第二章

【整数的表示】

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	...
...		0x56	0x34	0x12	0x00	...	

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	...
...		0x34	0x12	0x78	0x56	...	

2. 在 x86-64 机器上，有下列 C 代码

```
int main() {
    unsigned int A = 0x11112222;
    unsigned int B = 0x33336666;
    void *x = (void *)&A;
    void *y = 2 + (void *)&B;
    unsigned short P = *(unsigned short *)x;
    unsigned short Q = *(unsigned short *)y;
    printf("0x%04x", P + Q);
    return 0;
}
```

运行该代码，结果为：0x_____。【答案：0x5555】

3. 在 x86-64 机器上，有下列 C 代码

```
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，结果为：0x_____。【答案：0x0303】

【整数的运算】

4. 在 x86-64 机器上，有如下的定义：

```
int x = _________;
int y = _________;
```

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

判断下列表达式是否等价：

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。

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

【答】(1)取 x=1, y=-1 即不正确；(2)取 x=-1 即不正确；(3)正确，利用交换律、结合律，以及 x ^ x == 0；(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. 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。

```
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);
}
```

【答：sizeof 的结果是无符号数，因此 pos - sizeof(B) < 0 恒假，于是不会进行任何复制。】

【实数的表示】

6. 假设某浮点数格式为 1 符号+3 阶码+4 小数。下表给出了用该格式表达的浮点数$f = (-1)^S \times M \times 2^E$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	10000000	----------	--------	-0.0
----------	01000101	21/16	1	21/8
最小的非规格化负数	10001111	15/16	-2（注意不是-3）	-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
正无穷	01110000	----------	--------	-----

7．假设浮点数格式 A 为 1 符号+3 阶码+4 小数，浮点数格式 B 为 1 符号+4 阶码+3 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？

【答】 只有一个 01110000

(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？

【答】对于 A 格式，01101111 表示了 31/16*2^3=31/2=15.5，对于 B 格式，01110111 表示了 15/8*2^7=240，因此 B 大。

(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？

【答】 对于 A 格式，00000001 表示了 1/16*2^(-2)=1/64，对于 B 格式，00000001 表示了 1/8*2^(-6)=1/512，因此 A 大。

(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？

【答】A 能精确表达的非负数个数为 7*16=112，B 能精确表达的非负数个数为 15*8=120，因此 B 能精确表达的实数更多。实际上，A 格式表示 NaN 的数比 B 格式多。

【浮点数的运算】

8．判断下列说法的正确性

	描述	正确吗？
(1)	对于任意的单精度浮点数 a 和 b，如果 a > b，那么 a + 1 > b。	Y N
(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。	Y N
(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。	Y N
(4)	对于任意的双精度浮点数 d，如果 d < 0，那么 d * d > 0。	Y N
(5)	对于任意的双精度浮点数 d，如果 d < 0，那么 d * 2 < 0。	Y N
(6)	对于任意的双精度浮点数 d，d == d。	Y N
(7)	将 float 转换成 int 时，既有可能造成舍入，又有可能造成溢出。	Y N

【答】(1)正确;(2)取 a = INF、b = FLT_MAX；(3)取 a = 16777220、b = 16777218 即可。这里要特别注意取 a = INF、b = FLT_MAX 不能构成反例，因为 b + 1 因计算精度无法到达 INF;(4)d 取最大的非规格化负数;(5)正确;(6)NAN != NAN;(7)正确。

9．已知 float 的格式为 1 符号+8 阶码+23 小数，则下列程序的输出结果是：

```
for (int x = 0; ; x++) {
```

```
    float f = x;
    if (x != (int)f) {
           printf("%d", x);
           break;
    }
}
```

A. 死循环

B. 4194305 ($2^{22}+1$)

C. 8388609 ($2^{23}+1$)

D. 16777217 ($2^{24}+1$)

【答】D. 表示 16777217 需要 25 位，除了前导 1 以外还要 24 位，float 无法表示。

10. 已知 float 的格式为 1 符号+8 阶码+23 小数，有下列代码：

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

其运行结果是：2 -1 0 1 -2 -1 0 1 。

【答】注意 Round to Even 规则。

ICS 第三章

【操作数格式、数据传送】

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)

【答】1 不正确；2 不正确，$；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（参考 Intel 手册 Vol.1 3-23 与 Vol.2A 2-7）。

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。

	%rax	%rbx
movabsq $0x0123456789ABCDEF, %rax		
---->	0x0123456789ABCDEF	0x0000000000000000
movw %ax, %bx		
---->	0x0123456789ABCDEF	(1) ?????????????
movswq %bx, %rbx		
---->	0x0123456789ABCDEF	(2) ?????????????
movl %ebx, %eax		
---->	(3) ?????????????	(2) ?????????????
Movabsq $0x123456789ABCDEF, %rax		
---->	0x0123456789ABCDEF	(2) ?????????????
cltq		
---->	(4) ?????????????	(2) ?????????????

(1) 0x000000000000CDEF
(2) 0xFFFFFFFFFFFFCDEF
(3) 0x00000000FFFFCDEF
(4) 0xFFFFFFFF89ABCDEF

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

【答】C;　A．B．movzbl/movzwl 生成了四字节，把高位设为 0；　D．cltq 是对%eax 的符号拓展；　C．movl 和 movzlq 等价（所以不需要 movzlq 这条指令）

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

【答】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，这里要用 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。

【加载有效地址、算术运算】

5．在 32 位机器下，假设有如下定义
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

【答】C

6．将下列汇编代码翻译成 C 代码

func: 1 movq %rdi, %rax 2 salq $4, %rax 3 subq %rdi, %rax 4 movq %rax, %rdi 5 leaq 0(,%rsi,8), %rax 6 subq %rsi, %rax 7 addq %rdi, %rax 8 ret	*// a in %rdi, b in %rsi* long func(long a, long b) { return _____; }

【答】a * 15 + b * 7：第 1 行%rax = a；第 2 行%rax = 16a 第 3 行%rax = 15a；第 4 行%rax = %rdi = 15a；第 5 行%rax = 8b；第 6 行%rax = 7b；第 7 行%rax = 15a + 7b

【条件码】

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)

【答】A;　如果 a > b，那么 a – b 要不为正，要不发生上溢变负；同时 a 不等于 b。

【条件分支】

8. 将下列汇编代码翻译成 C 代码

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	*// a in %rdi, b in %rsi* long func(long a, long b) { long ans = _____; while (_____) { if (_____) ans = _____; b = _____; a = _____; } return ans; }

【答】

```
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．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。

long f1(long a, long b) { return (++a > --b) ? a : b; }	Y N
long f2(long *a, long *b) { return (*a > *b) ? --(*a) : (*b)--; }	Y N
long f3(long *a, long *b) { return a ? *a : (b ? *b : 0); }	Y N
long f4(long a, long b) { return (a > b) ? a++ : ++b; }	Y N

【答】f1 由于比较前计算出的 a 与 b 就是条件传送的目标，因此会被编译成条件传送；f2 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；f3 由于指针 a 可能无效，因此会被编译为条件跳转；f4 会被编译成条件传送，注意到 a 和 b 都是局部变量，return 的时候对 a 和 b 的操作都是没有用的。使用-O1 可以验证 gcc 的行为。

