

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”

2024 年 1 月 8 日

1. (2 分) 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 $[X]_{\text{补码}} = 0x07E1$, $[Y]_{\text{补码}} = 0xE200$, 则 $[X+Y]_{\text{补码}} = (0x\textcolor{red}{E9E1})$, $[X-Y]_{\text{补码}} = (0x\textcolor{red}{25E1})$ 。

2. (4 分) Windows 操作系统出现蓝屏是 同步 (同步/异步) 异常，页缺失 (page fault) 是 同步 (同步/异步) 异常，指令除以零操作引起的是 同步 (同步/异步) 异常，硬盘完成读写操作后打断 CPU 当前处理流程属于 异步 (同步/异步) 异常。

3. (2 分) 由 7 个 1, 9 个 0 组成的 16 位宽的带符号整数，其最小值是 -32705 (十进制表示)。

4. 假设存在一种 10 位浮点数 (符合 IEEE 浮点数标准), 1 个符号位, 5 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 10 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, 十进制或分数表示; “—”表示无需填入。

描述	Binary	M	E	Value
0.16	①	②	③	④
最大的非规格化数	⑤	⑥	⑦	⑧

5. 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数字 (请用十进制表示); 并回答一个类型为 typeTAG 的结构体所占据的空间大小是多少字节 (共 8 分)。**M=11 (3 分)、N=32 (3 分); 16 字节 (2 分)**

copy_element:

```

movslq    %esi, %rsi
movslq    %edi, %rdx
movq      %rdx, %rax
salq      $5, %rax
addq      %rsi, %rax
salq      $4, %rax
leaq      (%rsi,%rsi,4), %rcx
leaq      (%rsi,%rcx,2), %rdi
addq      %rdx, %rdi
salq      $4, %rdi
leaq      mat2(%rip), %rdx
movq      (%rdx,%rdi), %rsi
movq      8(%rdx,%rdi), %rdi
leaq      mat1(%rip), %rdx
movq      %rsi, (%rdx,%rax)
movq      %rdi, 8(%rdx,%rax)
movl      $0, %eax
ret

```

```

#define M ①
#define N ②
typedef struct typeTAG
{
    double  attribute_3;
    short   attribute_2;
    int     attribute_1;
} TAG;

TAG mat1[M][N];
TAG mat2[N][M];
int copy_element(int i, int j)
{
    mat1[i][j] = mat2[j][i];
    return 0;
}

```

6. 请看如下的 C 代码（有些代码被省略掉了，只给出了功能说明），并回答问题（8 分）。

（1）`c_v_a` 与 `p_v_a` 的关系，以及 `c_p_a` 与 `p_p_a` 的关系（相等 或者 不等）。均为相等（1-4 问各 2 分）

（2）如果将代码中间部分的注释符号去掉（即使得被注释掉的代码发生作用），请回答（1）。相等；不等

（3）请回答为何会产生（1）、（2）这样的现象。简要阐述 COW 的机理即可

（4）变量 `s` 自身存储的虚拟地址（不是其指向的地址）与 `c_v_a` 的关系（大于 或者 小于）。大于（因为存储在栈上）

```
int main()
{
    uint64_t phy = 0;
    char* s=(char*)malloc(128);
    pid_t pid=fork();
    /*
    if(pid==0)
    {
        strcpy(s,"hello");
    }
    else
    {
        strcpy(s,"world");
    }
    */
    if(pid==0)
    {
        //输出 s 指向的虚拟地址 c_v_a 与物理地址 c_p_a
        ...
    }
    else
    {
        //输出 s 指向的虚拟地址 p_v_a 与物理地址 p_p_a
        ...
    }
}
```

7. 请看如下的 C 代码，它被称为“Duff's device”(1983 年)，是提高执行效率的一种代码技巧，这儿做了简化修改。代码看起来不太寻常，所以要精确得到它的语义，可以看看右侧所示的函数 `func` 的汇编代码（共 9 分）。

