsp

Q1.2. bfloat16 (2分)

bfloat16 是由Google提出的一种半精度浮点数，`exp` 域为8位，`frac` 域为7位，`sign` 域为1位。除了位宽度差别外，bfloat16 的其它规格符合IEEE 754标准。

现定义一个C语言 `union`（联合体）数据类型，如下所示：

```
union {
    bfloat16 f;
    unsigned short s;
}
```

现在为 `f` 赋予 bfloat16 所能表示的最接近于 1 且大于 1 的数，在X86（小端序）机器上运行时，`s` 的十六进制值为多少？

符合IEEE 754标准，即 bfloat16 严格遵循 $(-1)^s \times M \times 2^E$ 的标准，且有 $E = \text{exp} - \text{bias} \Rightarrow \text{exp} = E + \text{bias}$



为什么需要bias?

bias的用意在于将带符号数 E 的表示范围"平移"至无符号`exp`的表示范围中，从而避免产生歧义

参见课件《2.1 整数》第13页、《2.2 浮点数》第9页

接近于1且大于1，说明这个数：

- 是一个规格化浮点数 $\Rightarrow$ 尾数省略前导 1、 $\text{bias} = 2^{E-1} - 1$
- 正数，符号域 $s = 0$ ，即 `s = 0b`
- 大于 1，说明 $M = 1.0000001$ ，省略前导 1，即 `frac = 0000001b`
- $E = 0$ ，则 $\text{bias} = 2^{8-1} - 1 = 127$ ，于是 `exp = 0b + 01111111b = 01111111b`

则二进制表示为 `0 01111111 0000001b = 0x3f81`

s = 0x3F81

Q2 Getter & Setter (10分)

过程调用以及返回的顺序在一般情况下都是"过程返回的顺序恰好与调用顺序相反"，但是我们可以利用汇编以及对运行栈的理解来编写汇编过程打破这一惯例。最典型的应用为有栈协程切换。

有如下汇编代码（X86-32 架构），其中 GET 过程唯一的输入参数是一个用于存储当前处理器以及栈信息的内存块地址（假设该内存块的空间足够大），而 SET 过程则用于恢复被 GET 过程所保存的处理器及栈信息，其唯一的输入参数也是该内存块地址。

GET:	SET:
movl 4(%esp), %eax #(A)	movl 4(%esp), %eax
...	...
movl %edi, 20(%eax)	movl 20(%eax), %edi
movl %esi, 24(%eax)	movl 24(%eax), %esi
movl %ebp, 28(%eax)	movl 28(%eax), %ebp
movl %ebx, 36(%eax)	movl 36(%eax), %ebx
movl %edx, 40(%eax)	movl 40(%eax), %edx
movl %ecx, 44(%eax)	movl 44(%eax), %ecx
movl \$1, 48(%eax)	movl ____(%eax), %esp #(D)
movl (%esp), %ecx #(B)	pushl 60(%eax) #(E)
movl %ecx, 60(%eax)	movl 48(%eax), %eax
leal 4(%esp), %ecx #(C)	ret
movl %ecx, 72(%eax)	
movl 44(%eax), %ecx	
movl \$0, %eax	
ret	

在理解代码的基础上，回答下列问题：

1. SET 过程的返回地址是什么，其返回值是多少？（
2. 代码段中的 (A) 指令执行后，eax 中存放的是什么？
 - (B) 指令执行后，ecx 中存放的是什么？
 - (C) 指令的作用是什么？
 - (E) 指令的作用是什么？并将 (D) 指令补充完整。

本题评分标准为 **基础分3分 + 正确性 7分**：每一小问1分，至少可得3分。

1. SET 过程的返回地址是 被 GET 过程保存的处理器执行的下一条指令的地址；返回值为 1
2.
 - (A) 指令执行后，eax 存放的是 保存处理器和栈信息的内存块的地址
 - (B) 指令执行后，ecx 存放的是 返回地址（即 调用 GET 过程后要执行的下一条指令所处的地址）
 - (C) 指令 记录 esp 栈顶指针，用于在 (D) 在将返回地址压入栈中
 - (D) 指令完整应为 movl 72(%eax), %esp
 - (E) 指令 将返回地址压入栈

Q3 Fibonnaci求解 (8分)

Fibonacci数列是一个非常经典的数列，其定义如下：

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0, F(1) = 1$$

为了巩固对X86-64汇编的掌握，我们提供了一个基于X86-64和Linux系统调用、**求解Fibonacci数列中第 n 项**的手写汇编代码，但是里面有一些bug，需要交给你来修复。（预期用时：< 8小时）

详见 fib.s 的 TODO：以及对应的 hint 。具体来说，你需要：

1. 手动实现 print_int 函数缺失的 itoa_digits （4分）

```
.type print_int, @function
print_int:
    # number to be printed in %rdi
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, %rax

    # 先令换行符入栈，最后输出时换行
    movq $0xa, %rsi
    pushq %rsi
    movq $1, %rbx    # 记录当前待输出的字符数量，因为含'\n'所以初始为1

    itoa_digits:
        # TODO: 将数字转换为字符串，并逆序保存在栈上，使得打印时次序正确
        # hint0: 从低位开始向高位处理，低位先入栈，高位先出栈
        # hint1: 使用cqto指令从32位扩展至64位，再用idivq指令获取 商(%rax) 和 余数(%rdx)
        # hint2: 通过加上 $DIGIT_0 将余数转换为字符，并压入栈中保存，注意压入的是 X86-64寄存器
        # hint3: 更新待输出的字符总数(%rbx)中，这将作为print_digits的结束依据
```

提示：可以先注释掉 call fib ，通过调用 print_int 函数，观察是否能正确输出 scan_int 的结果。

2. 修正 fib.s 中的 fib 函数（4分）

```
.type fib, @function
fib:
    # TODO: 修正 fib函数，使得能够正确计算fibonacci数列
    # hint0: 可以使用GDB打断点，进行指令粒度的调试
    # hint1: 检查栈帧的分配与回收是否正确
    # hint2: 检查函数调用前后寄存器是否被正确保存与恢复
    # hint3: 检查函数调用时的参数是否正确传递

    # n is in %rdi
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, %rax
    cmpq $2, %rdi
    jl fib_end
    decq %rdi
    call fib
    movq %rax, %r8
    decq %rdi
    call fib
    addq %r8, %rax
fib_end:
    movq %rsp, %rbp
    popq %rbp
    ret
```

提示：你可以手写一个C程序，生成汇编代码后交叉对比，找出 fib 函数中的bug；善用 pushq / popq 指令能优化栈的使用体验。

使得 fib.s 最终能通过编译并正确执行。

本题给分方式为 **参与分 6分 + 正确分 2分**：即完成书面作业其它题目（不论正确性），却未能通过 check.py 随机化测试，仍可获得6分。

Q3.1 itoa_digits

print_int 的本质其实就是实现 itoa() ，然后逐一遍历输出字符，对应的C++伪代码如下：

```

void print_int(int number) {
    std::stack<char> stk;
    stk.push('\n');

    int quotient, remainder; // 商、余数
    quotient = number;
    while (quotient) { // itoa_digits
        remainder = quotient % 10;
        stk.push((char)('0' + remainder));
        quotient /= 10;
    }

    while (!stk.empty()) { // print_digits
        char poll = stk.top();
        putchar(poll);
        stk.pop();
    }
}

```

我们需要做的就是将上面 itoa_digits 循环"翻译"为X86-64汇编代码。
 首先要做的，就是搞清楚除法与取模操作分别对应什么汇编指令；按照作业提示与 hint1，可以确定基本的指令为 cqto 和 divq（或 idivq）

指令	效果（= 表示赋值）	描述
cqto	R[%rdx]:R[%rax] = 符号扩展(R[%rax])	%rax 中的数符号扩展至128位，高64位保存在 %rdx 中，低64位保存在 %rax
divq S	R[%rdx] = R[%rdx]:R[%rax] % S R[%rax] = R[%rdx]:R[%rax] / S	无符号除法：除数为 S 中的值，被除数为经 cqto 扩展后128位，余数保存在 %rdx 中，商保存在 %rax 中
idivq S	R[%rdx] = R[%rdx]:R[%rax] % S R[%rax] = R[%rdx]:R[%rax] / S	带符号除法，同上

由于 $F(n) \geq 0$ ，因此这里使用 divq 或 idivq 都可以

执行 divq 前，必须先执行 cqto，否则每次迭代前必须手动置 %rdx 为 0，否则会进入死循环，引起SEGFault（可以通过 gdb 指令级调试）

获得余数后，我们通过（addq 指令）加上 '0'（保存在常量 \$DIGIT_0 中）将其转换为ASCII字符，并（调用 pushq 指令）压入运行栈中；重复上述过程，直到商为 0，即 R[%rdx] == R[%rax] == 0

注意到在伪代码中，我们可以通过查询 stk.size() != 0（即 !stk.empty()）来判断 print_digits 是否结束，可实际的运行栈并没有这样的接口，因此我们需要维护一个变量，记录需要打印的字符数量，结合上下文（的注释）可知变量保存在 %rbx 中