10．根据汇编指令补充机器码中缺失的字节。

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

【答】03; f8．程序进行到第五行时，开始跳转，跳转位置为 0xf8(-8) + 0x4004dd = 0x4004d5，注意当程序走到第五行进行跳转之前，PC 指向该指令的下一条指令，即第六行 repz 的开头

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
```

将主函数的汇编代码转换成 C 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
```

```
// 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;
}
```

【答】

```
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
```

【过程调用】

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 (_____, _____);
}
```

【答】n == 0; m; n – 1, m * n

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;
    return;
}

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

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

```
caller:
    pushq  %r12
    pushq  %rbp
    pushq  %rbx
    movq   %rdi, %rbp
    movq   %rsi, %r12
    movl   $0, %ebx
    jmp .L4
```

```
    xorq   %rax, (%rdi)
.L1:
    rep ret
```

```
.L5:
    movq   %rbp, %rax
    subq   %rbx, %rax
    leaq   (%r12,%rax,8), %rsi
    leaq   (%r12,%rbx,8), %rdi
    call   callee
    addq   $1, %rbx
.L4:
    movq   %rbp, %rax
    shrq   $63, %rax
    addq   %rbp, %rax
    sarq   %rax
    cmpq   %rbx, %rax
    jg  .L5
    popq   %rbx
    popq   %rbp
    popq   %r12
    ret
```

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（不确定的空格填-）
%rsp	0xffffffffff80	0xf...f88~8f	-
%rax	0x0	0xf...f80~87	-
%rbx	0x15	0xf...f78~7f	0x4000b1
%rbp	0x18	0xf...f70~77	0x213
%r12	0x213	0xf...f68~6f	0x18
%rsi	0x0	0xf...f60~67	0x15
%rdi	0x0	0xf...f58~5f	0x40008d
		0xf...f50~57	-

【答】注意不要弄错 Return Address 的内容。

【结构与联合】

14．在 x86-64、Linux 操作系统下有如下 C 定义：

```
struct A {
    char CC1[6];
    int II1;
    long LL1;
    char CC2[10];
    long LL2;
    int II2;
};
```

(1) sizeof(A) = _____字节。
(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。

【答】56，40；CC1：0；II1：8，必须 4 字节对齐；LL1：16，必须 8 字节对齐；CC2：24；LL2：40，必须 8 字节对齐；II2：48。基本元素最大的为 long，因此 sizeof(A)是 56。重排顺序：LL1 LL2 II1 II2 CC1 CC2，刚好没有空白空间，得到的大小为 8+8+4+4+10+6=40 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;
```

回答如下问题：

(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、将 h 的类型改成 int，那么 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___，___，___，___。

【答】

4; 8; 12; 16; 14; 8; 8, 16; 3, 7, 12, 16。具体解释如下：

(1)~(4):

d1	d2	d3		h							i				
				c1	c2	c3	c4	c5	c6	c7					

(5):

d1	d2	d3		h								i	
				c1	c2	c3	c4	c5	c6	c7			

(6):

d1	d2	d3		h							i				
				c1	c2	c3	c4	c5	c6	c7					

(7):

d1	d2	d3		h							i			
				c1	c2	c3	c4	c5	c6	c7					

对于(7)，虽然和(5)很像，并且 sizeof(union_e)也没变，但是实际上对齐要求的是所有基本元素都要对齐。所以在考虑 sizeof 的时候，不妨假设开辟一个包含两个元素的结构体数组，检查一下第二个结构体是否对其了所有的基本元素。

(8):

d1	d2	d3	c1	c2	c3	c4	c5	c6	c7			i

【调试工具】

16. 写出使用 gcc 编译源代码 lab.c、生成可执行文件 lab、采用二级编译优化的命令。

【答】gcc lab.c -o lab -O2

17. 写出使用 gcc 编译源代码 foo.c、生成汇编语言文件 foo.s 的命令

【答】gcc foo.c -S

18. 将可执行文件 bar 逆向工程为汇编代码的工具是(　　)
A. gcc　　B. gdb　　C. objdump　　D. hexedit　　E. gedit

【答】C

19. gdb 中，单步指令执行的命令是
A. r　　B. b　　C. p　　D. finish　　E. si　　F. disas

【答】E

【综合】

20. 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

汇编:

```
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 语言与堆栈:

```
typedef struct{
    long a;
    long b;
} pair_type;
long func(pair_type *p) {
    if (p -> a < p -> b) {
        long temp = p -> a;
        p -> a = p -> b;
        p -> b = temp;
    }
    if ((7)          ) {
        return p -> a;
    }
    pair_type np;
    np.a = (8)          ;
    np.b = (9)          ;
    return func(&np);
}
int main(int argc, char* argv[]) {
    pair_type np;
    np.a = (10)          ;
    np.b = (11)          ;
    printf("%ld", func(&np));
    return 0;
}
```

.....
0x0000000000000000
0xc76d5add7bbeaa00
0x00007fffffffdf60
(a)
(b)
0x0000000000400629
(c)
(d)
0x0000000000000001
0x0000000000000069
0x0000000000000093
(e)
0x00000000ff000000
(f)
0x0000000000000000
(g)
(h)
(i)
0x0000000000000000
(j)
(k)
0x000000000000002a
0x000000000000003f
0x00000000004005e2
（栈顶）[低地址]

一些可能用到的字符的 ASCII 码表:

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

回答问题：

I. 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

0x4006e4: 0x_____ 0x_____ 0x_____ 0x_____

II. 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）。

(1) _____<func+____>__
(2) _____<func+____>__
(3) _____
(4) _____
(5) _____
(6) _____
(7) _____
(8) _____
(9) _____
(10)_____
(11)_____

III. 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。

(a) _____
(b) _____
(c) _____
(d) _____
(e) _____
(f) _____
(g) _____
(h) _____
(i) _____
(j) _____
(k) _____

IV. 程序运行结果为_____。

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

汇编代码：

```
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    4005bd <func+0x27>
  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    4005e2 <func+0x4c>
  4005cb:  mov    (%rdi),%rdx
  4005ce:  sub    %rax,%rdx
  4005d1:  mov    %rdx,(%rsp)
  4005d5:  mov    %rax,0x8(%rsp)
  4005da:  mov    %rsp,%rdi
  4005dd:  callq  400596 <func>
  4005e2:  mov    0x18(%rsp),%rcx
  4005e7:  xor    %fs:0x28,%rcx
  4005f0:  je     4005f7 <func+0x61>
  4005f2:  callq  400460 <__stack_chk_fail@plt>
  4005f7:  add    $0x28,%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    %fs:0x28,%rdx
  400649:  je     400650 <main+0x54>
  40064b:  callq  400460 <__stack_chk_fail@plt>
  400650:  mov    $0x0,%eax
  400655:  add    $0x28,%rsp
  400659:  retq
```

C 语言与 stack 的情况：