(1) C 代码的输出是什么？如果 `count` 初值为 18，那么输出是多少？

(2) 补齐右侧代码的空格。（每格 1 分）

(1) 10 (2 分) ; 18 (2 分)

```
int func(int n, int m)
{
    int i = 0;
    switch (m) {
        case 0 :    do {    i++;
        case 5 :        i++;
        case 4 :        i++;
        case 3 :        i++;
        case 2 :        i++;
        case 1 :        i++;
                    } while ( -- n > 0 );
    }
    return i;
}

int main()
{
    int count = 10;
    int res = func((count+5)/6, count%6);
    printf("%d\n", res);
    return 0;
}
```

```
func:
    cmpl $5, %esi
    ja .L10
    movl %esi, %edx
    leaq ① (%rip), %rcx .L4
    movslq (%rcx,%rdx,②), %rax 4
    addq ③, ④ %rcx, %rax
    jmp *%rax #用%rax的值作为直接跳转目标
    .section .rodata
.L4:
    .long .L11-.L4
    .long .L12-.L4
    .long .L13-.L4
    .long .L14-.L4
    .long .L15-.L4
    .long .L16-.L4
    .text
.L11:
    movl %esi, %eax
    jmp .L3
.L16:
    movl $0, %eax
.L9:
    addl $1, %eax
.L8:
    addl $1, %eax
.L7:
    addl $1, %eax
.L6:
    addl $1, %eax
.L5:
    addl $1, %eax
    subl $1, %edi
    testl %edi, %edi
    jle .L17
.L3:
    addl $1, %eax
    jmp .L9
.L15:
    movl $0, %eax
    jmp .L8
.L14:
    movl $0, %eax
    jmp .L7
.L13:
    movl $0, %eax
    jmp .L6
.L12:
    movl $0, %eax
    jmp .L5
.L17:
    ⑤ ret
.L10:
    movl $0, %eax
    ret
```

8、右侧的 C 代码被编译链接为一个动态库（.so 结尾），通过对该动态库进行反汇编（objdump）得到左侧的对应汇编语言代码（仅展示相关部分），与 C 程序全局变量相关的 Global Offset Table 的地址用 .got 表示。请回答如下问题（9 分）：

（1）以标有下划线的一行为例，请问 %rax 中存储的值是什么？**变量 c 的地址；**

（2）为何没有生成给 a, b 赋值的代码（提示：这个共享库只有这一个模块）？

因为两者为静态变量，无法被外部模块直接引用，而只有这一个模块内也没有其他语句引用它们，故而这一赋值无意义。

（3）将 GOT 看作是一个以 GOT 为起始地址的数组，每项的大小为 8 字节，那么请写出 GOT[0]、GOT[1]直至 GOT[8]这些项中每项存储的值是什么（如无意义，则直接写“无意义”）。

[0]f 的地址 [1]无意义 [2] g 的地址 [3] 无意义 [4] c 的地址 [5] 无意义 [6] d 的地址 [7]无意义 [8] e 的地址

000000000000067a <bar>:

```
67a: 48 8b 05 xx xx xx xx  mov    0x200957(%rip),%rax
681: 48 c7 00 03 00 00 00  movq   $0x3,(%rax)
688: 48 8b 05 xx xx xx xx  mov    0x200959(%rip),%rax
68f: 48 c7 00 04 00 00 00  movq   $0x4,(%rax)
696: 48 8b 05 xx xx xx xx  mov    0x200953(%rip),%rax
69d: 48 c7 00 05 00 00 00  movq   $0x5,(%rax)
6a4: 48 8b 05 xx xx xx xx  mov    0x20090d(%rip),%rax
6ab: 48 c7 00 06 00 00 00  movq   $0x6,(%rax)
6b2: 48 8b 05 xx xx xx xx  mov    0x20090f(%rip),%rax
6b9: 48 c7 00 07 00 00 00  movq   $0x7,(%rax)
6c0: c3                    retq
```

.....