参考实现如下：

```

itoa_digits:
    # TODO: 将数字转换为字符串，并逆序保存在栈上，使得打印时次序正确
    # hint0: 从低位开始向高位处理，低位先入栈，高位先出栈
    # hint1: 使用cqto指令从32位扩展至64位，再用divq指令获取 商（%rax）和 余数（%rdx）
    # hint2: 通过加上 $DIGIT_0 将余数转换为字符，并压入栈中保存，注意压入的是 X86-64寄存器
    # hint3: 更新待输出的字符总数（%rbx）中，这将作为print_digits循环结束的依据
    movq $10, %rcx
    cqto
    divq %rcx
    # quotient is in %rax, remainder is in %rdx
    addq $DIGIT_0, %rdx
    pushq %rdx
    incq %rbx
    cmpq $0, %rax
    jnz itoa_digits

```

Q3.2 fib

对于 fib 函数，不难发现其实就是（最朴素的）递归实现，很容易写出标准的C程序：

```
int fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}
```

将它转为X86-64汇编代码：

```
fib:
    # n in %rdi
    pushq %rbp                # (1)
    pushq %rbx                # (3) 进入子节点后，保存父节点的n，并在返回时恢复
    subq $8, %rsp             # (1) 分配栈帧
    movl %edi, %ebx           # (2) 拷贝n到别处，递归调用时直接修改原本的n（在%rdi中）
    cmpl $1, %edi
    jle .L3                   # if (n <= 1), 亦即 if (n < 2)
    leal -1(%rdi), %edi        # 计算 fib(n - 1)
    call fib
    movl %eax, %ebp           # 临时保存 fib(n - 1) 的结果
    leal -2(%rbx), %edi        # 计算 fib(n - 2)
    call fib
    addl %ebp, %eax           # 计算 fib(n - 1) + fib(n - 2)
.L1:
    addq $8, %rsp             # (4) 回收栈帧
    popq %rbx                 # (5) 恢复父节点的n
    popq %rbp                 # (4)
    ret
.L3:
    movl %edi, %eax
    jmp .L1
```

需要注意的几处细节：

- 在每个过程实例中，寄存器的值是会变化的，因此需要把递归前后会复用的寄存器值保存到一个 仅属于当前实例的运行空间上，即当前实例的**栈帧**上
- 需要将 %rdi 保存到别处，这里是保存到 %rbx 寄存器上，并且在进入 子递归实例 后，保存在运行栈上，待返回前恢复（思考一下二叉树的非递归后序遍历是如何保存变量的）
- 递归调用 fib(n - 1) 和 fib(n - 2) 后，计算结果必须保存到运行栈中
- 判断何时及如何高效地执行 if 分支

比对生成的汇编代码，找出原本的 fib 实现中出现的问题（共有4处错误）：

```
.type fib, @function
fib:
    # TODO: 修正fib函数，使得能够正确计算fibonacci数列
    # hint0: 可以使用GDB打断点，进行指令粒度的调试
    # hint1: 检查栈帧的分配与回收是否正确
    # hint2: 检查函数调用前后寄存器是否被正确保存与恢复
    # hint3: 检查函数调用时的参数是否正确传递

    # n is in %rdi
    pushq %rbp                # 分配栈帧
    movq %rsp, %rbp           # 分配栈帧
    movq %rdi, %rax            # 对应于 if (n < 2), 直接返回 n
    cmpq $2, %rdi
    jl fib_end
    decq %rdi                  # FIXME: 递归后%rdi不再被恢复了，需要提前保存在栈上
    call fib
    movq %rax, %r8             # FIXME: 如果%rdi保存在栈上但没恢复，那么这里的%rdi是某个后代节点的值
    decq %rdi                  # FIXME: 如果%rdi保存在栈上但没恢复，那么这里的%rdi是某个后代节点的值
    call fib
    addq %r8, %rax              # FIXME: 递归后子节点及其后代很有可能修改了%r8，那么这个%r8是谁的？
fib_end:
    movq %rsp, %rbp           # FIXME: 回收栈帧
    popq %rbp
    ret
```

可以看见，这里 fib 最大的问题在于寄存器的保存与恢复；一种 纯手写、极不优雅的 实现方式如下：

```
.type fib, @function
fib:
    # TODO: 修正fib函数, 使得能够正确计算fibonacci数列
    # hint0: 可以使用GDB打断点, 进行指令粒度的调试
    # hint1: 检查栈帧的分配与回收是否正确
    # hint2: 检查函数调用前后寄存器是否被正确保存与恢复
    # hint3: 检查函数调用时的参数是否正确传递

    # n is in %rdi
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, %rax
    cmpq $2, %rdi
    jl fib_end
    pushq %rdi
    decq %rdi
    call fib
    movq %rax, %r8
    popq %rdi
    pushq %r8
    subq $2, %rdi
    call fib
    popq %r8
    addq %r8, %rax
fib_end:
    movq %rbp, %rsp
    popq %rbp
    ret
```

作业四

本次作业为《计算机系统概论》课的期中复习，其中包含5道书面题，每题8分。

Q1 溢出 (8分)

有如下的C程序：

```
#include <stdio.h>
int overflow(void);
int one = 1;
int overflow() {
    char buf[4];
    int val, i = 0;
    while (scanf("%x", &val) != EOF) buf[i++] = (char)val;
    return 15213;
}

int main() {
    int val = overflow();
    val += one;
    if (val != 15213)
        printf("Boom!\n");
    else
        printf("????\n");
    exit(0);
}
```



scanf 回顾

`scanf("%x", &val)`；函数从标准输入读取由空格分割的代表一个十六进制整数的字符或字符序列，并将该字符或字符序列转换为一个32位 `int`，将转换结果赋给 `val`。

`scanf` 返回 1 表示转换成功，返回 `EOF`（即 `-1`）则表示 `stdin` 已经没有更多的输入序列了（通常输入 `Ctrl+D` 表示 `EOF`）。

例如，在输入字符串 `"0 a ff"` 上调用 `scanf` 四次将有以下结果：

1. 1st call: `val=0x0` and `scanf` returns `1` .
2. 2nd call: `val=0xa` and `scanf` returns `1` .
3. 3rd call: `val=0xff` and `scanf` returns `1` .
4. 4th call: `val=??` and `scanf` returns `EOF` .

对应的X86-64架构下的（反）汇编代码：

```

00000000004005d6 <overflow>:
4005d6: 55                push %rbp
4005d7: 48 89 e5          mov %rsp, %rbp
4005da: 48 83 ec 28       sub $0x28, %rsp
4005db: 53                push %rbx
4005df: bb 00 00 00 00    mov $0x0, %ebx
4005e4: eb 0d            jmp 4005f3 <overflow+0x1d>
4005e6: 48 63 c3          movslq %ebx, %rax
4005e9: 8b 55 dc          mov -0x8(%rbp), %edx    # edx = val
4005ec: 88 54 05 e0       mov %dl, -0x4(%rbp, %rax, 1) #buf
4005f0: 8d 5b 01          lea 0x1(%rbx), %ebx    # i++
4005f3: 48 8d 75 dc       lea -0x8(%rbp), %rsi
4005f7: bf d4 06 40 00    mov $0x4006d4, %edi    # "%x"地址是 0x4006d4
4005fc: b8 00 00 00 00    mov $0x0, %eax
400601: e8 ba fe ff ff    callq 4004c0 <scanf>
400606: 83 f8 ff         cmp $0xffffffff, %eax  # EOF 值为-1
400609: 75 db            jne 4005e6 <overflow+0x10>
40060b: b8 6d 3b 00 00    mov $0x3b6d, %eax     # 0x3b6d=15213
400610: 5b                pop %rbx
400614: 48 83 c4 28       add $0x28, %rsp
400615: 5d                pop %rbp
400616: c3                retq

0000000000400617 <main>:
400617: 55                push %rbp
400618: 48 89 e5          mov %rsp, %rbp
40061b: e8 b6 ff ff ff    callq 4005d6 <overflow>
400620: 03 05 22 0a 20 00 add 0x200a22(%rip), %eax # val += one
400626: 3d 6d 3b 00 00    cmp $0x3b6d, %eax
40062b: 74 0c            je 400639 <main+0x22>
40062d: bf d7 06 40 00    mov $0x4006d7, %edi
400632: e8 69 fe ff ff    callq 4004a0 <puts>    # printf("Boom!\n")
400637: eb 0a            jmp 400643 <main+0x2c>
400639: bf dd 06 40 00    mov $0x4006dd, %edi
40063e: e8 5d fe ff ff    callq 4004a0 <puts>    # printf("????\n")
400643: bf 00 00 00 00    mov $0x0, %edi
400648: e8 43 fe ff ff    callq 400490 <exit>    # exit 不使用 rbp

