

## “ 计算机系统基础 ” 模拟试卷

### 一、 计算机工作基本原理填空

1、在对某程序进行调试时，看到如下一段信息：

```
004116DA  83 7D FC 00          cmpl  .....
004116DE  75 08                jne   004116E8
004116E0  8B 45 FC             mov   .....
004116E3  89 45 F8             mov   .....
004116E6  EB 07                jmp   004116EF
004116E8  C7 45 F8 01 00 00 00 movl   .....
004116EF  E8 2D FB FF FF       callq func (00411221) ; func 为函数名
004116F4  C7 45 F4 21 12 41 00 movl   $0x411221, -0x0c(%ebp)
004116FB  FF 55 F4             callq -0x0c(%ebp)
.....
```

① 观察指令在内存中的存放形式(每个空对应一个 16 进制字节数据)：

004116DA    \_\_\_\_\_  
004116E2    .....

② 设当前 `eip` 为 `0x004116DE`，在取出 `eip` 指向的指令并进行译码后，`eip` = \_\_\_\_\_；

③ 在 `0x004116DE` 处的指令为：`75 08 jne 004116E8`，直观上是在 `zf` = \_\_\_\_\_ 时，会将 `0x004116E8`  $\rightarrow$  `eip`，计算出该地址值的方法 \_\_\_\_\_；当转移条件不成立时，则该指令执行完成，`eip` \_\_\_\_\_（会、不会）改变。

④ `004116EF` 处的指令中有 `0xFFFFFB2D`，计算出该值的方法是 \_\_\_\_\_；

⑤ `004116FB` 处的子程序调用指令，调用的子程序的入口地址是 \_\_\_\_\_，执行该处语句时，CPU 会将 `0x`\_\_\_\_\_ 压入堆栈中。

⑥ 执行子程序中的 `RET` 指令时，CPU 会 \_\_\_\_\_  $\rightarrow$  `eip`。

2、设一个 C 语言程序中定义有全局数组 `int array[5]`；`array` 的起始地址（即 `&array[0]`）为 `0x420100`，每个元素占 4 个字节，现要将数组的第 2 个元素(即 `array[2]`)置为 20。按指定的寻址

解  
答  
内  
容  
不  
得  
超  
过  
装  
订  
线

方式写出实现该功能的机器指令段（汇编语句段）。

注：完成该功能的 C 语句有：“array[2]=20; \*(array +2) =20; int i=2; array[i]=20; int \*p=array; p[i] = 20;”等等。不同的 C 语句写法编译后，生成的机器指令不同，会出现多种寻址方式访问同一单元。

- ① 使用直接寻址方式，机器指令中含有操作数的偏移地址：

\_\_\_\_\_

- ② 使用寄存器间接寻址方式，先将操作数的地址送 ebx，再访存：

\_\_\_\_\_； 给 ebx 赋值

\_\_\_\_\_

- ③ 使用变址寻址方式，先将元素的下标（即 2）送 eax，再访存：

\_\_\_\_\_； 给 eax 赋值

\_\_\_\_\_

- ④ 使用基址加变址寻址方式，先将数组的起始地址送 eax，第 2 个元素在数组中的偏移字节数送 ebx，再访存：

\_\_\_\_\_； 给 eax 赋值

\_\_\_\_\_； 给 ebx 赋值

\_\_\_\_\_

- ⑤ 对于全局变量，其空间分配在\_\_\_\_\_。而对于非静态的局部变量，其空间分配在\_\_\_\_\_，对应的地址表达方式一般为 disp(%ebp)、disp(%esp) 等。

## 二、 数据存储及 C 语句转换填空

在Linux环境下，对一个C语言程序进行编译、链接、调试运行，程序片段如下。

```
int fadd(int a, int b)
```

```
{ int temp;
```

```
temp = a + b;
```

```
return temp;
```

```
}
```

```
void main( )
```

```
{
```

```
int x = 0x1234;

int y = -32;

int result = 0;

char msg[6] = "abc12"; // '1'的ASCII是 0x31, 'a'的ASCII是 0x61

result = fadd(x, y);

result = *(int *) (msg+1);

}
```

调试时，设变量  $x$  的地址（即 $\&x$ ）为  $0xffffd508$ ； $y$  的地址（即 $\&y$ ）为  $0xffffd50c$ ， $result$  的地址为  $0xffffd510$ ，数组  $msg$  的起始地址为  $0xffffb516$ 。