```
typedef struct{
    long a;
    long b;
} pair_type;
long func(pair_type *p) {
    if (p -> a < p -> b) {
        long temp = p -> a;
        p -> a = p -> b;
        p -> b = temp;
    }
    if (p -> b == 0) {
        return p -> a;
    }
    pair_type np;
    np.a = p -> a - p -> b;
    np.b = p -> b;
    return func(&np);
}
int main(int argc, char* argv[]) {
    pair_type np;
    np.a = 105;
    np.b = 252;
    printf("%ld", func(&np));
    return 0;
}
```

```
......
0x0000000000000000(u)
0xc76d5add7bbeaa00
0x00007fffffffdf60(u?)
0x0000000000000069
0x00000000000000fc
0x0000000000400629
0x0000000000000000(u)
0xc76d5add7bbeaa00
0x0000000000000001(u)
0x0000000000000069
0x0000000000000093
0x00000000004005e2
0x00000000ff000000(u)
0xc76d5add7bbeaa00
0x0000000000000000(u)
0x000000000000002a
0x0000000000000069
0x00000000004005e2
0x0000000000000000(u)
0xc76d5add7bbeaa00
0x000000ff00000000(u)
0x000000000000002a
0x000000000000003f
0x00000000004005e2
（栈顶）[低地址]
```

一些可能用到的字符的 ASCII 码表：

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

回答问题：

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？

   0x4006e4: 0x25 0x6c 0x64 0x00

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）。
   (1) 4005bd <func+0x27>
   (2) 4005e2 <func+0x4c>
   (3) %rsp
   (4) %fs:0x28
   (5) je
   (6) $0x28
   (7) p -> b == 0
   (8) p -> a - p -> b
   (9) p -> b
   (10) 105
   (11) 252

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。
   (a) 0x0000000000000069
   (b) 0x00000000000000fc
   (c) 不确定
   (d) 0xc76d5add7bbeaa00
   (e) 0x00000000004005e2
   (f) 0xc76d5add7bbeaa00
   (g) 0x000000000000002a
   (h) 0x0000000000000069
   (i) 0x00000000004005e2
   (j) 0xc76d5add7bbeaa00
   (k) 不确定

4. 程序输出结果是？

   21

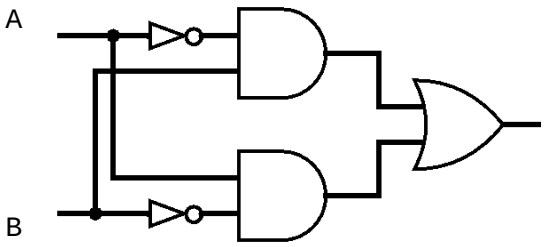
【注意】
1 最后一空为 0x00，因为要用\0 作字符串结尾
2(1)(2)较难
2(10)(11)不能颠倒，因为汇编如此写
3(a)(b)两空不能颠倒，因为 func 函数会修改内存的值
程序的作用是用（类似于）更相减损术求最大公约数

【体系结构基础】

1. 下列描述更符合（早期）RISC 还是 CISC?

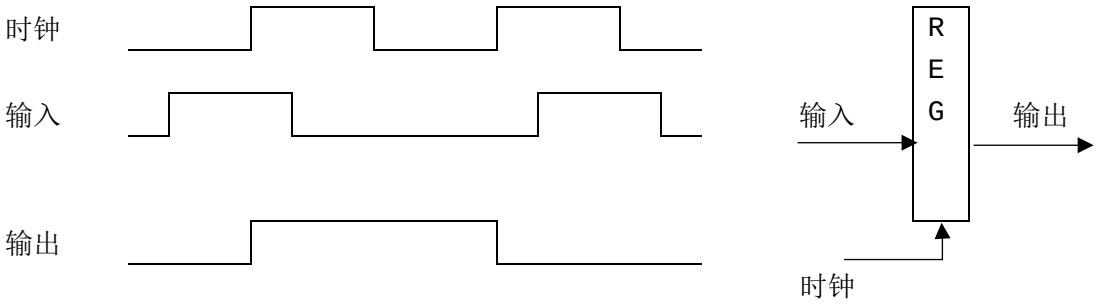
	描述	RISC	CISC
(1)	指令机器码长度固定	√	
(2)	指令类型多、功能丰富		√
(3)	不采用条件码	√	
(4)	实现同一功能，需要的汇编代码较多	√	
(5)	译码电路复杂		√
(6)	访存模式多样		√
(7)	参数、返回地址都使用寄存器进行保存	√	
(8)	x86-64		√
(9)	MIPS	√	
(10)	广泛用于嵌入式系统	√	
(11)	已知某个体系结构使用 add R1,R2,R3 来完成加法运算。当要将数据从寄存器 S 移动至寄存器 D 时，使用 add S,#ZR,D 进行操作（#ZR 是一个恒为 0 的寄存器），而没有类似于 mov 的指令。	√	
(12)	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. 写出下列电路的表达式



【答】(!A && B) || (A && !B)

3. 下列寄存器在时钟上升沿锁存数据，画出输出的电平（忽略建立/保持时间）



【顺序处理器】

3．根据 32 位 Y86 模型完成下表

		call	jXX
Fetch	icode,ifun	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]
	rA, rB	\	
	valC	valC <- M4[PC+1]	valC <- M4[PC+1]
	valP	valP <- PC+5	valP <- PC+5
Decode	valA,srcA	\	\
	valB,srcB	valB <- R[%esp]	\
Execute	valE	valE <- valB + -4	\
	Cond Code	\	Cnd <- Cond(CC, ifun)
Memory	valM	M4[valE] <- valP	\
Write back	dstE	R[%esp] <- valE	\
	dstM	\	\
PC	PC	PC <- valC	PC <- Cnd?valC:valP

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 : valC;
    icode == IJXX && Cnd: valC;
    icode == IRET : valM;
    1: valP;
]
```

【流水线的基本原理】

5．判断下列说法的正确性

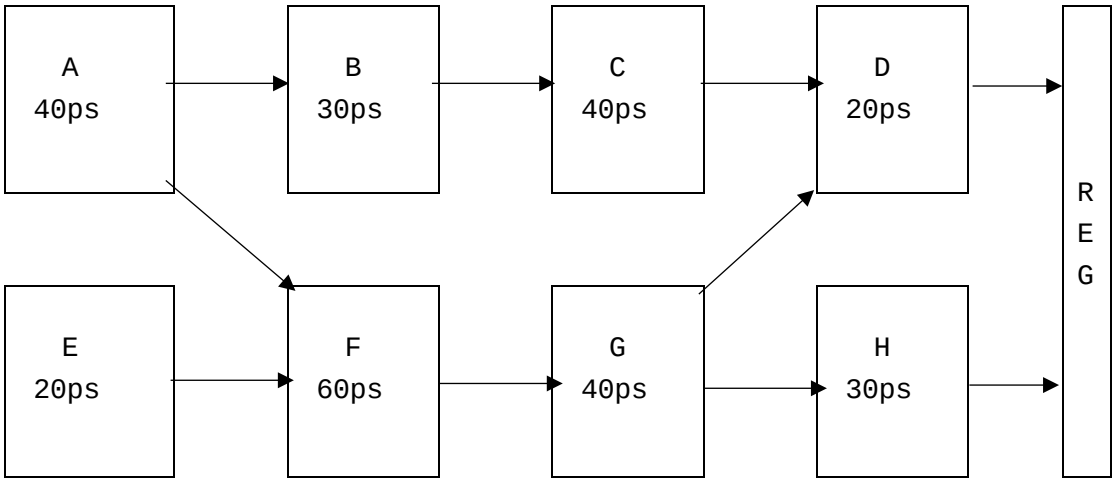
(1) (　　)流水线的深度越深，总吞吐率越大，因此流水线应当越深越好。
(2) (　　)流水线的吞吐率取决于最慢的流水级，因此流水线的划分应当尽量均匀。
(3) (　　)假设寄存器延迟为 20ps，那么总吞吐率不可能达到或超过 50 GIPS。
(4) (　　)数据冒险总是可以只通过转发来解决。
(5) (　　)数据冒险总是可以只通过暂停流水线来解决。