00000000000200fb8 <.got>:

.....

```
static long a,b;
long c,d,e,f,g;
void bar()
{
    a = 1;
    b = 2;
    c = 3;
    d = 4;
    e = 5;
    f = 6;
    g = 7;
}
```

9. 下面是一段 C 代码以及对应的 x86-64 汇编 (gcc -O0) (10 分):

```

main:
401168 push %rbp
401169 mov %rsp,%rbp
40116c sub $0x30,%rsp
401170 movl $0x1,-0x4(%rbp)
401177 movl $0x2,-0x8(%rbp)
...
40119a movl $0x7,-0x1c(%rbp)
4011a1 mov -0x18(%rbp),%r9d
4011a5 mov -0x14(%rbp),%r8d
4011a9 mov -0x10(%rbp),①
4011ac mov -0xc(%rbp),%edx
4011af mov -0x8(%rbp),%esi
4011b2 mov -0x4(%rbp),%eax
4011b5 sub $0x8,%rsp
4011b9 mov -0x1c(%rbp),%edi
4011bc push %rdi
4011bd mov %eax,%edi
4011bf call foo
4011c4 add $0x10,%rsp
4011c8 mov %eax,-0x20(%rbp)
4011cb cmpl $0xabcd,-0x20(%rbp)
4011d2 jne 40120a <main+0xa2>
4011d4 mov $0x402004,%edi
4011d9 call puts
...
4011fc call foo
401201 add $0x10,%rsp
401205 mov %eax,-0x24(%rbp)
401208 jmp 401214
40120a mov $0x402011,%edi
40120f call puts
401214 mov $0x0,%eax
...

foo:
401136 push %rbp
401137 mov %rsp,%rbp
40113a sub $0x30,%rsp
...
401152 lea -0x8(%rbp),%rax
401156 mov %rax,%rdi
401159 mov $0x0,%eax
40115e call gets
401163 mov 0x10(%rbp),%eax
401166 leave
401167 ret

```

```

int foo(int a,int b,int c,int d,int e,int f,int g){
    char buf[8];
    gets(buf);
    //...
    return g;
}
int main(){
    int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    int ret = foo(a,b,c,d,e,f,g);

    if(ret == 0xABCD){
        printf(" ICS is my");
        int tmp = foo(a,b,c,d,e,f,g);
    }else{
        printf(" favourite course !\n");
    }
    return 0;
}

```

提示：反汇编中“ favourite course !\n”字符串的起始地址是 0x402011。

- (1) 若 foo 函数栈帧已经创建完成，请根据要求填空(return addr 和参数 g 均指 foo 函数内的): (4 分)

汇编代码中①	_____
&buf[0]	%rsp+_____
return addr	&buf[0]+_____
参数 g	&buf[0]+_____

- (2) 栈溢出攻击可以使代码违反原有执行顺序，如果想让程序输出“ ICS is my favourite course !”，且最后 ret 等于 0xABCD，两次 foo 函数调用中的 gets 分别应该输入什么字符串（请用 16 进制字节填充，暂不考虑 rbp 内容）(4 分)

第一次:

&buf[0]→							
&buf[0]+8→	old %rbp						
&buf[0]+0x10→							
&buf[0]+0x18→					-	-	-

第二次:

&buf[0]→							
&buf[0]+8→	old %rbp						
&buf[0]+0x10→							
&buf[0]+0x18→				-	-	-	-

(3) 现代多数编译器编译得到的 foo 函数结构如下图所示，请描述编译器插入代码段的作用，并解释生效原理？（2 分）

```
foo:
    pushq %rbx
    subq $16, %rsp
    movl $40, %ebx
    movq %fs:(%rbx), %rax
    movq %rax, 8(%rsp)
    xorl %eax, %eax
    movq %rsp, %rdi
    call gets
    movq 8(%rsp), %rax
    xorq %fs:(%rbx), %rax
    jne .L4
    movl 32(%rsp), %eax
    addq $16, %rsp
    popq %rbx
    ret
.L4:
    call __stack_chk_fail@PLT
```