```

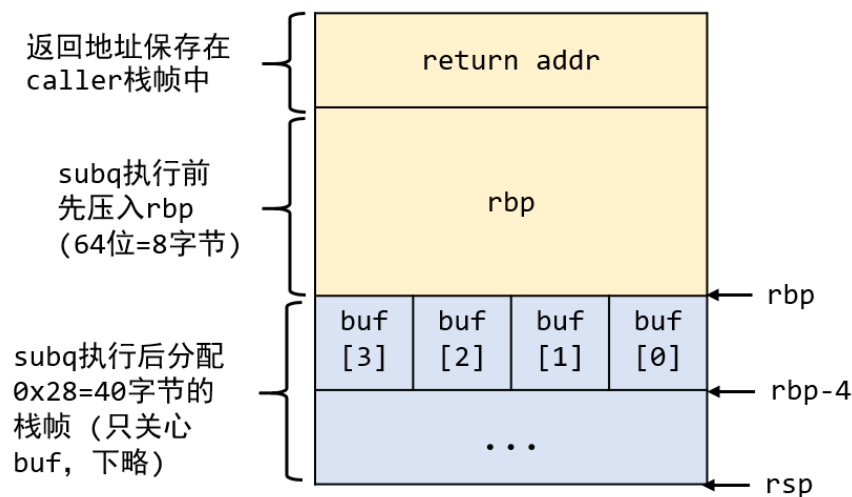
✓ （反）汇编代码勘误

（反）汇编代码中地址 4005e9 处的字节码有误，仅需观察反汇编的汇编代码即可。感谢 郭士尧-信计21、董业恺-计21、张宁远-计24/新雅22 的指正。

1. 请参照 buf[0]、buf[1] 的地址表示方式表示表格内其余对象的地址（4分）。

buf 是字符数组，因此 buf[2] 和 buf[3] 的地址显而易见，关键在于栈上的 %rbp（overflow() 内压入栈的和返回地址的位置）。

关键点在于 buf 的首地址，从 4005ec 处的（反）汇编指令可知 buf = %rbp - 0x4（初始时 %rax 被置为 0），而 %rbp 最后一次更新是在压入 %rbp、为 overflow() 分配栈帧前，其值为该时刻栈顶 %rsp 的值，因此可以还原出栈上内容（每一方格表示一个字节）：



得到这个关系后就能还原出其余对象的地址：

Stack object	Address of stack object
return address	<code>&buf[0] + 12</code>
old %rbp	<code>&buf[0] + 4</code>
<code>buf[3]</code>	<code>&buf[0] + 3</code>
<code>buf[2]</code>	<code>&buf[0] + 2</code>
<code>buf[1]</code>	<code>&buf[0] + 1</code>
<code>buf[0]</code>	<code>&buf[0] + 0</code>

做题贴士

做题时不要一上来就分析汇编代码：很费时间（尤其是在期末考试中，**题量很大**）；建议**先搞懂题目要的是什么**，或者说**希望从汇编代码中得到什么信息来解题**，然后逐行读，屏蔽一切不需要的细节。

2. 为了让这个程序输出 "????", 你需要在命令行输入什么字符串？以下每个下划线都是一个 1 位或 2 位的十六进制数字，下划线之间是空格（4分）。

由上一小题可知返回地址相对于 `buf[0]` 的位置，为了避免触发 `val += one;` 导致 `if()` 分支被执行，我们可以通过 `scanf()` 造成 `buf` 溢出，修改返回地址 `400620`（下一条待执行的指令的地址）为想要的目标地址，此时的方案有：

- 绕过 `val += one;`，直接跳转到 `400626` 处开始执行：

任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 26

- 直接跳转到 `400639` 处执行 `else` 分支（**必须先载入 "????\n" 才能 printf()**）：

任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 39

Q2 结构体（8分，1空2分）

有如下对应的C代码与对应的X86-64汇编代码：

```

struct matrix_entry {
    char a;
    char b;
    double d;
    short c;
};

struct matrix_entry matrix[5][__空格(1)__);

int return_entry(int i, int j) { return matrix[i][j].c; }

```

```

return_entry:
    movslq    %esi, %rsi
    movslq    %edi, %rdi
    leaq      (%rsi,%rsi,2), %rax
    leaq      0(,%rax,8), %rdx
    leaq      (%rdi,%rdi,4), %rax
    leaq      (%rdi,%rax,4), %rcx
    __空格(2)__ 0(,%rcx,8), %rax
    movswl    matrix+ __空格(3)__(%rdx,%rax), %eax
    ret

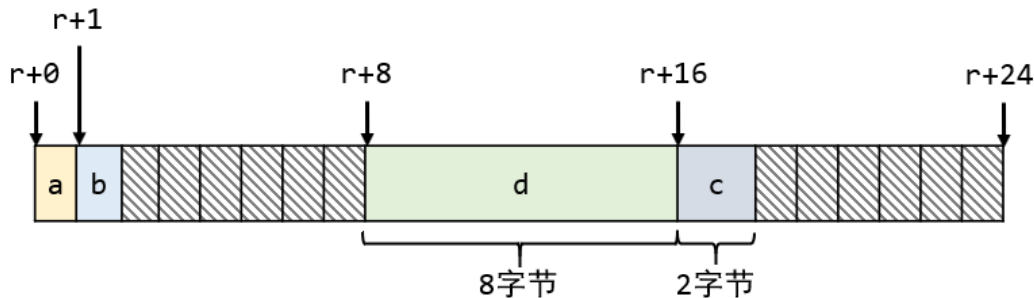
    .comm     matrix, __空格(4)__

```

.comm后的第二个参数表示matrix占据空间的大小，以字节为单位

请对照着填上代码中缺失的部分（数字请用十进制表示）。

注意到是在X86-64上（即机器字长为64位），因此内存按 $64/8 = 8$ 字节对齐，此时 struct matrix_entry 的内存布局如下（每一小格为1个字节，r 代表 matrix_entry 的首地址位置）：



matrix_entry 的总大小为 24 字节

可以看到 matrix_entry.c 相对首地址的偏移为 16 字节，那么 return_entry() 返回的结果位于 matrix[i][j] + 16 处，通过 movswl 加载到 %eax 中，因此 空格(3) 的值为 16。

参数 i 和 j 分别保存在 %edi 和 %esi 中，我们需要追踪它们所对应的寄存器值：

- 参数 j :
 - i. %esi 扩展至 %rsi ;
 - ii. $\%rax = \%rsi + 2 * \%rsi = 3 * \%rsi$;
 - iii. $\%rdx = 8 * \%rax = 8 * 3 * \%rsi = 24 * rsi$, 而 24 即 matrix_entry 的总大小。
- 参数 i :
 - i. %edi 扩展至 %rdi ;
 - ii. $\%rax = \%rdi + 4 * \%rdi = 5 * \%rdi$;
 - iii. $\%rcx = \%rdi + 4 * \%rax = 21 * \%rdi$;

iv. `%rax = 8 * %rax = 168 * %rdi = (7 * 24) * %rdi`，由上知一行共 7 个元素，因此 空格(1) 的值为 7；
因为是计算，所以 空格(2) 显然为 `leaq`。

可以确定出 `matrix[5][7]`，则总大小为 $5 \times 7 \times 24 = 840$ 字节，即 空格(4) 的答案。

空格	值
(1)	7
(2)	<code>leaq</code>
(3)	16
(4)	840

Q3 跳转表（8分）

课堂上我们结合 `switch` 语句示例讲解了跳转表（jump table）的应用。课上的例子采用的是绝对地址定位的方式，即跳转表中的每一项存放的是各个代码块的绝对地址。后面我们学习了“共享库中的全局变量寻址”，知道共享库被不同进程装载的时候，其绝对地址是不一样的，这就给绝对地址定位的方式带来了难度。一种解决方式是“相对定位”方式，其依据的事实是：不管对象文件被装载到进程的哪个地址，代码段中的任一给定指令与数据段（包括只读数据段）中的任一给定位置之间的“距离”是一个常量。据此编译器生成了 `switch` 示例的与绝对地址无关（Position Independent Code）代码（X86-64架构）如下（右侧是对应的C函数），其中 `.L4` 标识了该跳转表。

```
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch (x) {
        case 1:
            w = y * z;
            break;
        case 2:
            w = y / z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

```

switch_eg:
    cmpq    $6, %rdi
    movq    %rdx, %rcx
    ja      .L8                                # __指令1__
    leaq    __空格1__(%rip), __空格2__
    movslq  (__空格3__, __空格4__, 4), %rdi
    __空格5__ %r8, %rdi
    jmp     *__rdi                             # 以%rdi为目标地址直接跳过去