【答】N, Y, Y, N(mrmov + add), Y

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

【答】100+10+100+10+100+10=330ps, 1000/(100+10)=9.09 GIPS

7．A~H 为 8 个基本逻辑单元，下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 10ps。



(1) 计算目前的电路的总延迟。
【答】40+60+40+30+10=180ps

(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？
【答】BC 与 FG 之间。1000/(40+60+10)=9.09GIPS

(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？
【答】插在 AB、AF、EF、BC、FG 之间。1000/(40+30+10)=12.5GIPS

【流水线处理器】

8．一个只使用流水线暂停、没有数据前递的 Y86 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？

irmovl $1, %eax irmovl $2, %ebx addl %eax, %ecx addl %ebx, %edx halt	rrmovl %eax, %edx mrmovl (%ecx), %eax addl %edx, %eax halt	irmovl $0x40, %eax mrmovl (%eax), %ebx subl %ebx, %ecx halt
(1)	(2)	(3)

【答】(1) 2 个。第五个时钟周期结束以后，第三行代码才能开始译码。而原来第三行在第四个时钟周期开始译码，因此需要两周期停顿。(2) 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 3 个停顿。(3) 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。

9．考虑 Y86 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）：

```
IRET in {D_icode, E_icode, M_icode}
```

(2) 写出发现上述条件以后，流水线寄存器应设置的状态

	Fetch	Decode	Execute	Memory	Writeback
处理 ret	(any)	bubble	normal	normal	Normal

(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）：

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

(4) 写出发现上述条件以后，流水线寄存器应设置的状态

	Fetch	Decode	Execute	Memory	Writeback
分支错误	(any)	bubble	bubble	normal	Normal

(5) 写出下一条指令地址 f_pc 的控制逻辑

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

#  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
];
```

【答】(1) 容易。(2)由于第一次发现 ret 是在 Decode 阶段，因此 Execute 阶段应当设置为 normal，否则下一周期 ret 无法执行；下一周期，ret 后的指令进入 Decode 阶段，这是一条错误指令，因此 Decode 应当设置为 bubble。Fetch 阶段无所谓。(3) 容易。(4) 由于第一次发现 jXX 是在 Execute 阶段，因此 Memory 阶段应当设置为 normal，否则下一周期 jXX 无法执行；jXX 后面的两条指令均为错误指令，下一周期它们将进入 Decode 和 Execute 阶段，因此这两个阶段均为 bubble；而 jXX 正确地址在 valP 中，因此可以使下一周期取到正确的指令。 (5)在 Decode 阶段，valP 进了 d_valA，在 Execute 结束以后就可以将正确的 PC（自增）传回去，此时它在 M_valA 里。当 Memory

阶段结束以后，ret 才能从内存中取出正确的地址，因此正确地址在 W_valM 里。

10. (2016 期中 cret 题)
11. (2018 期中最后一题)

【程序性能优化】

1．有如下的定义：

```
// 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将下列左侧代码优化为右侧代码：

(1)

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

【答】会

(2)

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

【答】不会，因为浮点数不能结合

(3)

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

【答】不会，因为 foo 可能有副作用

(4)

`*p += *q;` `*p += *r`	`temp = *q + *r` `*p += temp;`

【答】不会，如果 pqr 指向同一个元素那么两个运算不等价

2．阅读下列 C 代码以及它编译生成的汇编语言

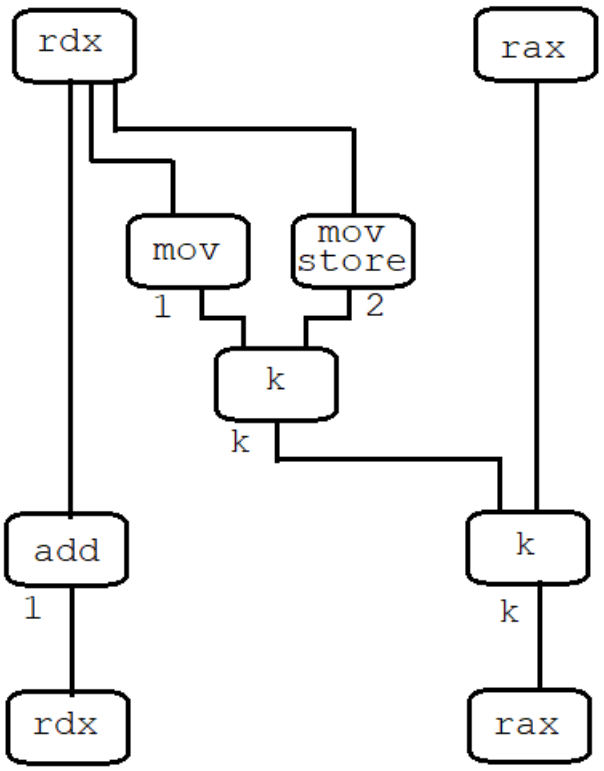
```
long func() {
    long ans = 1;
    long i;
    for (i = 0; i < 1000; i += 2) {
        ans = ans ? (A[i] ? A[i+1]);
    }
    return ans;
}
```

```
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 cycles
.L2:
    cmpq   $999, %rdx                 // 1 cycles
    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 4。
(2) 当问号处为加法时，k = 1。此时这段程序的 CPE 为 0.5。

【答】数据相关图如下



【存储技术】

3．对于下列描述，是 SRAM 更符合还是 DRAM 更符合，还是均符合？

		S	D
(1)	访问速度更快	S	
(2)	每比特需要的晶体管数目少		D
(3)	单位容量造价更便宜		D
(4)	常用作主存		D
(5)	需要定期刷新		D
(6)	断电后失去存储的信息	S	D
(7)	支持随机访问	S	D

4．勾出所有的易失性存储介质

(　)DRAM　(　)SRAM　(　)ROM　(　)软盘　(　)SSD　(　)U 盘

【答】DRAM 与 SRAM

5．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 8.192 GB。

【答】2*2*10000*400*512 = 8,192,000,000 Byte = 8.192GB

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

A．6　　B．8　　C．10　　D．14

【答】6ms + 0.5*(60/7500*1000)ms = 10ms

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

A．0.020　　B．0.025　　C．0.040　　D．0.050

【答】60 / 6000 （转一圈的时间）* 1 / 400（转过一个磁道的时间）* 1000 = 0.025ms

8．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。
【答】错误，正确，正确，正确

【高速缓存】

9．一个容量为 8K 的直接映射高速缓存，每一行的容量为 32B，那么它有 256 组，每组有 1 行。

10．一个容量为 8K 的全相联高速缓存，每一行的容量为 32B，那么它有 1 组，每组有 256 行。

11．一个容量为 16K 的 4 路组相联告诉缓存，每一行的容量为 64B，那么一个 16 位地址 0xCAFE 应映射在第 43 组内。

【答】计算 16K/(64*4)=64 组，因此组号长度为 6，块内偏移长度为 6 字节，因此 0xCAFE 的组号为 43。

12．考虑下列的程序