(1) 每空一分，共 4 分

汇编代码中①	<u>%ecx</u>
&buf[0]	<u>%rsp + 0x28</u>
Return Addr	<u>&buf[0] + 0x10</u>
参数 g	<u>&buf[0] + 0x18</u>

(2) (1 个 2 分)

第一次，更改返回值 g，不更改返回地址

00 00 00 00 00 00 00 00

old %rbp

c4 11 40 00 00 00 00 00

CD AB 00 00 - - - -

第二次，更改返回地址，g 无所谓

00 00 00 00 00 00 00 00

old %rbp

0a 12 40 00 00 00 00 00

00 00 00 00 - - - -

(3) 金丝雀。在栈上保留标记，在返回前比较，防止栈溢出。(2 分)

10. 关于虚拟地址到物理地址转换：已知某系统内存是字节可寻址的（8 分）

- 每次内存访问针对的是 32-bit 的 word、物理地址 20 位、虚拟地址 26 位
- 页面大小为 4096 字节、每个页表项 4Byte
- TLB 是四路组相联，共有 16 个 TLB 项(即 4-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：

Page table base register

0x4B000

TLB			
Index	Tag	PPN	Valid
0	4C0	3B	1
	24A	26	0
	004	53	1
	030	A7	1
1	004	54	1
	02D	1D	1
	04E	74	1
	206	B4	1
2	0A4	3C	0
	005	4B	0
	3B4	63	1
	43C	7E	1
3	23B	4C	1
	578	74	0
	00F	55	1
	19A	2D	1

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	3C	1	10	53	1
01	2D	0	11	54	1
02	34	1	12	37	1
03	A3	1	13	42	0
04	2E	0	14	C9	1
05	54	1	15	A7	1
06	32	0	16	E6	1
07	26	1	17	F8	1
08	1A	1	18	9D	0
09	79	1	19	2C	1
0A	92	1	1A	59	1
0B	3F	0	1B	57	0
0C	B8	1	1C	64	1
0D	B9	1	1D	32	0
0E	2F	1	1E	7E	1
0F	A8	1	1F	7F	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 (全对得 1 分)

2524232221201918171615141312111098765432

类似的, 在下图标出物理地址的格式: PPO/PPN

191817161514131211109876543210

(2) 对于虚拟地址 0x001BDE4 (请注意前导 0), 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)。如果 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 10^8 ns(已经包含更新 TLB 和页表的时间) (每空 0.5 分, 共 4 分)

● 0x001BDE4

(a) 转换成二进制

2524232221201918171615141312111098765432

(b)

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

Page Fault? (Y/N)	
PPN	
总访问（拿到访存数据）时间	

(3) 对于虚拟地址 0x0015CB5，需要几次内存访问，依次写出每次访问的物理地址？
提示：page table base register 里存放的是物理地址。(3 分)

答案：

(1)

2 2 2 2 2 2 1 1 1 1 1 1 13 12 1 1 9 8 7 6 5 4 3 2 1 0
5 4 3 2 1 0 9 8 7 6 5 4 1 0

TLBT(12bit)	TLBI (2bit)	VPO (12bit)
-------------	-------------	-------------

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

PPN	PPO
-----	-----

(2)

● 0x001BDE4

(a)

24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0	1	1	1	1	0	0	1	0

(b)

Parameter	Value
VPN	0x001B
TLB Index	0x3
TLB Tag	0x006
TLB Hit? (Y/N)	N
Page Fault? (Y/N)	Y
PPN	—
访问时间	100000220 ns

(3) 两次 (1 分)。

VPN = 0x015

- TLB Miss 查询页表，页表内存地址 = Page table base register + VPN * 4Byte = 0x4B000 + 0x54 = 0x4B054，页表 Hit (1 分)
- Physical Addr = 0x A7CB5 (1 分)