    .section      .rodata                      # 只读数据段
.L4:
    .long        .L8-.L4
    .long        .L3-.L4
    .long        .L5-.L4
    .long        .L9-.L4
    .long        .L8-.L4
    .long        .L7-.L4
    .long        .L7-.L4
    .text
.L9:
    movl        __空格6__, %eax
    addq        %rcx, %rax
    ret
.L5:
    movq        %rsi, %rax
    cqto
    idivq       %rcx
    addq        %rcx, %rax
    ret
.L3:
    movq        %rdx, %rax
    imulq       %rsi, %rax
    ret
.L7:
    movl        $1, %eax
    subq        %rdx, %rax
    ret
.L8:
    movl        $2, %eax
    ret

```

请根据上述事实以及C函数语义，回答以下问题：

这道题的另一种考察形式请参考 作业二Q1.3：画出分支树还原 switch 语句。

1. 请问为何 __指令1__ 处用的是 ja 指令来进行带符号数的条件判断？（2分）

ja 用于无符号数比较；在这段代码里，如果将小于0的带符号数当成无符号数（绝对值较大）处理，比较后执行的代码段是不变的，均为 default。

2. 填写空格处缺失的值以补齐指令（6分，1空1分）：

不妨从后往前推理：

- .L9 逻辑比较简单：z 存在寄存器 %rdx 中，一开始令 %rcx = %rdx；，先对 %rax 赋值，然后 %rax += %rcx；
对应C代码中的 case 3：w = 1；w += z；return w；由此推理出 空格(6) 为 \$1；
- 由注释可知最后以 %rdi 为目标地址跳过去，可 %rdi 一开始存储参数 x，什么时候变成 switch 中对应分支的地址？
注意看在这之前涉及了 %r8 和 %rdi 的数据操作，那么 %r8 什么时候赋值的呢？由 AT&T 风格 可知只能是 空格(2)；

- 空格(2) 对应的汇编代码是典型的RIP相对寻址（见课件《4 汇编与C语言-3》P52），因此 空格(1) 只需填入跳转表的标识 `.L4` 即可。
- 综上，`%r8` 存储的是跳转表的地址；
- 仔细阅读题目：**PIC代码采用"相对定位"**，即跳转表中的每一项不再是对应分支的实际地址，而是相对于跳转表基地址的偏移。

访问对应的分支的C语言伪代码如下：

```
r8 = jump_table_addr;
jump_table_offset = r8[x]; // x是索引，实际地址偏移还需要乘以 sizeof(long) = 4
target_branch_addr = jump_table_addr + jump_table_offset;
rdi = target_branch_addr;
```

求出 `jump_table_offset` 对应于 `movslq` 操作，即 `rdi = r8[x] = *(r8 + x * 4)`，

故 空格(3) 和 空格(4) 分别为 `%r8` 和 `%rdi`，且由伪代码可知 空格(5) 的指令显然为 `addq`。

最终答案为：

空格	值
(1)	<code>.L4</code>
(2)	<code>%r8</code>
(3)	<code>%r8</code>
(4)	<code>%rdi</code>
(5)	<code>addq</code>
(6)	<code>\$1</code>

Q4 echo (8分, 1空1分)

下面是一段 C 代码以及对应的 X86-64 汇编：

```
void echo() {
    char buf[8];
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushq    %rbx
    xorl     %eax, %eax
    subq     $16, %rsp
    leaq     8(%rsp), %rbx
    movq     %rbx, %rdi
    call     gets
    movq     %rbx, %rdi
    call     puts
    addq     $16, %rsp
    popq     %rbx
    ret
```

填充下面空格缺失的值：

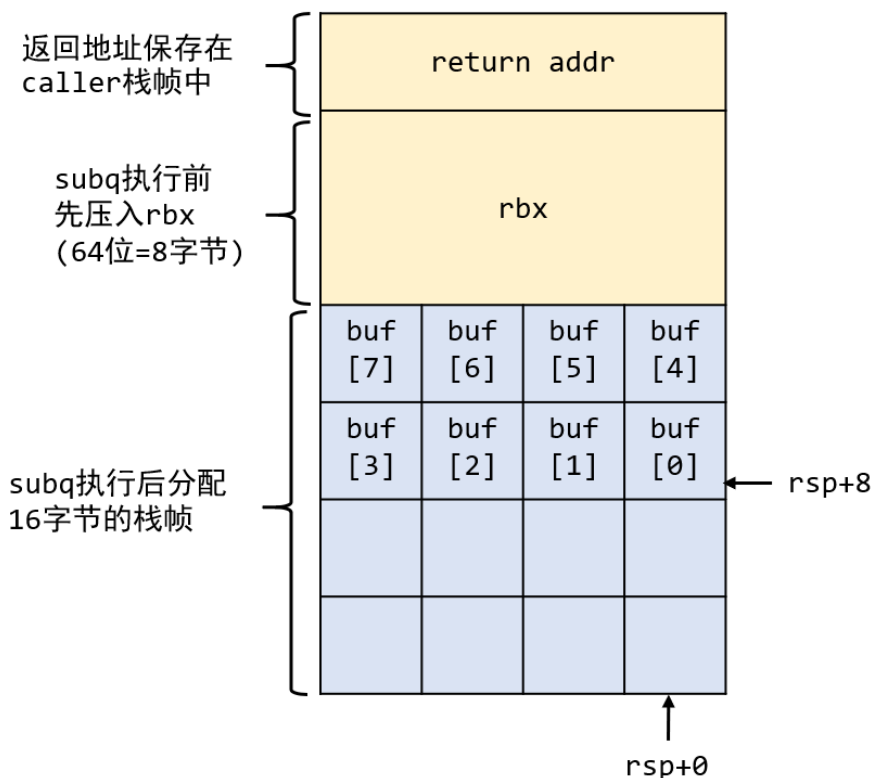
1. `gets` 的参数通过寄存器 空格(1) 传递;
2. 寄存器 `%rbx` 属于 空格(2) (caller saved/callee saved) 寄存器;
3. 以 `subq` 指令后的 `%rsp` 计算, `buf` 数组的基地址是 `%rsp +` 空格(3) , `%rbx` 寄存器保存在 `%rsp +` 空格(4) ;
4. 上面的代码有缓冲区溢出的漏洞, 已知 `gets` 会从标准输入读入一行字符串, 把字符串保存在其第一个参数指向的缓冲区, 最后填入 `NULL` (亦即 `'\0'`) 作为结尾。请计算, 为了满足以下的要求, 输入的字符串长度 (不计入 `NULL`) 需要满足的要求, 以区间表示:
 - 该字符串输入到上面的程序, `buf` 数组被更新但是缓冲区没有溢出: 长度范围是 `[0,` 空格(5) `]`
 - 该字符串输入到上面的程序, 保存在栈上的 `%rbx` 寄存器被更新, 但是栈上的返回地址没有被更新: 长度范围是 `[` 空格(6) `,` 空格(7) `]`
 - 该字符串输入到上面的程序, 保存在栈上的 `%rbx` 寄存器和返回地址被更新: 长度范围是 `[` 空格(8) `, +inf)`

1. `gets()` 只有一个参数, 按照X86-64参数寄存器约定, 第一个参数使用 `%rdi` 传递;
2. `%rbx` 属于callee saved寄存器 (见课件《4-C语言-3》第39页);
3. • 先讨论 `buf` 的基地址: 不妨先回忆 `gets()` 和 `puts()` 的原型:

```
char *gets(char *str)
int puts(const char *str)
```

它们的参数均为一个指向 `str` 内存地址的指针, 也就是 `str` 数组的基地址, 保存在参数寄存器 `%rdi` 中
因此只要从汇编代码中确定 `%rdi` 的值, 就能知道 `buf` 的基地址为 `%rdi = %rbx = %rsp + 8`

- 在分配栈帧前, `%rbx` 先被保存在栈上, 因此 `%rsp - 栈帧的大小` 就是 `%rbx` 的位置
这点可以通过最后 `addq $16, %rsp` 回收栈帧来验证, 所以 `%rbx` 保存在 `%rsp + 8` 上
4. 栈上内容如下所示 (每一个方格代表一个字节):



- i. 返回地址保存在 caller 栈帧上;
- ii. 先执行了 `pushq %rbx`, 将 `%rbx` 压入栈中, 由于是X86-64因此 `%rbx` 为 64/8=8个字节;
- iii. `subq $16, %rsp` 分配16字节的 callee 栈帧。

以下的长度是指 `strlen()` 结果，即输入的字符串长度（不计入 `NULL`）：

- 写入 `buf` 但不溢出：长度介于 `[0, 7]`，此时 `buf[7]` 恰好为 `NULL`，不会溢出；
- 写入 `buf` 且更新栈上 `%rbx` 但不更新返回地址：即要求 `NULL` 覆盖栈上 `%rbx` 中的任意一个字节，且不会覆盖返回地址，那么长度介于 `[8, 15]`：
 - 8 的时候 `buf[7]` 为最后一个输入字符，`NULL` 恰好覆盖栈上 `%rbx` 的最低字节；
 - 15 的时候 `NULL` 恰好覆盖栈上 `%rbx` 的最高字节，并不影响返回地址。
- 写入 `buf` 且更新栈上 `%rbx` 和返回地址：由上一小题可知长度至少为 `15 + 1 = 16`，此时 `NULL` 恰好覆盖返回地址的最低字节；由于我们只考虑攻击是否成功，因此输入的字符串长度上限没有限制，即长度为 `[16, +inf)`。

最终答案为：

空格	值
(1)	<code>%rdi</code>
(2)	callee-saved
(3)	8
(4)	16
(5)	7
(6)	8
(7)	15
(8)	16

Q5 Shellcoding（8分）

代码注入攻击（也称为Shellcoding），是在缓冲区溢出的基础上，向缓冲区注入攻击指令（专业术语为shellcode），通过修改函数的返回地址，实现代码的任意执行，步骤如下：

1. 攻击者找到可以缓冲区溢出攻击的函数：