- 1、执行到“ $result = fadd(x, y);$ ”时，以字节为单位观察内存内容（用16进制数的形式填空，最左边是内存地址）。

$0xffffd508$	_____	_____	_____	_____	_____	_____	_____
$0xffffd510$	_____	_____	_____	_____	XX	XX	_____
$0xffffd518$	_____	_____	_____	_____	XX	XX	XX XX

- 2、数据传送指令解读

“ $int x = 0x1234;$ ”对应的机器指令为： $movl \ \$0x1234, -0x20(\%ebp)$ ，执行该语句时， $ebp = 0x$ \_\_\_\_\_。

“ $int result = 0;$ ”对应的反汇编指令为\_\_\_\_\_。

执行“ $result = *(int *) (msg+1);$ ”后， $result$  中的值为  $0x$ \_\_\_\_\_。

- 3、函数调用语句解读

语句“ $result = fadd(x, y);$ ”对应的反汇编代码（最左边的是机器指令的地址）如下。

$0x56556210 <+72>:$	$push \ -0x1c(\%ebp)$		$0xffffd4ec$
$0x56556213 <+75>:$	$push \ -0x20(\%ebp)$		$0xffffd4f0$
$0x56556216 <+78>:$	$call \ 0x5655619d <fadd>$		$0xffffd4f4$
$0x5655621b <+83>:$	$add \ \$0x8, \ \%esp$		$0xffffd4f8$
$0x5655621e <+86>:$	$mov \ \%eax, -0x18(\%ebp)$		$0xffffd4fc$
		XXXXXXXX	$0xffffd500$

设执行“ $result = fadd(x, y);$ ”之前， $esp$  的值为  $0xffffd500$ 。

- ① 在表格的适当位置填写刚进入函数  $fadd$  内部时，堆栈中存放的相关数据（6分）；
- ② 刚进入函数  $fadd$  内部时， $esp = 0x$ \_\_\_\_\_；
- ③ 执行“ $add \$0x8, \%esp$ ”之后， $esp = 0x$ \_\_\_\_\_。

#### 4. 函数 fadd 的指令解读

函数体对应的反汇编代码有：

```
0x5655619d <+0>:    push  %ebp
0x5655619e <+1>:    mov   %esp, %ebp
0x565561a0 <+3>:    sub   $0x10, %esp
0x565561a3 <+6>:    mov   0x8(%ebp), %edx
0x565561a6 <+9>:    mov   0xc(%ebp), %eax
0x565561a9 <+12>:   add   %edx, %eax
0x565561ab <+14>:   mov   %eax, -0x4(%ebp)
0x565561ae <+17>:   mov   -0x4(%ebp), %eax
0x565561b1 <+20>:   .....
```

- ① 函数参数 a 的地址（即&a）是 0x\_\_\_\_\_。
- ② 局部变量 temp 的地址（即&temp）是 0x\_\_\_\_\_。
- ③ 执行 “add %edx,%eax” 后， CF=\_\_\_\_, SF=\_\_\_\_, ZF=\_\_\_\_, OF=\_\_\_\_\_。
- ④ 在函数的结束处，有程序段

\_\_\_\_\_

\_\_\_\_\_

ret

执行后可返回到调用函数的断点处。

### 三、 程序阅读与理解

#### 1、阅读下面的程序，回答问题 。

```
void func ()
{
    unsigned short us = 65535;
    unsigned int ui;
    short s = -1;
    int i;
    int x = 0, y = 0;           ; ①
    ui = us;
    i = s;                       ; ②
    if (ui > 0) x = 1;
    if (i > 0) y = 1;           ; ③
    ui = ui >> 1;                ; >>1 右移一个二进制位
    i = i >> 1;                  ; ④
    ui = ~ui;                    ; 按位取反
    i = -i;                      ; ⑤
}
```

}

(1) 执行完 ① 处语句后,

us = 0x \_\_\_\_\_ s = 0x \_\_\_\_\_ (2 个字节的 16 进制数)

(2) 执行完 ② 处语句后,

ui = 0x \_\_\_\_\_ i = 0x \_\_\_\_\_ (4 个字节的 16 进制数)

ui=us; 对应的机器指令: \_\_\_\_\_ us, %eax 、 mov %eax, ui

i=s; 对应的机器指令: \_\_\_\_\_ s, %eax 、 mov %eax, i

(3) 执行完 ③ 处语句后,

x = \_\_\_\_\_ y = \_\_\_\_\_

if (ui>0) x=1; 对应的机器指令: cmp \$0, ui、 \_\_\_\_\_ lp1、 mov \$1, x、 lp1:...

if (i >0) y=1; 对应的机器指令: cmp \$0, i、 \_\_\_\_\_ lp2、 mov \$1, y、 lp2:...