```
// A 有定义 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，运行之前 cache 所有行都是无效的。A 的起始地址为 0。

(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 24 %

(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 0 %

(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）

	有效位	内容	有效位	内容
组 0	1	M[96 - 103]	1	M[80 - 87]
组 1	1	M[104 - 111]	1	M[88 - 95]

Cache 命中率是 86 %

(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 72 %

13．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。

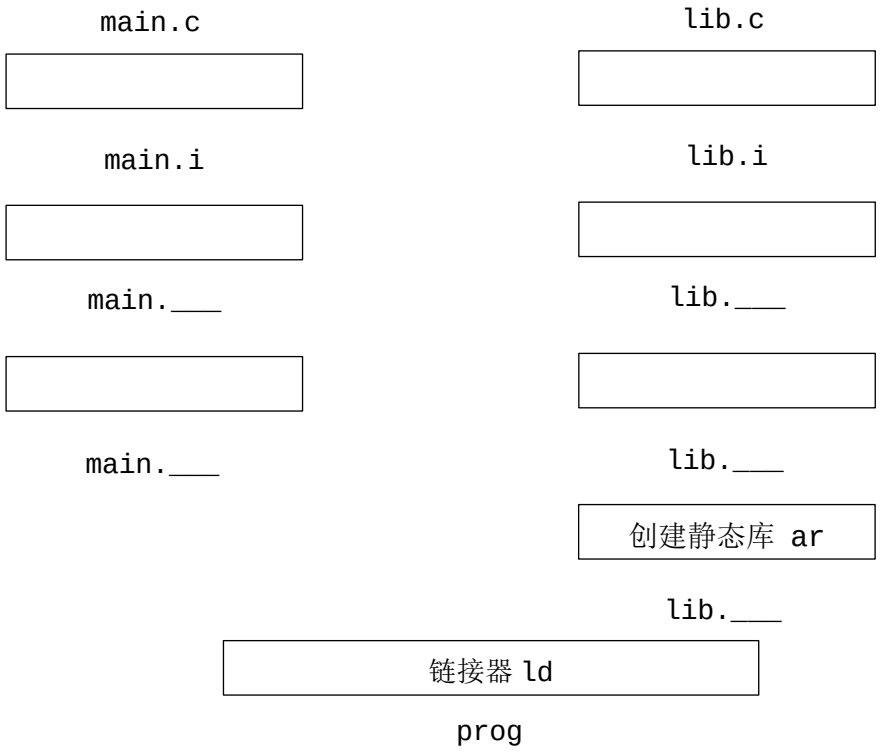
【答】1 正确；2 错误；3 错误；4 正确。

ICS 第七章

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。

A. 汇编器 as　　B. 预处理器 cpp　　C. 编译器 ccl

D. *.a　　E. *.s　　F. *.o



【答】BCA、EFD

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。

【答】(1)错，只会拷贝被用到的模块。(2)错，会把所有函数拷贝出来。(3)对。(4)错。考虑下面这个极端的例子。foo.o 中定义了 p5()并引用了 p0()；对于所有的 i=1,2,3,4,5，bar.o 中定义了 *pi-1*()并引用了 *pi*()。Q135.a 包含了 bar1,bar3,bar5，R24.a 包含了 bar2,bar4。输入文件为 Q135.a，R24.a 与 foo.o。可以证明，gcc foo.o Q135.a R24.a Q135.a R24.a Q135.a 是最短的命令，并且 Q135.a 必须要出现至少 3 次、R24.a 必须要出现至少 2 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。

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;
}
```

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，那么在名字那一栏打×；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.o	x	x	×	√
	bar	bar	×	×
	ans	ans.1597	√	
count.o	x	x	×	×
	bar	bar	×	√
	ans	ans.0344	√	

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？

1 1 3 3 6 6

(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.o	x	x	√	
	bar	bar	×	×
	ans	ans.1597	√	
count.o	x	x	√	
	bar	bar	×	√
	ans	ans.0344	√	

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？1 1 3 2 6 3。两个 x 在各自的.o 文件中的名字都为 x，因为它们不是过程中的静态变量。思考：对于非过程间的静态变量，为什么**编译器**不需要作这样的区分？

(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？

链接错误，x 被定义多次。

4. 有如下 C 代码

```
#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：

节头：

[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

t.txt 中的部分符号表如下：

Symbol table '.symtab' contains ?? entries:

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

PART A. 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
√/×	×	×	×	√	×	×	×

PART B. 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）。

PART C. 问答题

(1) 字符串"%ld %ld\n"位于哪个节中？_____。

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

(2) 假设，在全局区定义 long A[1000000]。那么在 test.o 中，.bss 节占用的空间为_____字节。

PART D. 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。

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，之后五个空格是机器码）

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_____。

PART E. 对 done 使用 objdump，发现有如下的函数：

```
0000000000400430 <_start>
000000000040054b <main>
00000000006000f0 <__libc_start_main@GLIBC_2.2.5>
```

则 done 的入口点地址是 0x_____。

【答】

C. (1) C (2) 0

D. 由题意，
ADDR(s) = ADDR(.text) = 0x400517
ADDR(r.symbol) = ADDR(bar) = 0x400517
r.offset = 0x6c
r.addend = -4
因此
refaddr = ADDR(.text) + r.offset = 0x400583
*refptr = ADDR(r.symbol) + r.addend – refaddr = 0xFFFFFF90
所以代码地址为 400582，机器码为 e8 90 ff ff ff
0x40069e-0xa=0x400694

E. 0x400430

ICS 第八章

【异常的基本概念】

1. 区分各个异常（打√）

异常的种类	是同步（Sync）的吗？	可能的行为？		
		重复当前指令	执行下条指令	结束进程运行
中断 Interrupt			√	
陷入 Trap	√		√	
故障 Fault	√	√		√
终止 Abort	√			√

行为	中断	陷入	故障	终止
执行指令 `mov $57, %eax; syscall`		√		
程序执行过程中，发现它所使用的物理内存损坏了				√
程序执行过程中，试图往 `main` 函数的内存中写入数据			√	
按下键盘	√			
磁盘读出了一块数据	√			
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			√	

