清华大学本科生考试试题专用纸 A

考试课程“计算机系统概论”　　2022 年 12 月 31 日

1.（X、Y 的数据位宽均为 16 位，计算结果用 16 进制的补码表示）已知$[X]_{补码}$＝0x0431，$[Y]_{补码}$＝0xC150，则$[X+Y]_{补码}$＝（　　①　　），$[X-Y]_{补码}$＝（　　②　　）。

2. 在所有由 1 个“1”和 7 个“0”组成的 8 位二进制整数中，最小的带符号数是（　①　　），最大的无符号数为（　②　　）；请用十进制表示结果。

3. 假设存在一种 9 位浮点数（符合 IEEE 浮点数标准），1 个符号位，4 位阶码，4 位尾数。其数值被表示为 $V = (-1)^S \times M \times 2^E$ 形式。请在下表中填空(每空 1 分)。

Binary: 这一列请填入 **9 位二进制表示**；M：**十进制数表示**；E：**十进制整数表示**；

Value: 被表示的具体数值，**十进制表示**；“—”表示无需填入。

描述	Binary	M	E	Value
7.5	①	②	③	④
最大的非规格化浮点数	⑤	⑥	⑦	⑧

以上每个填空 1 分。

4. 有如下对应的 C 代码与汇编代码（x86-64），请对照着填上代码中缺失的部分（数字请用十进制表示，共 8 分）。

```
struct matrix_entry{
   char a;
   char b;
   double d;
   short c;
};
struct matrix_entry matrix[5][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
            ②         0(,%rcx,8), %rax
            movswl   matrix+ ③ (%rdx,%rax), %eax
            ret

            .comm     matrix, 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
```

5. 已知三个二维矩阵的定义如下，并已初始化（12 分）。

```
#define n 20
int a[n][n] ;
int b[n][n] ;
int c[n][n] ;
```

需要进行如下的矩阵函数操作。

```
void column()
{
    int j, k , i;
    int r;
    for (j=0; j<n; j++) {
        for (k=0; k<n; k++) {
            r = b[k][j];
            for (i=0; i<n; i++)
                c[i][j] += a[k][i] * r;
        }
    }
}
```

编译完的汇编代码（x86-64）如右下示（先左后右），请填出代码中的空缺部分。

```
column:
        pushq   %rbx
        movl    $0, %r9d
        jmp     ①
.L5:
        movslq  %r9d, %rax
        ②       %r8d, %rdx
        leaq    (%rdx, ③,  ④), %rcx
        leaq    0(,%rcx,4), %rdx
        addq    %rdx, %rax
        movl    b(,%rax, ⑤), %r10d
        movl    $0, %esi
        jmp     ⑥
.L4:
        movslq  %esi, %rax
        movslq  %r8d, %rdx
        leaq    (%rdx,%rdx,4), %rcx
        leaq    0(,%rcx,4), %rdx
        addq    %rax, %rdx
        movl    ⑦, %edi
        imull   a(,%rdx,4), %edi
        movslq  %r9d, %rdx
        leaq    0(,%rax,4), %r11
        leaq    (%r11,%rax), %rbx
        leaq    0(,%rbx,4), %rcx
        addq    ⑧, %rcx
        movl    %edi, %eax
        addl    c(,%rcx,4), %eax
        movl    %eax, c(,%rcx,4)
        addl    $1, %esi
.L3:
        cmpl    $19, %esi
        jle     ⑨
        addl    $1, %r8d
        jmp     ⑩
.L7:
        movl    $0, %r8d
.L6:
        cmpl    $19, %r8d
        jle     ⑪
        addl    $1, %r9d
.L2:
        cmpl    $19, %r9d
        jle     ⑫
        popq    %rbx
        ret
```

6.（8 分）假设文件 file1.txt 有下列内容：aabbccdd。下列 C 文件分别被编译成./program1 和./program2。其中 open(*filename*, O_RDWR)表示打开一个已有文件 *filename*，对其进行读写操作，起始位置为 0；execl (*exename*, …)调用则是执行 *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？

```
/*
 *  Program1
 */
#include <unistd.h>
#include <fcntl.h>

int main()
{
  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);
}
```

```
/*
 *  Program2
 */
#include <unistd.h>
#include <fcntl.h>

int main()
{
  char buf[2];

  read(STDIN_FILENO, buf, 2);
  write(STDOUT_FILENO, buf, 2);
}
```

7. （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。请根据上述事实以及 C 函数语义，填写下面用带圈数字表示的空白或回答问题。

```
switch_eg:
        cmpq    $6, %rdi
        movq    %rdx, %rcx
        ja      .L8             #①
        leaq    ②(%rip), ③     #
        movslq  (④, ⑤, 4), %rdi
        ⑥       %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                   #正文段
.L9:
        movl    ⑦, %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
```

```
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;
}
```

（1） 请问为何①处用的是 ja 指令来进行带符号数的条件判断？（2 分）

（2） 填写带圈数字表示的空白处以补齐指令（6 分）：

②_____ ③_____ ④_____ ⑤_____

⑥_____⑦_____

8. 下面是一段 C 代码以及对应的 x86-64 汇编(8 分)：

```
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
```

回答以下问题：

a. gets 的参数通过寄存器 __①__ 传递，寄存器 %rbx 属于 __②__ (caller saved/callee saved) 寄存器。以 subq 指令后的 %rsp 计算，buf 数组的基地址是 %rsp + ③ ，%rbx 寄存器保存在 $rsp + __④__ 。

b. 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 NUL 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 NUL）需要满足的要求，以区间表示：

a. 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, __⑤]

b. 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 [⑥, ⑦]

c. 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 [⑧ , +inf)