(4) 执行完 ④ 处语句后,

ui = 0x \_\_\_\_\_ i = 0x \_\_\_\_\_ (4 个字节的 16 进制数)

ui = ui >> 1; 对应的机器指令: \_\_\_\_\_ \$1, ui

i = i >> 1; 对应的机器指令: \_\_\_\_\_ \$1, i

(5) 执行完 ⑤ 处语句后,

ui = 0x \_\_\_\_\_ i = 0x \_\_\_\_\_ (4 个字节的 16 进制数)

ui = ~ui; 对应的机器指令: \_\_\_\_\_ ui

i = -i; 对应的机器指令: \_\_\_\_\_ i

2、调试一个 C 程序时, 在函数的反汇编中看到源程序语句与对应的反汇编语句如下。根据观察到的信息填空。

int a[2][5];

int i = 1;

movl \$1, -0x2c(%ebp)

i 的地址为 0x0019feb8, ebp = 0x \_\_\_\_\_

int j = 3;

movl \$3, -0x30(%ebp) &j = 0x \_\_\_\_\_

a[i][j] = 10;

imul \$0x14, -0x2c(%ebp), %eax

执行后, eax = \_\_\_\_\_

该值的含义是 \_\_\_\_\_

lea -0x28(%ebp, %eax, 1), %ecx

ecx 中存放的值的含义是 \_\_\_\_\_

mov -0x30(%ebp), %edx

movl \$0xa, (%ecx, %edx, 4)

指令中的 `edx` 乘 4 的原因是\_\_\_\_\_

`a[0]` 的地址表达形式是 \_\_\_\_\_;

`global[i] = 18;`      `global` 为一个全局整型数组

```
mov    -0x2c(%ebp), %eax
```

```
movl    $0x12, 0x417120(,%eax, 4)
```

`global` 数组的起始地址是\_\_\_\_\_

```
int* p = &global[1];
```

```
mov     $4, %eax
```

```
shl     $0, %eax
```

```
add     $0x417120, %eax
```

```
mov     %eax, -0x34(%ebp)
```

变量 `p` 中的值是 `0x`\_\_\_\_\_

```
*p = 19;
```

```
mov     -0x34(%ebp), %eax
```

```
movl     $0x13, (%eax)
```

第二条汇编语句访存的寻址方式是\_\_\_\_\_

3、 以下是结构 `test` 的声明。假设在 32 位 Windows 平台上编译（采用缺省的自然对齐方式），问结构成员 `d` 和 `v` 的偏移量是多少？ 结构体所占存储空间是多少字节？ 如何调整成员的先后顺序使得结构所占存储空间最小？

```
struct{
    char c;
    int i;
    double d;
    short s;
    double *v;
    long l;
} test;
```

4、 阅读下面的程序，回答问题 。

```
.section .data
    array: .long 10, -20, 30, -40, 50
    length = (. - array)/4          # length 为array中元数的个数, = 5
    format: .ascii "%d\n\0"
.section .text
.global _start
_start:
    mov     $0, %eax
```

```

        mov $length, %ecx
        lea array, %edi      # ①
lp_1:
        cml $0, (%edi)
        jl  lp_2            # ②
        inc %eax
lp_2:
        add $4, %edi
        sub $1, %ecx        # ③
        jne lp_1
        push %eax
        push $format
        call printf
        mov $1, %eax      # 程序正常退出
        mov $0, %ebx
        int $0x80

```

- 1) 上述程序的功能是什么？运行后，屏幕上显示的是什么？
  - 2) 若标号 lp\_1 写到 ①处语句前，程序运行的结果是什么？为什么？
  - 3) 若将 ② 处的语句改为 “jb lp\_2”，程序运行的结果是什么？
  - 4) 若漏写了 ③ 处的语句，程序运行会出现什么现象？为什么？
- 5、已知 **function\_test** 的 C 语言代码框架如下，根据对应的汇编代码填写 C 代码中缺失部分。

```

int function_test( unsigned x)    // unsigned int
{
    int result=0;
    int i;
    for ( _____ ① _____ ; _____ ② _____ ; _____ ③ _____ ) {
        _____ ④ _____ ;
        _____ ⑤ _____ ;
    }
    return result;
}

```

上述函数过程体对应的汇编代码如下：

```

1  movl    8(%ebp), %ebx
2  movl    $0, %eax
3  movl    $0, %ecx
4  .L12:

```

```

5  leal    (%eax,%eax), %edx
6  movl    %ebx, %eax
7  andl    $3, %eax
8  orl     %edx, %eax
9  shrl    %ebx          // shr $1, %ebx
10 addl    $1, %ecx
11 cmpl    $64, %ecx
12 jne     .L12
    .....
    ret

```

参见教材 P138 -P139 例 3-13 的分析

#### 四、 程序优化

- 1、 举例说明编写 C 程序或者编译器优化时，利用 CPU 特性的做法（包括优化前的方法，优化后的方法）。
  - ① 提高 CPU 中 cache 的命中率
  - ② 提高 CPU 中指令流水线的利用率
  - ③ 使用 CPU 中处理速度更快的指令
  - ④ 使用 CPU 中单指令多数据流指令或串操作指令
  - ⑤ 使用多核 CPU 中多线程处理能力
- 2、 在编写程序（也包括编译器）可以做哪些 与 CPU 无关的优化工作？（给出 5 种不同类型的示例）。
- 3、 为了提高程序运行的安全性，编译器有多种编译开关（选项），增强发现程序漏洞的能力。试举出二种编译开关的例子，并说明增强安全性的实现原理。【堆栈帧检查、未初始化变量使用检查】

#### 五、 链接和异常控制流问答

- 1、 设一个函数中有语句 `int temp=global;` 其中 `global` 是一个初值为 35 的 `int` 类型全局变量。编译器对 `global` 的定义(`int global=35;`)和 `temp=global` 编译时，分别会在可重定位目标文件中的哪些节生成哪些信息？
- 2、 可重定位目标文件中有哪些节？各节中主要有什么信息？什么是符号解析？链接的过程



是什么？

- 3、什么是中断和异常？两者有何差别？什么是中断描述符表？中断和异常的响应过程是什么？
- 4、在一个程序的运行过程中（如正在执行一个二维数组求累加和），用户按了键盘上的某个键，计算机系统会做出哪些响应（即一系列的处理过程）？中断分哪几类？请举例说明各类中断在何种情况下产生。

六、已知以下关于 Lab3 Bang 阶段的信息，请完成填空，注意涉及数值全部使用 16 进制。

```
08048e6d <test>:
8048e6d: 55                push    %ebp
8048e6e: 89 e5            mov     %esp, %ebp
8048e70: 53              push    %ebx
8048e71: 83 ec 24        sub     $0x24, %esp
8048e74: e8 6e ff ff ff  call    8048de7 <uniqueval>
8048e79: 89 45 f4        mov     %eax, -0xc(%ebp)
8048e7c: e8 6b 03 00 00  call    80491ec <getbuf>
8048e81: 89 c3          mov     %eax, %ebx
8048e83: e8 5f ff ff ff  call    8048de7 <uniqueval>
.....
```

```
080491ec <getbuf>:
80491ec: 55                push    %ebp
80491ed: 89 e5            mov     %esp, %ebp
80491ef: 83 ec 38        sub     $0x38, %esp
80491f2: 8d 45 d8        lea     -0x28(%ebp), %eax
80491f5: 89 04 24        mov     %eax, (%esp)
80491f8: e8 55 fb ff ff  call    8048d52 <Gets>
80491fd: b8 01 00 00 00  mov     $0x1, %eax
8049202: c9              leave   %ebp
8049203: c3              ret
```

```

08048d05 <bang>:
8048d05: 55                push    %ebp
8048d06: 89 e5             mov     %esp,%ebp
8048d08: 83 ec 18          sub     $0x18,%esp
8048d0b: a1 18 c2 04 08    mov     0x804c218,%eax
8048d10: 3b 05 20 c2 04 08 cmp     0x804c220,%eax
8048d16: 75 1e             jne     8048d36 <bang+0x31>
8048d18: 89 44 24 04        mov     %eax,0x4(%esp)
8048d1c: c7 04 24 e4 a2 04 08 movl    $0x804a2e4, (%esp)
8048d23: e8 a8 fb ff ff    call    80488d0 <printf@plt>
.....

```

```

Breakpoint 2, 0x080491f2 in getbuf ()
(gdb) info r
eax            0x6f50c1c5    1867563461
ecx            0xf7fbd068    -134492056
edx            0xf7fbd3cc    -134491188
ebx            0x0      0
esp            0x55683458    0x55683458 <_reserved+1037400>
ebp            0x55683490    0x55683490 <_reserved+1037456>
esi            0x55686018    1432903704
edi            0x1      1
eip            0x80491f2    0x80491f2 <getbuf+6>
.....