```
int Q() {
    char buf[64];
    gets(buf); // 漏洞！
    ...
    return ...;
}
```

2. 攻击者构造一段输入，包括了shellcode，以及保存了shellcode的栈上地址；
3. 程序执行到 `int Q()` 处，读取攻击者构造的输入，此时返回地址被覆盖为栈上的地址，栈则保存了攻击指令；
4. `int Q()` 执行 `ret` 指令（对应于 `return` 语句），跳转到攻击者构造的攻击指令，进而执行攻击。

回答以下问题：

1. 请说明如何修改源代码，以修复缓冲区溢出的漏洞。（2分）

在上一题中我们研究了 `gets()` 的原型，它最大问题在于不限制输入串的长度，很容易导致缓存区溢出；可以改用显式限制字符串长度的 `fgets()` 或内含检查机制的 `std::getline()` 来规避这个问题。

2. 即使不能修改代码，也有多种措施可以防御代码注入攻击。请结合上面的攻击流程，解释下面的措施为什么可以防御代码注入攻击：（4分）

i. No-eXecute (NX)：把栈标记为不可执行

把栈标记为不可执行，使得即使通过缓冲区溢出攻击跳转到栈上的指令，尝试执行时，CPU检测到现处于非可执行区域，抛出异常。

ii. Stack Canaries：在栈上保存"金丝雀值"

在缓冲区之上的栈内保存"金丝雀值"，攻击者无法预测"金丝雀值"，在当前函数 `ret` 之前检查这个值是否被修改，若被修改直接退出。

3. ROP攻击利用程序已有的指令来攻击，它利用了 x86-64 指令的特性，即指令的后缀可能也是一个合法的指令。对于上一题提到的两个措施，它可以绕过哪一个？但是必不可以绕过哪一个？（2分）

ROP是复用已有代码来构建代码片段(gadgets)，能应对NX下很难插入二进制代码的问题，但无法应对 Stack Canaries。

作业五

本次作业共包含3道大题，总计25分。

Q1 页表（16分）

已知内存是字节可寻址的、每次内存访问针对的是32-bit的word、虚拟地址24位、物理地址20位、页面大小为4096字节、TLB是二路组相联，共有16个TLB项（即2-way set associative with 16 total entries）。在下面的表格中，所有的数字都是十六进制的。TLB和页表前32项的内容如下：

TLB			
Index	Tag	PPN	Valid
0	011	1C	1
	02C	B3	1
1	13C	A4	0
	0B2	7E	1
2	001	05	1
	1A3	B6	0
3	002	08	1
	003	17	1
4	1C2	21	1
	013	09	0
5	1CF	38	1
	08B	51	1
6	003	7A	1
	13C	7F	1
7	031	B2	0
	0A4	3C	1

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	04	1	10	03	1
01	13	0	11	24	0
02	28	1	12	1E	1
03	1F	1	13	08	1
04	3E	1	14	02	1
05	6C	0	15	2A	1
06	09	1	16	3C	0
07	17	0	17	3B	0
08	24	1	18	1C	1
09	07	1	19	16	0
0A	05	1	1A	21	0
0B	2D	1	1B	17	1
0C	06	0	1C	22	1
0D	3B	1	1D	2E	0
0E	21	1	1E	7A	1
0F	08	0	1F	4B	1

Q1.1 地址格式（2分）

在图上标注出虚拟、物理地址中所包含的字段（下面所列举的字段若存在则标记，若不存在跳过即可）：

参考课件《7-虚存》P.20 的基本参数和符号。

约定虚拟地址为 `vaddr[]`、物理地址为 `paddr[]`，使用Python切片来描述：如 `vaddr[1:12]` 表示虚拟地址的第 1 位至第 11 位（从 0 开始，区间左闭右开）、`vaddr[12:]` 表示虚拟地址第 12 位至最后一位。

由 $P = 4096 = 1024 \times 4 = 2^{12} \Rightarrow p = 12$ 可知 `VP0` 与 `PP0` 的位宽为12位，于是有 `VP0 = vaddr[0:12]`、`VPN = vaddr[12:] = vaddr[12:]` `PP0 = paddr[0:12]`，`PPN = paddr[12:]`。

使用 `VPN` 去查询TLB，因此 `TLBI` 和 `TLBT` 为 `VPN` 的子域。从上表中可知 TLB 一共有 8 个 set，index 为 `[0, 8)`，每个 set 为二路组相连：`TLBI` 为3位，即 `TLBI = VPN[0:3] = vaddr[12:15]`；余下为 `TLBT = VPN[3:] = vaddr[15:]`。

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TLBT									TLBI			VPO											
VPN																							

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PPN								PPO											

Q1.2 虚拟地址转换（14分，一项1分）

对于给定的两个虚拟地址 0x01DBE3 、 0x9E6CF2 ，请分别写出 相应的TLB 表项和物理地址，并指出：

- TLB 是否命中（回答 Y 或 N ）
- 是否发生 page fault（回答 Y 或 N ）
 - 以页表前 32 项为准；即如果不在页表前 32 项内，则发生page fault；系统保证page fault异常处理后可以解决问题
 - 如果发生 page fault，请在 PPN 一项填 - 。
- 获得该地址所存储数据的访问时间（以下简称 "访问时间"）：
 - 假设一次内存访问时间100ns，一次快表（TLB）访问时间为10ns，处理一次缺页需要108 ns （含更新TLB和页表的时间）

参考课件《7-虚存》P.28~33的示例。

假定所有的数据保存在内存中，对照上面的地址格式，写出虚拟地址对应的域后依次查TLB和页表，并记录从获取有效物理地址到取出实际数据的实际用时：

- 0x01DBE3 ：
 - i. VPO = 0xBE3 ， VPN = 0x1D ， TLBI = 0x5 ， TLBT = 0x003
 - ii. 查询 index = 5 的 TLB set，不包含 TLB tag 0x003 ， 触发 TLB Miss：用时 10 ns
 - iii. 按 VPN = 0x1D 查询页表， Valid 位为 0 ， 触发 Page Fault（见课件《7-虚存》P.23）：用时 100 ns
 - iv. 处理 Page Fault，更新页表与TLB：用时 108 ns （见下面勘误）
 - v. 再次访问TLB，TLB Hit：用时 10 ns
 - vi. 获得物理地址，访问内存取出数据：用时 100 ns总共用时：10 + 100 + 108 + 10 + 100 = 328 ns
- 0x9E6CF2 ：
 - i. VPO = 0xCF2 ， VPN = 0x9E6 ， TLBI = 0x6 ， TLBT = 0x13C
 - ii. 查询 index = 6 的 TLB set，包含 TLB tag 0x13C ， TLB Hit： PPN = 0x7F ， 用时 10 ns
 - iii. 获得物理地址，访问内存取出数据：用时 100 ns总共用时：10 + 100 = 110 ns

	0x01DBE3	0x9E6CF2
VPN	0x01D	0x9E6
TLB Index	0x5	0x6
TLB Tag	0x003	0x13C
TLB Hit? (Y/N)	N	Y
Page Fault? (Y/N)	Y	N
PPN	-	0x7F
访问时间（单位为ns）	328	110

⚠ 勘误：实际缺页处理用时

缺页处理需要向外存发出请求，取出新的页表加载到内存中，显然 外存访存速度 $\ll$ 内存访存速度（回顾课件《3.1-C语言与汇编》P.7）。

本次作业出题时出现了笔误：处理缺页用时实际为 10^8 ns，尽管不影响作答，严谨起见在此特别纠正。

Q2 重定位（3分）

foo.c 的内容如下所示，其中 test_call() 共执行5次调用，依次为 test1 ~ test5：