【fork】

2. 阅读下列程序

```
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：

(1) （ √ ）AABBBC (2) （ √ ）ABCABB (3) （ × ）ABBABC

(4) （ × ）AACBBC (5) （ √ ）ABABCB (6) （ × ）ABCBAB

【wait】

3. 阅读下列程序

```
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：

(1) （ √ ）ABBCCD (2) （ √ ）ABBCDC (3) （ × ）ABBDCC

(4) （ × ）ABDBCC (5) （ × ）ABCDBC (6) （ × ）ABCDCB

【信号】

4. 阅读下列程序

```
void handler() {
    printf("D\n");
    return;
}
int main() {
    signal(SIGCHLD, handler);
    if (fork() > 0) {
        /* parent */
        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（忽略换行）：

(1) （ √ ）ACBC (2) （ √ ）ABCCD (3) （ × ）ACBDC

(4) （ × ）ABDCC (5) （ √ ）BCDAC (6) （ √ ）ABCC

5. 在 2018 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，运行它以后，输出能创建的进程的最大数目。下面是几位同学的答案。

PART A. 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。

【答】将(pid = fork()) != 0 改为(pid = fork()) > 0 即可

PART B. 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的上限。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。

【答】Bob 的说法不正确。子进程结束以后变成僵死进程，继续占用系统资源。

PART C. 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);
}
```

运行 Carol 同学的答案两次，发现结果分别如下：

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

(1) 解释为什么会发生这种情况。

【答】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。

(2) 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：

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

再假设，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分配。

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。请填入正确的值。

```
linux $ kill -9 _______
```

A. 22725　　B. 22724　　C. -22725　　D. -22724

【答】D. 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。这里要 kill 掉整个进程组，子进程的进程组号均为 22724，所以应当选择 D。

PART D. 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：

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

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？

【答】他加了\n，这样缓冲区就会被立刻刷新。否则下一句 kill 进程组（包括自己）以后，缓冲区的内容还没来得及写进 stdout。

ICS 第九章

【虚拟内存机制】

1．在某一 64 位体系结构中，每页的大小为 4KB，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_____Bit。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。这个体系结构支持多种页大小，最小的三个页大小分别是 4KB、_____MB、_____GB

【答】VPO 长度为 12 位。4KB/32=512 个条目每页表。因此 VPN 长度均为 9 位，虚拟地址长度为 12+9+9+9=39 位。由于 12+9*5=57<64，12+9*6=66>=64，因此需要采用六级页表才能映射满。

2．Intel IA32 体系中，每页的大小为 4KB，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（4KB 页）	指向的页表的物理地址 31~12 位					0					US	RW	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（4KB 页）	虚拟页的物理地址 31~12 位					0					US	RW	V

如果采用一级翻译（大页模式），那么每个 PDE 条目格式如下：

	31~22	21~17	16~13	12	11	10	9	8	7	6	5	4	3	2	1	0
PDE（4MB 页）	虚拟页的物理地址 31~22 位	00000	物理 35~32 位						1					US	RW	V

上表中的最低位均为第 0 位。部分位的意义如下：

V=1：当前的条目有效（指向的页在物理内存中）
RW=1：指向的区域可写。只有两级页表均为 1 的时候，该虚拟内存地址才可以写。
US=1：指向的区域用户程序可访问。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	00	00DA0CA5D	9A	00DA0CA5E	88	00DA0CA5F	EF

不采用 TLB 加速翻译。

PART A. 现在需要访问虚拟内存地址 0x00A97088。

(1) 将该地址拆成 VPN1+VPN2+VPO

VPN1	VPN2	VPO
0x**2**	0x**297**	0x**088**

(2) 对应的 PDE 条目的物理地址是 0x00C188008，读出第二级页表的起始地址为 0x00D32A000，PTE 条目的起始地址为 0x00D32AA5C，因此翻译得到的物理地址为 0x09A83C088。

(3) 用户模式能否访问该地址？[Y/N] 能否写该地址？[Y/N] YY

PART B. 现在需要访问虚拟内存地址 0x3003C088。

最终翻译得到的物理地址为 0x09A83C088。

【答】VPN1=0xC0，PDE 条目物理地址为 0x00C188300，读出这一页是大页，地址高 35~22 位为 0x09A800000。因此物理地址为 0x09A83C088。

PART C. 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
```

【答】%ebx 与%edx 指向同一物理内存，因此答案为 0。

PART D. 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 0xC0300008, %eax
```

【答】C0002A5C，VPN1=0x300, VPN2=0x2，因此 PDE 条目的物理地址为 0x00C188C00，读出第二级页表的起始地址为 0x00C188000（就是第一级页表!），PTE 条目起始地址为 0x00C188008，读出页地址为 0x00D32A000，物理地址为 0x00D32AA5C，因此%eax 的值为 0x9A83C067。为了指向第一级页表的第三个条目（VPN1=2），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。因此前两次映射都应当映射到页表自己。发现如果 VPN1=0x300 的话，那么第一级映射就映射到自己了，于是如果 VPN2=0x300 的话，第二级映射也还是映射回自己。对应的 VPO=0x2*4=0x8，拼起来就是虚拟地址 0xC0300008。

【内存映射】

3. 有下列程序：

```
int main() {
    pid_t pid;
    int child_status;
    long* f = mmap(NULL, 8,
           PROT_READ | PROT_WRITE,
           [    X    ] | MAP_ANONYMOUS, -1, 0);
```

```
    *f = 0;
    if ((pid = fork()) > 0) {               // Parent
        waitpid(pid, &child_status, 0);
        *f = *f + 1;
        printf("Parent: %ld\n", *f);
    } else {                                // Child
        *f = *f + 1;
        printf("Child: %ld\n", *f);
    }
    return 0;
}
```

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？

【答】1 1；1 2

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\0。LINUX 采用写时复制（Copy-on-Write）技术。

```
char* f;
int count = 0, parent = 0, child = 0, done = 0;
void handler1() {
    if (count >= 4) {
        done = 1;
        return;
    }
    f[count] = '0' + count;
    count ++;
    kill(parent, SIGUSR2);
}
void handler2() {
    [ Y ]
    write(STDOUT_FILENO, f, 7);
    write(STDOUT_FILENO, "\n", 1);
    kill(child, SIGUSR1);
}
int main() {
    signal(SIGUSR1, handler1); signal(SIGUSR2, handler2);
    int child_status;
    parent = getpid();
    int fd = open("hello.txt", O_RDWR);
    if ((child = fork()) > 0) {             // Parent
        f = mmap(NULL, 8, PROT_READ | PROT_WRITE,
               MAP_PRIVATE, fd, 0);
        sleep(3);
        kill(child, SIGUSR1);
        waitpid(child, &child_status, 0);
    } else {                                // Child
```

```
        f = mmap(NULL, 8, PROT_READ | PROT_WRITE,
               MAP_SHARED, fd, 0);
        while (done == 0)
            ;
    }
    return 0;
}
```

(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？

【答】(1) 0BCDEFG、01CDEFG、012DEFG、0123EFG，文件内容为 0123EFG (2)四行 0BCDEFX，文件内容为 0123EFG。注意只有写时才复制。