```

```

(gdb) x 0x804c218
0x804c218 <global_value>: 0x00000000

```

```

acd@ubuntu:~/Lab3$ ./makecookie U201414XXX
0x250d3ee8

```

```

int global_value = 0;

void bang(int val)
{
    if (global_value == cookie) {
        printf("Aha Bang!: You set global_value to 0x%x.\n", global_value);
        validate(2);
    } else
        printf("Oh Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}

```

1) 调用 getbuf 后的返回地址是: (1) \_\_\_\_\_

- 2) 调用 getbuf 时, 存放 getbuf 返回地址的单元是: (2) \_\_\_\_\_
- 3) 调用 getbuf 时, 执行到断点 2 时, getbuf 的栈帧栈顶地址是: (3) \_\_\_\_\_
- 4) getbuf 中缓冲区 buf 的起始地址是: (4) \_\_\_\_\_
- 5) 攻击字符串的长度为: (5) \_\_\_\_\_ 字节
- 6) Cookie 值 0x250d3ee8 所在的存储单元地址是: (6) \_\_\_\_\_
- 7) 以下是所设计的一个带有注释的攻击字符串文件的内容, 请填空:
- ```

c3 c3 c3 c3
b8 (7) _____ /* mov (8) _____, %eax */
a3 18 c2 04 08 /* mov %eax, (9) _____ */
68 05 8d 04 08 /* push (10) _____ */
c3 /* ret */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
(11) _____ (12) _____ (13) _____ 55

```
- 8) 上述攻击字符串可使 bang 输出: (14) \_\_\_\_\_

解答内容不得超过装订线

- 七、假设一个 c 语言程序有两个模块: main.c 和 swap.c, 对它们单独编译, 分别生成可重定位目标文件 main.o 和 swap.o。 main.c 和 swap.c 代码如下:

main.c

```

int buf[2] = {1, -4};
extern void swap( );
int sum=0;
int main( )
{
    swap( );
    return 0;
}
int ave( ) {
    int a;
    static int count=2;
    sum=buf[0]+buf[1];
    a=sum/count;
    return a;
}

```

swap.c

```

extern int buf[];
extern int ave();
int *bufp0 = &buf[0];
static int *bufp1;
int sum;
void swap( ) {
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
    ave( );
}

```

请指出 main.o 和 swap.o 中所有强符号和弱符号分别有哪些?

请指出 swap.o 的符号表中分别有哪些符号？

请指出 swap.o 的符号表中各符号分别是什么类型的符号（全局符号，本地符号或外部符号）？

请指出 swap.o 的符号表中各符号分别出现在 swap.o 中的哪个节（.text, .data 或 .bss）？

(2) 假定 swap.o 的反汇编代码如下：

```
00000000 <swap>:
 0: 55                push    %ebp
 1: 89 e5             mov     %esp, %ebp
 3: 83 ec 18          sub     $0x18, %esp
 6: c7 05 00 00 00 04 movl    $0x4, 0x0      # 重定位位置①
 d: 00 00 00
    8: R_386_32 .bss
    c: R_386_32 buf
10: a1 00 00 00 00    mov     0x0, %eax      # 重定位位置②
    11: R_386_32 bufp0
15: 8b 00             mov     (%eax), %eax
17: 89 45 f4          mov     %eax, -0xc(%ebp)
1a: a1 00 00 00 00    mov     0x0, %eax      # 重定位位置③
    1b: R_386_32 bufp0
1f: 8b 15 00 00 00 00 mov     0x0, %edx
    21: R_386_32 .bss      # 重定位位置④
25: 8b 12             mov     (%edx), %edx
27: 89 10             mov     %edx, (%eax)
29: a1 00 00 00 00    mov     0x0, %eax      # 重定位位置⑤
    2a: R_386_32 .bss
2e: 8b 55 f4          mov     -0xc(%ebp), %edx
31: 89 10             mov     %edx, (%eax)
33: e8 fc ff ff ff    call    34 <swap+0x34> # 重定位位置⑥
    34: _____ ⑥
38: 90                nop
39: c9                leave
3a: c3                ret
```

假定可执行目标文件里 swap 函数代码的起始地址是 0x00808100，swap 函数代码紧接在 ave 函数代码的后面，且 ave 函数代码占 0x2e 个字节。对 swap.o 中的重定位位置⑥进行重定位。回答：此处待重定位的符号是什么？重定位类型是什么？重定位前的值是什么？该值的含义是什么？重定位后的值是什么（需要给出计算过程）？重定位后最终指向的虚拟地址是多少？