```
// foo.c
extern void (*test1)();

static void test2() {}

void test3() {}

extern void test4();

void test_call(void (*test5)()) {
    test1();
    test2();
    test3();
    test4();
    test5();
}
```

现生成 foo.o 文件，在链接期间需要全局重定位的调用有哪些（即生成的 foo.o 文件中，需要重定位的符号有哪些）？

在链接阶段需要全局重定位的：**被引用的** 全局变量、全局函数入口地址（见课件《6.1 汇编与C语言-5》P.25小结）。

所以需要全局重定位的调用为 test1，test3，test4。

Q3 fork()（6分，每题2分）

阅读程序，并回答问题（每题2分）：

1. 该程序共输出多少行 hello？

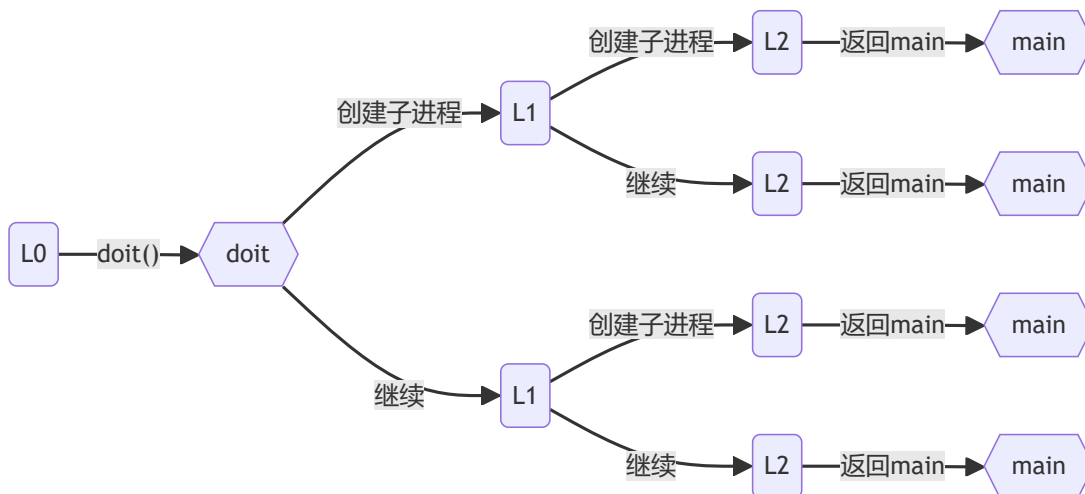
```

void doit() {
    fork(); // L0 -> L1, L1
    // L1 从这开始
    fork(); // L1 -> L2, L2, 注意有两个L1, 所以有两个L2
    // L2 从这开始
    printf("hello\n"); // 1次
    return;
}

int main() {
    doit(); // L0
    // L2 从doit()返回后继续执行
    printf("hello\n"); // 2次
    exit(0);
}

```

画出父子进程关系：



所有L2在 doit() 中输出一行 hello 后返回（各进程中的） main() 再输出一行 hello，**总共8行 hello**。

2. 该程序共输出多少行 hello？

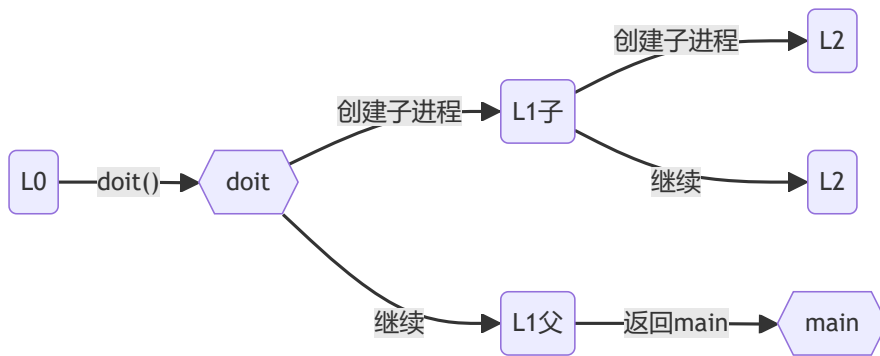
```

void doit() {
    if (fork() == 0) { // L0 -> L1(子), L1(父), L1(子) 继续
        fork(); // L1 -> L2, L2
        printf("hello\n");
        exit(0);
    }
    return; // L1(父) 从这里返回
}

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

```

画出父子进程关系：



doit() 里第一次 fork() 后，父进程 (L1父) 直接返回 main()；子进程 (L1子) 继续 fork() L2，它们输出后立即 exit(0)；而不会返回 main()，**总共3行 hello**。

3. 该程序输出的 counter 值应为多少？

```
int counter = 1; // 从相同的状态开始，但各自具有私有副本
int main() {
    if (fork() == 0) {
        counter--;
        exit(0);
    } else {
        wait(NULL);
        counter++;
        printf("counter = %d\n", counter);
    }
    exit(0);
}
```

考察点在于 fork() 了之后，尽管从相同状态开始，父子进程实际各自具有私有副本（见课件《9.1-异常控制流》P.27~P.30）。

即，子进程修改自身的 counter，不会影响父进程的 counter。在这里只有父进程会输出（自身的）counter 的值，所以**输出的 counter 的值为 2**。

作业六

本次作业一共有4题，总计30分，同时也是本学期最后一次作业。

Q1 文件读写（2+8分）

假设文件 `file1.txt` 中有一个字符串 `aabbccdd`。

下列C文件分别被编译成 `./program1` 和 `./program2`：

```

/* Program 1 */
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char const *argv[]) {
    int pid, fd_x, fd_y, fd_z;
    char buf[8];

    fd_x = open("file1.txt", O_RDWR);
    fd_y = open("file1.txt", O_RDWR);
    fd_z = open("file1.txt", O_RDWR);

    read(fd_x, buf, 2);
    read(fd_y, buf + 2, 4);

    // -----

    if ((pid = fork()) == 0) {
        dup2(fd_x, STDOUT_FILENO);
        dup2(fd_y, STDIN_FILENO);
        execl("program2", "program2", NULL);
    }

    wait(NULL);

    // -----

    read(fd_y, buf + 6, 2);
    write(fd_z, buf + 6, 2);
    write(fd_x, buf + 4, 2);
    write(fd_x, buf + 2, 2);

    close(fd_x);
    close(fd_y);
    close(fd_z);

    return 0;
}

```

```

/* Program 2 */
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char const *argv[]) {
    char buf2[2];
    read(STDIN_FILENO, buf2, 2);
    write(STDOUT_FILENO, buf2, 2);
}

```

对所调用的一些方法的解释

- `open(filename, O_RDWR)` : 打开一个已有文件 `filename` , 对其进行读写操作, 起始位置为 `0` ;
- `execl(exename,...)` : 执行 `exename` 程序, 与我们讲的 `execve()` 功能类似。

当执行 `./program1` 后, 文件 `file1.txt` 的内容是什么? (2分)

从以下几点解释一下为什么是这个结果 (8分, 一点2分, 言之有理即可, **建议不超过256个字**) :

- `./program2` 做了什么?
- `./program1` 在 `fork()` 之前做了什么, 相应的 `buf` 的内容如何变化?
- `./program1` 调用 `fork()` 了以后, 子进程在干什么?
- 子进程返回后, `./program1` 又做了什么, 相应的 `buf` 的内容如何变化?

参考课件《10-IO处理》P.26~32, 同时建议完成 P.33~35 的练习题。

对于同一文件描述符 `fd` 来说, 每次成功的 `read()` 和 `write()` , 都会更新当前文件的访问位置 (见课件《10-IO处理》P.8,9 的提示) , **此次的偏移量即当前成功读写的字节数**:

`read()` , `write()` 更新文件访问位置

这里节选了关于文件访问位置更新的部分内容, 详细可通过 [Linux manual pages](#) 查阅:

- [read\(2\) — Linux manual page](#):
On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and `read()` returns zero.
- [write\(2\) — Linux manual page](#):
For a seekable file (i.e., one to which [lseek\(2\)](#) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was [open\(2\)](#)ed with `O_APPEND` , the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

以 `^` 表示当前文件的访问位置, 举例来说:

```

int fd_x = open("file1.txt", O_RDWR);
/* fd_x: aabbccdd
   ^

*/

read(fd_x, buf, 2);
/* fd_x: aabbccdd
   ^

*/

write(fd_x, "gg", 2);
/* fd_x: aaggccdd
   ^

*/

```

阅读源代码，可知 `./program2` 从自己的标准输入流 `stdin` 中读入2个字节保存到 `buf[]` 后，再将 `buf[]` 写入自己的标准输出流 `stdout` 中。

按照第二小题的提示，我们可以分为三个部分来分析：

不妨假定初始时 `buf` 为"空"（每一个元素为 `'\0'`），即 `buf = ""`。

在 `fork()` 之前：

1. 从 `fd_x` 中读入2个字节，即 `"aa"`，保存到 `buf + 0` 上：`buf = "aa"`；
2. 从 `fd_y` 中读入4个字节，即 `"aabb"`，保存到 `buf + 2` 上：`buf = "aaaabb"`。

此时各文件描述符的访问位置如下：

```

/* fd_x: aabbccdd
   ^
* fd_y: aabbccdd
   ^
* fd_z: aabbccdd
   ^

*/

```

调用 `fork()` 了以后，子进程通过 `dup2()` 将它的 `stdin` 重定向至 `fd_y`、`stdout` 重定向至 `fd_x`，然后执行 `./program2`（为了方便区分，下面以 `buf2` 表示 `program2.c` 中的 `buf[]`）：

1. 先从 `fd_y` 中读入两个字节 `"cc"`，保存到 `buf2 + 0` 上：`buf2 = "cc"`；
2. 将 `buf2` 保存的两个字节 `"cc"` 写入 `fd_x` 中：`file1.txt` 的内容为 `aacccdd`

父进程等待子进程结束后再继续，继续前各文件描述符的访问位置与内容如下：