9. 考虑如下程序(4 分)。

```
int counter = 0;
int pid, status;

void handler1(int sig) {
    counter += 1;
    printf("%d", counter);
    int ppid = getppid(); // get pid of parent
    kill(ppid, SIGUSR2);
}

void handler2(int sig) {
    counter += 7;
    printf("%d", counter);
}

int main(){
```

```
    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);
    if (pid = fork()) {
        kill(pid, SIGUSR1);
    } else {
        printf("%d", counter);
    }
    return 0;
}
```

假设所有信号量 handler 内的函数都是 Async-Signal-Safe 的，且 printf 的输出即时显示在 stdout 上，以下哪些是可能的输出结果（多选）：
A.1　　B.01　　C.11　　D.017　　E.018　　F.171　　G.1187

10. 考虑如下程序(3 分)。

```
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);
}
```

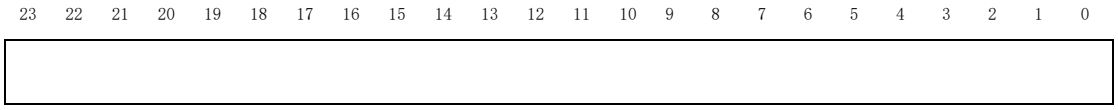
以下哪些是可能的输出结果（多选）：
A. ABCBE7E　　B. ABD7E　　C. ABBCE4E　　D. ABCDB4E　　E. ABBD4E

11. （9 分）关于虚拟地址到物理地址转换：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：

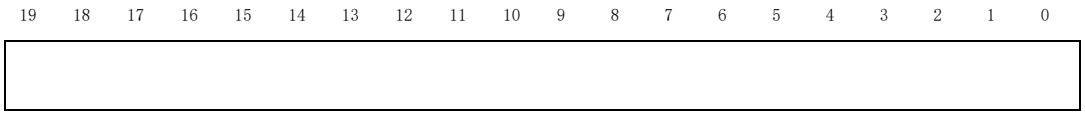
TLB			
Index	Tag	PPN	Valid
0	011	1C	1
	02C	B3	1
1	13C	A4	0
	0B2	7E	1
2	001	05	1
	1A3	B6	0
3	002	08	1
	003	17	1
4	1C2	21	1
	013	09	0
5	1CF	38	1
	08B	51	1
6	003	7A	1
	13C	7F	1
7	031	B2	0
	0A4	3C	1

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

（1）下面的框显示了虚拟地址的格式。指出(通过在图上标注)字段(如果存在)，这些字段将用于确定以下内容(如果一个字段不存在，就不要在上面绘制)：VPO / VPN / TLBI / TLBT



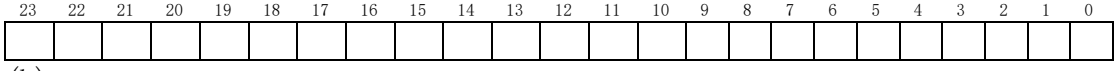
类似的，在下图标注出物理地址的格式：PPO/PPN



（2）对于给定的两个虚拟地址 0x01DBE3、0x9E6CF2，请分别表示出相应的 TLB 表项和物理地 址，并指出 TLB 是否命中、是否发生 page fault(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”中输入“-”。 另，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 10^8ns(已经包含更新 TLB 和页表的时间)

- 0x01DBE3

（a）



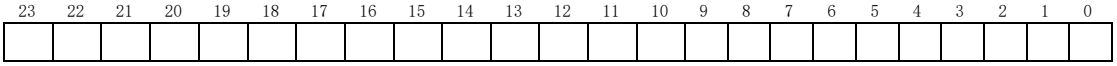
(b)

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	

Page Fault? (Y/N)	
PPN	
获得该地址所存储数据的访问时间	

- 0x9E6CF2

（a）



（b）

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	
获得该地址所存储数据的访问时间	

12. （8 分）代码注入攻击，是在缓冲区溢出的基础上，向缓冲区注入攻击指令，通过修改函数的返回地址，实现代码的任意执行，步骤如下：
    a. 攻击者找到可以缓冲区溢出攻击的函数，如：

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

    b. 攻击者构造一段输入，包括攻击指令（exploit code），以及保存了攻击指令的栈上地址。
    c. 程序运行到 Q 函数，读取攻击者构造的输入，此时函数的返回地址被覆盖为栈上的地址，栈则保存了攻击指令。
    d. Q 函数执行 ret 指令，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. 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：措施一：把栈标记为不可执行；措施二：在栈上保存金丝雀（Stack Canary）

3. Return Oriented Programming 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。它可以绕过上面二种措施中的 _①_，但是不可以绕过上面二种措施中的 _②_。

13. (10 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)___);
    }
}
```

14. （7 分）阅读程序写结果（需列出所有可能输出，不考虑进程/线程创建失败的情况，假设 printf 的输出不会被其他 printf 打断）。

(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;
}
```

(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;
}
```

(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;
}
```

(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;
}
```

15. （3 分）对于如下代码中 test_call 函数中的五次调用 test1~test5，其生成的.o 文件需要在链接期全局重定位的调用有哪些，即生成.o 文件中，需要重定位的符号有哪些？

```
//foo.c
extern void (*test1)();
static void test2() {

}

void test3() {
}

extern void test4();

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