5. 有如下 C 程序

```
int main() {
    char* hello;
    char* bye;
    int fd1 = open("hello.txt", O_RDWR);
    int fd2 = open("bye.txt", O_RDWR);
    hello = mmap(NULL, 16, PROT_READ,
        MAP_SHARED, fd1, 0);
    bye = mmap(NULL, 16, PROT_READ | PROT_WRITE,
        MAP_SHARED, fd2, 0);
    for (int i = 0; i < 8; i++)
        bye[i] = toupper(hello[i]);
    /************A**************/
    munmap(hello, 16);
    munmap(bye, 16);
    return 0;
}
```

代码运行到 A 处的时候，/proc/2333/maps 中的内容如下：

ADDRESS	PERM	PATH
00400000-00401000	(1)	/home/pw384/map/pstate
00600000-00601000	r--p	/home/pw384/map/pstate
00601000-00602000	rw-p	/home/pw384/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/pw384/map/bye.txt

```
7fb5975cc000-7fb5975cd000  (3)    /home/pw384/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]
```

PERM 有四位。前三位是 r=readable，w=writeable，x=executable，如果是-表示没有这一权限。第四位是 s=shared，p=private，表示映射是共享的还是私有的。填写空格的内容：

1: r-xp

2: rw-s

3: r--s

4: ld

5: stack

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？

【答】不会，在内核的映射表中标出它是可写的，因此这是 COW 还未发生的情况。

ICS 第十章

1. 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)	(2)	(3)
int main() { printf("a"); fork(); printf("b"); fork(); printf("c"); return 0; }	int main() { write(1, "a", 1); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }	int main() { printf("a"); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }

对于(1)号程序，写出它的一个可能的输出：_____。这个可能的输出是唯一的吗？_____。

对于(2)号程序，它的输出中包含_____个 a，_____个 b，_____个 c。输出的第一个字符一定是_____。

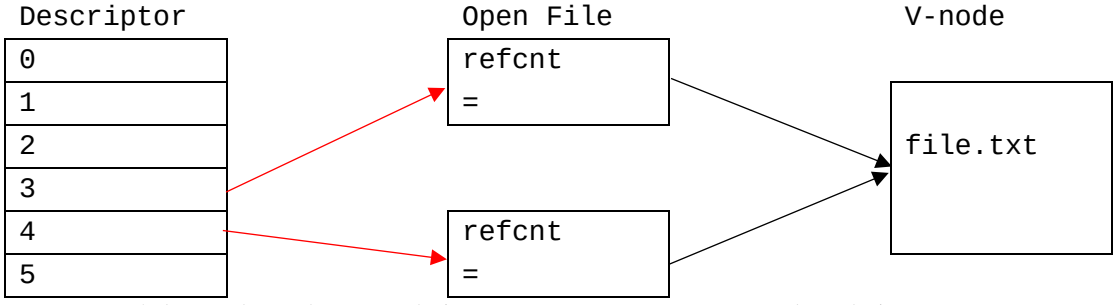
对于(3)号程序，它的输出中包含_____个 a，_____个 b，_____个 c。输出的第一个字符一定是_____。

【答】abcabcabcabc，是唯一的。1，2，4，a。4，2，4，b。

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

```
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，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。



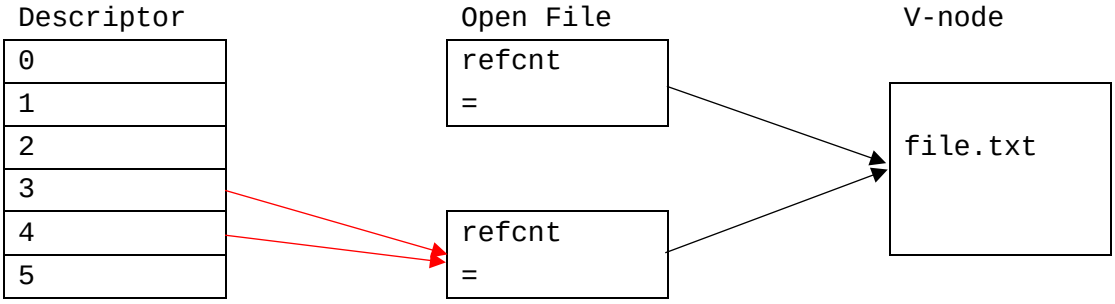
(2) 程序结束时，标准输出上的内容是_____，file.txt 中的内容是_____。

【答】(1) 1, 1 (2)3 4; 453。

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

```
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);
    return 0;
}
```

(1) 程序关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。



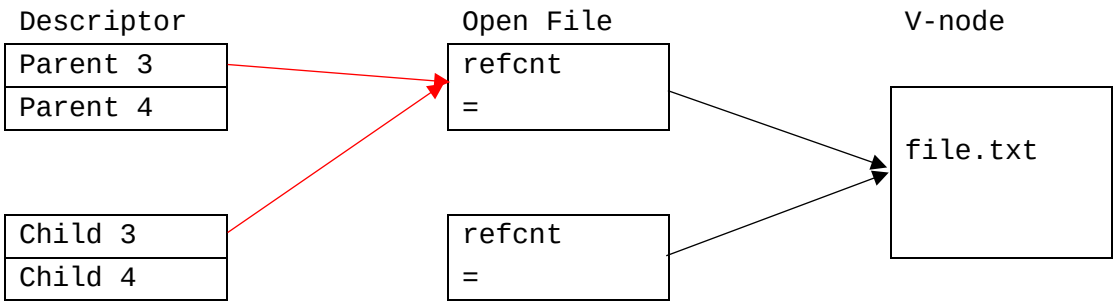
(2) 程序结束时，标准输出上的内容是_____，file.txt 中的内容是_____。

【答】(1) 0, 2; (2) 3 4; 12345。

4. 假设磁盘上有空文件 file.txt。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。

```
int main() {
    pid_t pid;
    int child_status;
    int fd1 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if ((pid = fork()) > 0) {
        // Parent
        printf("P:%d ", fd1);
        write(fd1, "123", 3);
        waitpid(pid, &child_status, 0);
    } else {
        // Child
        printf("C:%d ", fd1);
        write(fd1, "45", 2);
    }
    close(fd1);
    return 0;
}
```

(1) 子进程关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。



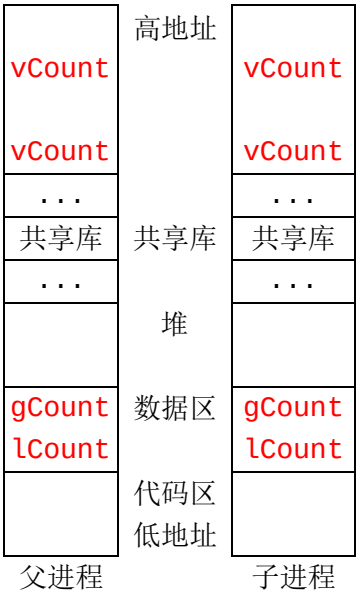
(2) 程序结束时，标准输出上的内容是_____，file.txt 中的内容是_____。

【答】(1) 2, 0; (2)C:3 P:3 ;12345 或 45123。注意 C 必在 P 前面输出。

ICS 第十二章

1. volatile 保证定义的变量存放在内存中，而不总是在寄存器里。右侧为两个进程的地址空间。请在合适的位置标出变量 gCount、vCount 与 lCount 的位置。如果一个量出现多次，那么就标多次。