```
/* fd_x: aacccdd
      ^
* fd_y: aacccdd
      ^
* fd_z: aacccdd
      ^
*/
```

子进程返回后：

1. 从 `fd_y` 中读入两个字节 "dd"，保存到 `buf + 6` 上：`buf = "aaaabdd"`
2. 将 `buf + 6` 起始的两个字节 "dd" 写入 `fd_z` 中：`file1.txt` 的内容为 `ddccdd`
3. 将 `buf + 4` 起始的两个字节 "bb" 写入 `fd_x` 中：`file1.txt` 的内容为 `ddccbdd`
4. 将 `buf + 2` 起始的两个字节 "aa" 写入 `fd_x` 中：`file1.txt` 的内容为 `ddccbbaa`

在 `close()` 之前，不妨看一下各文件描述符的访问位置：

```
/* fd_x: ddccbbaa
      ^
* fd_y: ddccbbaa
      ^
* fd_z: ddccbbaa
      ^
*/
```

最终 `file1.txt` 的内容为 `ddccbbaa`。

Q2 信号量（10分，1空1分）

老师的办公室有一个空白板。老师会在白板为空时往白板上写一道物理题或一道化学题。如果是一道物理题，喜爱物理的小A会解答出这道题并把题目擦掉，如果是一道化学题，喜爱化学的小B会解答出这道题并把题目擦掉，请使用信号量和 `P`、`V` 原语实现老师、小A、小B三者的同步。

```

sem_t board;          // 白板是否为空
sem_t physics;        // 白板上是否为物理题
sem_t chemistry;      // 白板上是否为化学题

void init() {
    Sem_init(&board, 0, __ (A) __);
    Sem_init(&physics, 0, __ (B) __);
    Sem_init(&chemistry, 0, __ (C) __);
}

void teacher() {
    while (1) {
        Course c = (rand() & 1) ? PHYSICS : CHEMISTRY;
        __ (D) __;
        在白板上写题目;
        if (c == PHYSICS) {
            __ (E) __;
        } else { // c == CHEMISTRY
            __ (F) __;
        }
    }
}

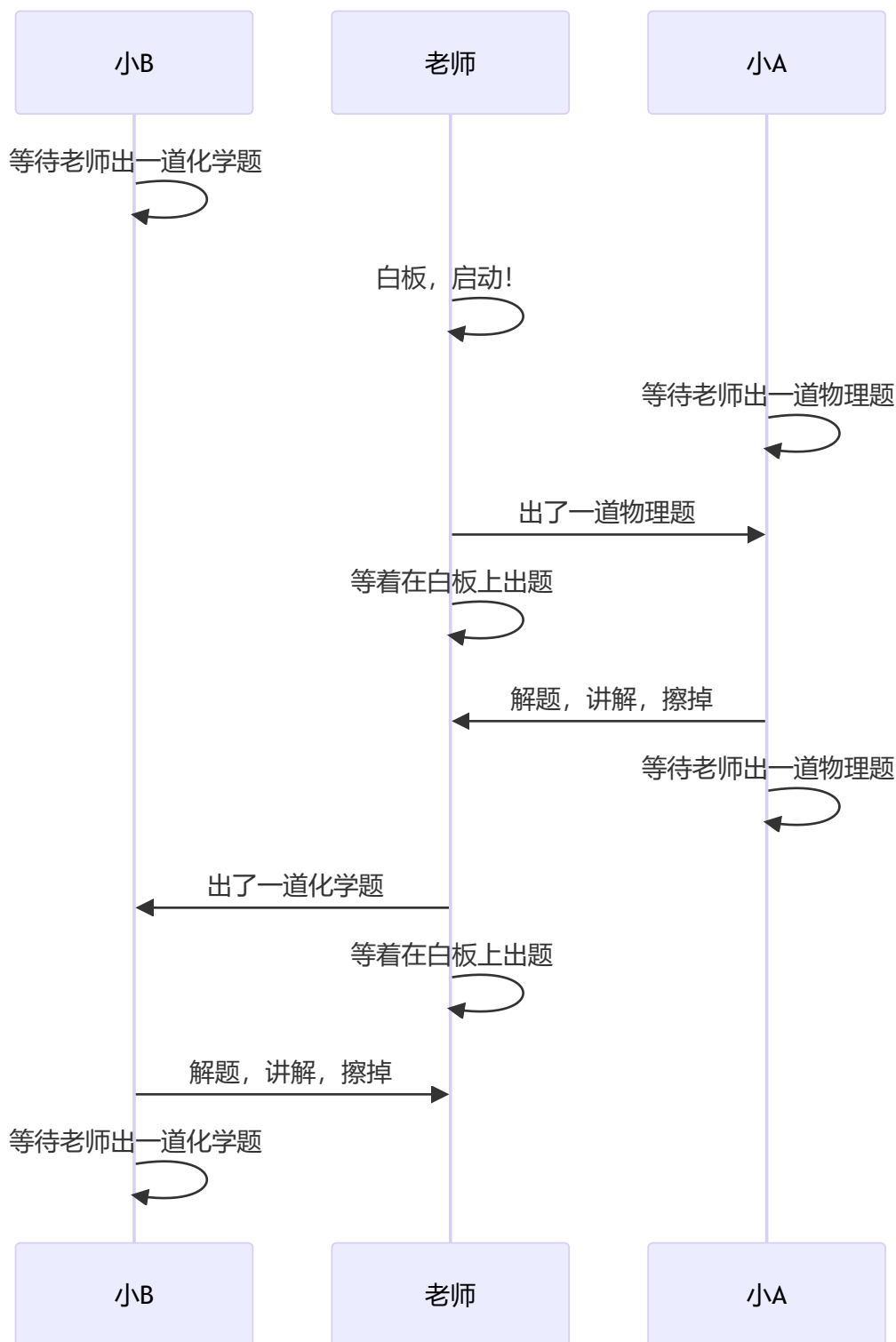
void studentA() {
    while (1) {
        P(__ (G) __);
        解答物理题, 将其擦掉;
        V(__ (H) __);
    }
}

void studentB() {
    while (1) {
        P(__ (I) __);
        解答化学题, 将其擦掉;
        V(__ (J) __);
    }
}

```

P, V 原语的定义见课件《11-线程与线程同步基础》P.31, 这里不加赘述。

不妨分析刚开始时白板为空, 老师先出一题物理题让小A作答, 再出一题化学题让小B作答的过程:



获得这样的序列图后，结合 P，V 原语可知：

- "自环"表示当前线程正在监听某个信号量直到其大于 0，对应于 P()；
- "单向传递"表示线程设置某个信号量以通知其他线程工作，对应于 V()。

同样也能明确信号量的设置：

- 老师监听 board，设置 physics 和 chemistry；小A监听 physics，小B监听 chemistry，他们作答完毕后设置 board；
- 初始时小A和小B都在监听，直到老师出题，因此 physics 和 chemistry 初值为 0；
- 白板一开始就是空的，这样老师就能开始出题；同时为了保证老师能顺利等待 board 被释放，P(&board) 结束后应进入等待（即 board 的值为 0），所以 board 初值只能为 1。

综上，答案为：

空格	值
A	1
B	0
C	0
D	P(&board)
E	V(&physics)
F	V(&chemistry)
G	&physics
H	&board
I	&chemistry
J	&board

注：__空格(G)__ 至 __空格(J)__ 中漏写 &、信号量漏写字母 不扣分。

Q3 信号处理（3分，选对满分）

考虑如下程序：