11. 多线程题。

(1) 请阅读程序，选择可能的输出 (3 分)

<pre>volatile sig_atomic_t sigusr1_count = 0; volatile sig_atomic_t sigusr2_count = 0; void handle_sigusr1(int sig) { sigusr1_count++; printf("A"); } void handle_sigusr2(int sig) { printf("B"); } void child1_action(pid_t parent_pid) { printf("C"); kill(parent_pid, SIGUSR1); } void child2_action(pid_t parent_pid) { printf("D"); while (sigusr1_count < 2) sleep(1); kill(parent_pid, SIGUSR2); } void child3_action(pid_t parent_pid) { printf("E"); kill(parent_pid, SIGUSR1); }</pre>	<pre>int main() { pid_t parent_pid = getpid(); pid_t pid; signal(SIGUSR1, &handle_sigusr1); signal(SIGUSR2, &handle_sigusr2); if((pid = fork()) == 0){ child1_action(parent_pid); } if((pid = fork()) == 0){ child2_action(parent_pid); } if((pid = fork()) == 0){ child3_action(parent_pid); } while (wait(NULL) > 0); return 0; }</pre>
--	---

sig_atomic_t 是一种异步信号安全 (Async-Signal-Safe) 的整数类型, 对这一类型的数据访问/修改都是“原子”的, 不会被信号处理中断。同时假设所有信号量 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. CDEAAB B. CADAEB C. EAD CAB D. DCAEAB E. CDAAEB

(2) 阅读下面程序并选择出可能的运行结果, printf 的输出即时显示在 stdout 上 (3 分, 多选)

- A. ACBD B. CDAB C. ACDB D. CABD E. 发生死锁

(每空 3 分, 答对一个选项+1, 答错一个-1)

(1) ACD

(2) ABCDE (两个及以下 1 分, 3 个 2 分, 全写 3 分)

```

int ready = 0;

void* child1(void* arg) {
    printf("A\n");
    P(&mutex);
    while (!ready) sleep(1);
    V(&mutex);
    printf("B\n");
    return NULL;
}

void* child2(void* arg) {
    printf("C\n");
    P(&mutex);
    ready = 1;
    V(&mutex);
    printf("D\n");
    return NULL;
}

int main() {
    pthread_t t1, t2;
    sem_init(&mutex, 0, 1);

    pthread_create(&t1, NULL, child1, NULL);
    pthread_create(&t2, NULL, child2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}

```

12、阅读以下代码，在程序正常运行结束后，tmp.txt 文件的内容是什么？（3 分）解释为什么是这个结果（可以把关键函数调用后文件的内容和读写位置列出来，并简要说明原因，要求必须写出 12 行、23 行、28 行、30 行函数调用后文件的内容，5 分）。

说明：lseek 函数的第三个参数表示设置文件读写位置的方式，SEEK_SET 为相对于文件头部，SEEK_END 为相对于文件尾部，SEEK_CUR 为相对于当前读写位置。

fgets/fputs 函数完成 C 语言的字符串的读写，fgets 函数的缓冲区长度参数 len 包含结尾的 \0 字符（也就是在读取时遇到 \n 或者读够 len-1 个字符就结束，并把 \0 字符加到读入的字符后面），fputs 函数输出整个字符串（不包括 \0）。

```

01 #include <fcntl.h>
02 #include <stdio.h>
03 #include <stdlib.h>
04 #include <sys/wait.h>
05 #include <unistd.h>
06
07 int main() {
08     int fd, fd0, fd1;
09     char buf[4];
10
11     fd = open("tmp.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
12     write(fd, "dogcatfoxpig", 12);
13     close(fd);
14
15     fd0 = open("tmp.txt", O_RDONLY);