```
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。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。

```
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;
}
```

【答】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 输出了 2 1，线程 2 将 bar 改为 2，并输出了 2 2。

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	①P(w) ②P(x) ③P(y) ④P(z) ⑤V(w) ⑥V(x) ⑦V(y) ⑧V(z)
线程 2	ⅠP(x) ⅡP(z) ⅢP(y) ⅣP(w) ⅤV(x) ⅥV(y) ⅦV(w) ⅧV(z)

【答】①→Ⅰ→Ⅱ→Ⅲ，此时线程 1 占用了 w 而在等待 x，线程 2 占用了 x 而在等待 w。

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。所有空缺语句均为 PV 操作。

全局变量：

stu_count: int 类型，表示考场中的学生数量，初值为 0

信号量：

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 **1**

mutex_all_present: 保证学生都到了，初值为 **0**

mutex_all_handin: 保证学生都交了，初值为 **0**

mutex_test[30]: 表示学生拿到了试卷，初值均为 **0**

```
Teacher:  // 老师
    P(mutex_door)
    从门进入考场
    V(mutex_door)

    P(mutex_all_present) // 等待同学来齐
    for (i = 1; i <= 30; i++)
        V(mutex_test[i]) // 给 i 号学生发放试卷

    P(mutex_all_handin) // 等待同学将试卷交齐
    P(mutex_door)
    从门离开考场
    V(mutex_door)
```

```
Student(x):  // x 号学生
    P(mutex_door)
    从门进入考场
    V(mutex_door)

    P(mutex_stu_count);
    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)
```

ICS 第十一章习题

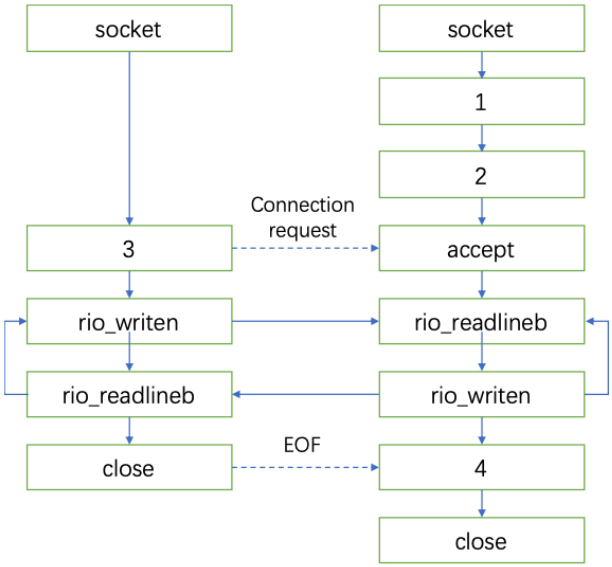
【TCP/IP 协议】请根据 web 应用在计算机网络中的定义以及其在协议栈自上而下在软件中的实现，把以下关键字填入表格

注：同一个关键词可能被填入多次；不是每一个关键词都必须被填入

Streams (end to end), Datagrams, web content, IP, TCP, UDP, Kernel code, User code

协议	数据包类型	软件实现
HTTP	Web content	User code
TCP	Streams	Kernel code
IP	Datagram	Kernel Code

【套接字编程】补充下面的空格



bind listen connect rio_readlineb

【套接字编程】判断下列说的的正确性

2018 期末

(　　) 套接字接口常常被用来创建网络应用

(　　) Windows 10 系统没有实现套接字接口

(　　) getaddrinfo()和 getnameinfo()可以被用于编写独立于特定版本的 IP 协议的程序

(　　) socket()函数返回的描述符，可以使用标准 Unix I/O 函数进行读写

(　　) 数字数据只能通过数字信号传输

对错对对错　还包括模拟信号。另，第四个判断有歧义。

2015 期末

( ) 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

( ) 在 client-server 模型中，套接字是一种文件标识符

( ) 准确地说，IP 地址是用于标识主机的 adapter (network interface card)，并非主机

( ) Web 是一种互联网协议

( ) 域名和 IP 地址是一一对应的

( ) Internet 是一种 internet

错对对错错对

说明:server 会为每一个 client 单独创建一个套接字进行通信;Web 是一种基于 HTTP 协议的互联网应用;一个域名可以对应多个 IP 地址，一个 IP 地址也可以对应多个域名

2016 期末/2014 期末

( ) HTTP 协议规定服务器端使用 80 端口提供服务

( ) 使用 TCP 来实现数据传输一定是可靠的

( ) Internet 上的两台的主机要通信必须先建立端到端连接

( ) 在 Linux 中只能通过 Socket 接口进行网络编程

( ) Web 浏览器与 web 服务器通信采用的协议是 HTML

错对错错错 80 是默认端口而非必须;IP 协议本身不基于连接，传输层中的 UDP 协议便不基于连接;可以通过网卡驱动接口直接收发数据;应该是 HTTP

【Web】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

① DNS ② TCP ③ IP ④ HTTP

4

【Web】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

C.

【套接字】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 上最多能监听多少个独立的 socket 连接？该服务器在所有 web 应用端口上最多可以监听多少个独立的 socket 连接?

(不考虑特殊 IP 地址以及知名端口套接字)

（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 分）

答案：$2*2^{48}$

服务器端	客户端	结果
2 个独立固定 IP	**任意 32 位 IP** **任意 16 位 port number**	**$2*2^{32+16}$**

（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？（2 分）

答案：$2*2^{64}$

服务器端	客户端	结果
2 个独立固定 IP **任意 16 位 port number**	**任意 32 位 IP** **任意 16 位 port number**	**$2*2^{16+32+16}$**

【代理】

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 128.2.194.242，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？答：（　　）

	客户端请求	代理请求
A	(128.2.194.242:25, 155.232.108.39:80)	(128.2.194.242:51213, 208.216.181.15:80)
B	(128.2.194.242:51213, 155.232.108.39:80)	(128.2.194.242:12306, 208.216.181.15:80)
C	(128.2.194.242:25, 208.216.181.15:80)	(155.232.108.39:51213, 208.216.181.15:80)
D	(128.2.194.242:51213, 208.216.181.15:80)	(155.232.108.39:12306, 208.216.181.15:80)

答案：D

解释：考察所有三部分内容

（客户端请求）这一段，客户端发送的请求，源地址是客户自己地址 128.2.194.242，端口号是临时端口号；目标地址仍是服务器地址 208.216.181.15:80 默认端口 80

（代理请求)这一段,代理发送的请求,源地址是代理自己地址 155.232.108.39,端口号是临时端口号；目标地址仍是服务器地址 208.216.181.15:80 默认端口 80。

所以，选项 D 正确。

（选项 A/B 中，目标地址不是代理的地址，而是目标服务器的地址。）

（选项 C 中，端口号 25 属于电子邮件应用默认端口号，不作为客户端的临时端口号。）