```
void handler (int sig) {
    printf("D");
    exit(4);
}
int main() {
    int pid, status;
    signal(SIGINT, handler);
    printf("A");
    // -----
    pid = fork();
    printf("B");
    // -----
    if (pid == 0) {
        printf("C");
    } else {
        kill(pid, SIGINT);
        waitpid(pid, &status, 0);
        printf("%d", WEXITSTATUS(status));
    }
    printf("E");
    exit(7);
}
```

以下哪些是可能的输出结果（多选，根据标答完全匹配给分）： _____

A. ABCBE7E B. ABD7E C. ABBCE4E D. ABCDB4E E. ABBD4E

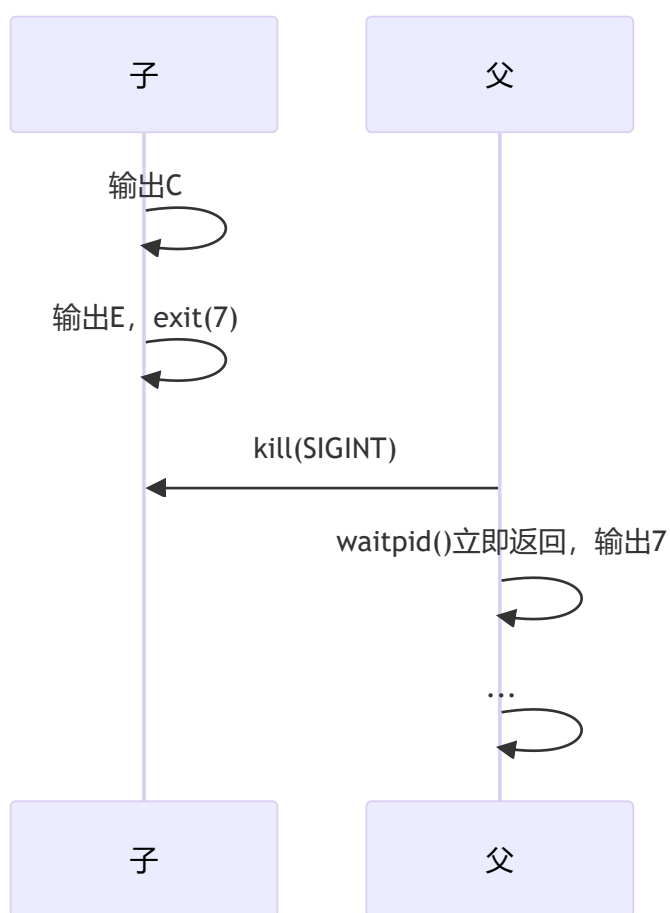
面对多选题，不妨采取排除法；与第一题类似，我们分成几个阶段来分析：

在 `fork()` 之前：

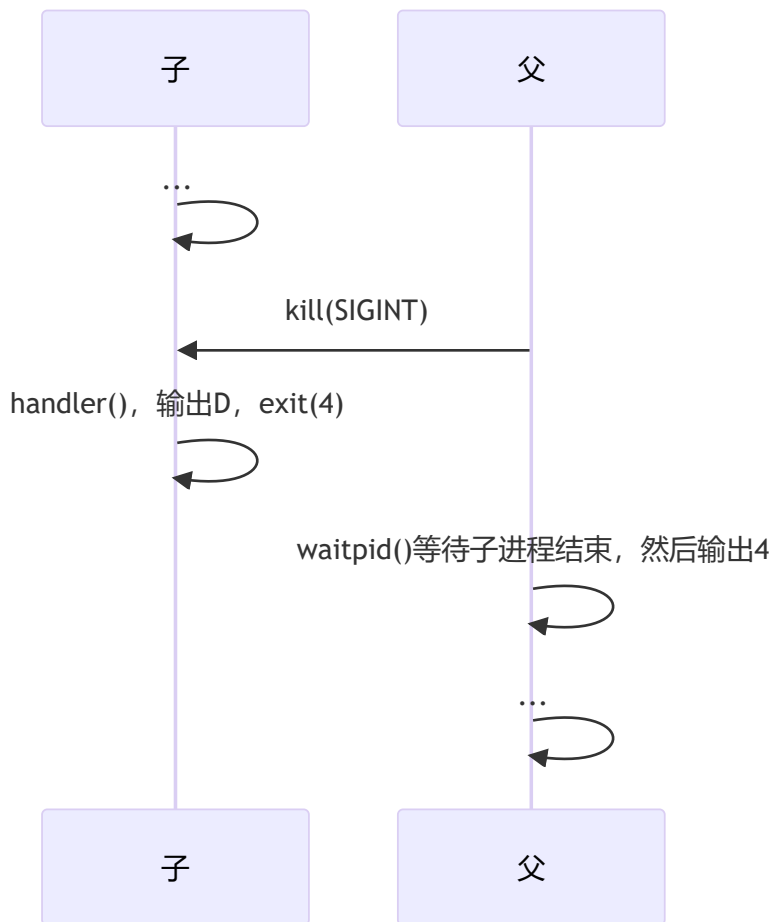
1. 通过 `signal()` 修改接收到信号 `SIGINT` 后，触发 `handler`
2. 在 `fork()` 之前只有一个进程，因此首先必定会输出一个 `A`：选项A,B,C,D,E符合

考虑到父进程与子进程的调度可能存在差异，分两种情况讨论：

- 子进程在 `exit(7)` 后接收到父进程的 `SIGINT`



- 子进程在 `exit(7)` 之前接收到父进程的 `SIGINT`



- 由上述分析可知，输出结果中 4 和 D 要么同时存在要么同时不存在，不可能只出现其中之一，选项C是不合理的；
- 父进程与子进程都会输出一个 B，除非子进程被提前kill了（同上，4 和 D 同时存在），而选项B的输出只有一个 B，有 D 却没有 4，不合理；
- 如果子进程是在输出 B 后才被kill，对应的父进程也应该执行完 printf("B")；操作才会执行 kill()，因此2个 B 必须出现在 4 和 D 之前，在操作系统顺序响应中断请求的情况下，选项D是不合理。

简单描述A,E对应的情况（可以画出序列图来辅助思考）来验证是否合理：

- 选项A：子进程在结束后，SIGINT 才发出去
- 选项E：子进程在输出 C 之前就接收到 SIGINT

选项 A,E 对应的情况是合理的。

Q4 线程与子进程（7分，全填对满分）

阅读程序写结果，**要求列出所有可能输出**，不考虑进程/线程创建失败的情况，并假设 printf 的输出不会被其他 printf 打断。换行请用 \n 表示。

为了方便大家看得清楚，参考解答中"人为地"在 \n 前后加入空格，实际输出是不包含空格的；同时，漏写末尾的 \n 不扣分。

1. a=1 \n

```
// (1)
#include <pthread.h>
#include <stdio.h>
int a = 0;
void* test(void* ptr) { a++; return NULL; }
int main() {
    pthread_t pid;
    pthread_create(&pid, NULL, test, NULL);
    pthread_join(pid, NULL);
    printf("a=%d\n", a);
    return 0;
}
```

只有一个线程使用 `a`，不会发生竞争。

2. `a=1 \n a=1 \n`

```
// (2)
#include <unistd.h>
#include <stdio.h>
int a = 0;
void test() { a++; }
int main() {
    int pid = fork();
    test();
    printf("a=%d\n", a);
    return 0;
}
```

父进程与子进程各有一个变量 `a` 的拷贝，且都执行了 `a++`，都会输出 `a=1`。

3. `a=1 \n` 或 `a=2 \n`

```
// (3)
#include <pthread.h>
#include <stdio.h>
int a = 0;
void* test(void* ptr) { a++; return NULL; }
int main() {
    pthread_t pid1, pid2;
    pthread_create(&pid1, NULL, test, NULL);
    pthread_create(&pid2, NULL, test, NULL);
    pthread_join(pid1, NULL);
    pthread_join(pid2, NULL);
    printf("a=%d\n", a);
    return 0;
}
```

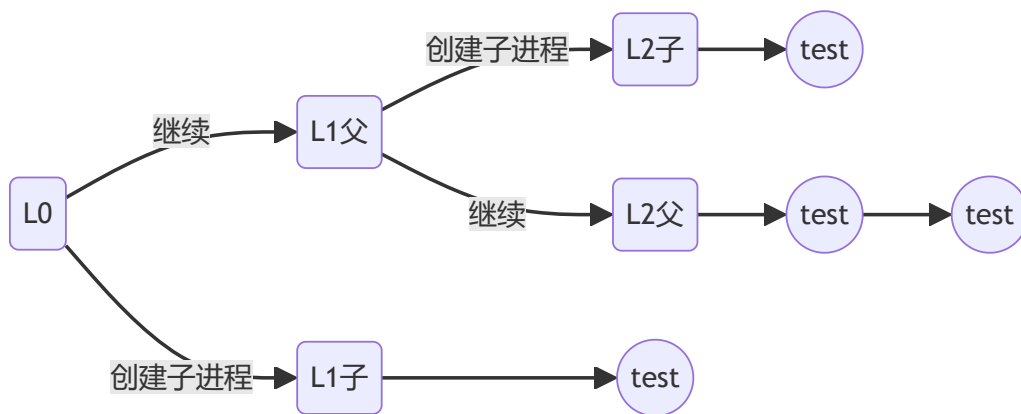
具体结果取决于 `pid1` 和 `pid2` 之间对于共享变量 `a` 的竞争 (races) 是否发生：

- 如果发生，只有一个线程抢占到 `a`，`a=1`；
- 如果没发生，两个线程先后使用 `a`，`a=2`。

4. `a=1 \n a=1 \n a=2 \n` 或 `a=1 \n a=2 \n a=1 \n` 或 `a=2 \n a=1 \n a=1 \n`

```
// (4)
#include <unistd.h>
#include <stdio.h>
int a = 0;
void test() { a++; }
int main() {
    int pid = fork();
    if (pid != 0) pid = fork();
    test();
    if (pid != 0) test();
    printf("a=%d\n", a);
    return 0;
}
```

记住：子进程 `pid == 0`，父进程的 `pid` 为子进程的进程号
画出父子进程关系：



按照三个进程可能的先后顺序写出输出即可。