```

```

16  fd1 = open("tmp.txt", O_WRONLY);
17  dup2(fd0, 0);
18  dup2(fd1, 1);
19  read(fd0, buf, 3);
20  lseek(fd0, -3, SEEK_END);
21  if (fork() == 0) {
22      fgets(buf, 4, stdin);
23      fputs(buf, stdout);
24      fflush(stdout);
25      exit(0);
26  }
27  wait(NULL);
28  write(fd1, "r", 1);
29  lseek(fd1, 5, SEEK_CUR);
30  write(fd1, buf, 3);
31  close(fd0);
32  close(fd1);
33
34  return 0;
35 }

```

pigratfoxdog

第 12 行后: dogcatfoxpig

第 23 行后: pigcatfoxpig (写 dogcatfoxpig 也可以, fputs 写到缓冲区还没有 write)

第 28 行后: pigratfoxpig

第 30 行后: pigratfoxdog

13、阅读以下代码片段回答问题, 假设 handler 中的函数都是信号异步安全的。

(1) 该代码编译后的程序执行是会正常退出、陷入死循环还是会异常退出? (2 分)

(2) 该代码陷入死循环或者退出之前可能输出什么内容 (不包含异常退出的异常信息)?

如果有多种可能, 给出任意一种即可。 (4 分)

(3) 该内容是唯一的可能吗? (2 分)

```

01 int pid = 0, level = 0, status = 0;
02 void handler(int sig) {
03     if (sig == SIGUSR1) {
04         kill(pid, SIGUSR2);
05         int ppid = getppid();
06         if (level != 0) {
07             waitpid(pid, &status, 0);
08             status >>= 8; // status / 256 可以得到等待进程的返回值。
09             kill(ppid, SIGUSR1);
10         } else {
11             exit(0);
12         }
13     } else if (sig == SIGUSR2) {
14         printf("%d: %d\n", level, status);
15         exit(level);
16     }
17 }
18

```

```

19 int main() {
20     signal(SIGUSR1, handler);
21     signal(SIGUSR2, handler);
22     for (int i = 0; i < 10; i++) {
23         pid = fork();
24         if (pid != 0)
25             break;
26         else
27             level++;
28     }
29     while (pid) ;
30     int ppid = getppid();
31     kill(ppid, SIGUSR1);
32     while (1) ;
33     return 0;
34 }

```

参考答案：

1. 正常退出
2. 10: 0
9: 10
8: 9
7: 8
6: 7
5: 6
4: 5
3: 4
2: 3
1: 2
3. 唯一

14、理发店有 N 个理发师和 M 个顾客，每个顾客进入理发店后会排队等待理发师呼叫，并找对应的理发师理发。理发师会在空闲时呼叫队列中最早到达的顾客。每个理发师和每个顾客分别为一个线程，顾客编号为函数参数 c，理发师编号为函数参数 b。请补充下列基于信号量和 PV 原语的代码（10 分）。

```

sem_t sem_count; // 正在等待的顾客人数
sem_t sem_queue; // 排队队列互斥锁
sem_t sem_customers[__(A)]; // 顾客等待叫号
volatile int assigned_barber[M]; // 顾客被分配到的理发师
std::queue<int> q;
// 初始化工作

```

```

void init() {
    sem_init(&sem_count, 0, 0);
    sem_init(&sem_queue, 0, __ (B)__);
    for (int i = 0; i < __ (C)__; i++)
        sem_init(&sem_customers[i], 0, __ (D)__);
}
// 理发师线程
void barber(int b) {
    while (true) {
        P(__ (E)__);
        P(__ (F)__);
        int c = q.front();
        q.pop();
        V(sem_queue);
        __ (G) __
        V(sem_customers[b]);
        // 为顾客 c 理发
    }
}
// 顾客线程
void customer(int c) {
    P(__ (H)__);
    q.push(c);
    V(sem_queue);
    V(__ (I)__);
    P(__ (J)__); // 等待至有理发师叫号
    b = assigned_barber[c];
    // 找理发师 b 理发
}

```

参考答案:

A: M

B: 0

C: M

D: 0

E sem_count

F: sem_queue

G: assigned_barber[c] = b;

H: sem_queue

I: sem_count

J: sem_customers[c]