

南京邮电大学 2024/25 学年 第 2 学期

《计算机系统基础 I》期末试卷 (A)

院(系)_____ 班级_____ 学号_____ 姓名_____

注意：答案必须写在答题纸上，写在本试卷上无效

得 分	一、单项选择题（每题 2 分，共 20 分）

1. 某云计算公司优化其图像渲染服务。原始单核版本处理一帧耗时 100 秒，分析显示 95% 的计算可并行化（即并行比例 $P=0.95$ ）。但在多核改造中，串行部分指令数增加了 50%（因同步开销），并行部分指令数不变。单核配置：时钟频率 4.0 GHz，串行和并行部分 CPI 均为 1.0。现改用 8 核处理器，但因功耗限制，频率降至 3.5 GHz，且并行部分 CPI 增至 1.2（因内存争用）。假设并行任务均匀分布，求 8 核处理一帧的耗时（）

- A. 12.5 秒
- B. 15.0 秒
- C. 20.0 秒
- D. 25.0 秒

2. 在 32 位补码系统中，以下关于补码特性的描述中，正确的是（）

- A. 所有正数的补码与其原码相同，且符号位为 0。
- B. 计算表达式 $(-2147483648) + (-1)$ 的 32 位补码结果，与 $2147483647 + 2$ 的补码结果相同。
- C. 对任意整数 x ， $-x$ 的补码可通过 x 的补码按位取反后加 1 得到。
- D. -125 的 8 位二进制补码可表示为 10000010

3. C 语言代码：

```
#include <stdio.h>
```

```
typedef unsigned short US;
```

```
typedef short S;
```

```
S arr[5] = {0, 1, 2, 3, 4};
```

```
US gen_idx() { return 32768; } // 返回 US: 0x8000 (16 位无符号)
```

```
int recurse(int depth, US bound) {
```

```
    if (depth == 0) return 0;
```

```
    S idx = gen_idx();
```

```
    int val = arr[idx] + (bound - 1);
```

```

        return val + recurse(depth - 1, bound);
    }

int main() {
    US limit = 32769; // 0x8001 (16 位无符号)
    int res = recurse(2, limit);
    printf("%d\n", res);
    return 0;
}

```

关于程序行为，错误的是 ()

- A. idx_gen()返回值转换为 S 后，idx 的值为 -32768
- B. 第一次调用 recurse 时，arr[idx] 的访问会触发数组越界
- C. 表达式 (bound - 1) 中，bound (US) 与 1 (int) 的运算按无符号整数规则执行
- D. 关系表达式 -1 < 0U 的结果为 0

4. 在 IEEE 754 单精度浮点数格式中，以下描述中错误的是： ()

- A. 最小正规格化数是 2^{-126} ，最大正非规格化数是 $2^{-126} \times (1 - 2^{-23})$
- B. 表达式 $1.0e20 + 1.0 - 1.0e20$ 的计算结果为 0.0，而 $1.0e20 - 1.0e20 + 1.0$ 的计算结果为 1.0。
- C. 最小正规格化数 ($1.17549435e-38$) 的 2 倍等于最小正非规格化数 ($1.40129846e-45$)。
- D. 若 x 是 NaN，则 $x != x$ 为真；若 x 是无穷大 (Inf)，则 $x + 1.0 == x$ 为真。

5. 将 -131072.5 转换为 IEEE 754 单精度浮点格式，并给出十六进制结果 ()

- A. 0xC8000080
- B. 0xC8000040
- C. 0xC8000020
- D. 0xC8000010

6. 给定三个源文件：main.c、module1.c 和 module2.c，内容如下：

```

// main.c:
#include <stdio.h>

void foo();
void bar();
int x = 42;
static int y = 10;

int main() {
    foo();
    bar();
    printf("x = %d, y = %d\n", x, y);
    return 0;
}

module1.c:

```

```
Cstatic int x;  
int y = 20;
```

```
void foo() {  
    x = 100;  
    y = 200;  
}
```

module2.c:

```
Cint x;  
int y;
```

```
void bar() {  
    x = 300;  
    y = 400;  
}
```

当这三个文件被编译并链接成一个可执行程序后，运行程序输出的结果是（）

- A. x = 42, y = 10
- B. x = 300, y = 10
- C. x = 42, y = 400
- D. x = 300, y = 400

7. 在函数 helper 的汇编实现中, 为了遵循调用约定, 被调用者保存寄存器的使用是必要的。假设在 helper 函数中, 它需要使用寄存器 rbx 和 r12 来执行一些计算, 并且在返回前恢复它们。已知在进入 helper 时, 寄存器的状态如下: rbx 保存的是调用者 caller 中的某个值 (比如 c 的值), r12 保存的是调用者 caller 中的另一个值 (比如 d 的值)。在 helper 中, 我们想使用这些寄存器来临时存储一些中间结果, 但必须确保在返回时恢复它们原来的值。在 helper 的汇编代码中, 我们应该如何正确地使用 rbx 和 r12? 请选择正确的操作序列 ()

A.

```
push rbx  
push r12  
... ; 函数体使用 rbx 和 r12  
pop rbx  
pop r12  
ret
```

B.

```
mov [rsp-8], rbx  
mov [rsp-16], r12  
... ; 函数体使用 rbx 和 r12  
mov rbx, [rsp-8]  
mov r12, [rsp-16]  
ret
```

C.

```

push rbx
push r12
... ; 函数体使用 rbx 和 r12
pop rbx
pop r12
ret

```

D.

```

push r12
push rbx
... ; 函数体使用 rbx 和 r12
pop rbx
pop r12
ret

```

8. 假设有以下依赖关系: $\text{main.o} \rightarrow \text{liba.a} \rightarrow \text{libb.a} \rightarrow \text{libc.a} \rightarrow \text{liba.a} \rightarrow \text{main.o}$ 。请给出最小的命令行 (即一个含有最少数量的目标文件和库参数的命令), 使得静态链接器能解析所有的符号引用 ()

- A. gcc main.o liba.a libb.a libc.a
- B. gcc main.o liba.a libb.a libc.a liba.a
- C. gcc main.o liba.a libb.a libc.a liba.a main.o
- D. gcc main.o liba.a libb.a libc.a liba.a libb.a

9 在 x86-64 架构中, 考虑以下 C 语言函数:

```

long complex_calc(long a, long b) {
    long x = a * a;      // 语句 1
    long y = b * b;      // 语句 2
    external_func();     // 语句 3 (外部函数, 遵循调用约定)
    long z = x * y;      // 语句 4
    return z + a + b;    // 语句 5
}

```

编译时使用-O1 优化。已知编译器分配: a 和 b 分别存储在%rdi 和%rsi, x 和 y 分别存储在%rbx 和%r12 (被调用者保存寄存器), z 存储在%rax

当执行到语句 4 时 ($z = x * y$), 以下关于寄存器状态的描述正确的是 ()

- A. %rbx 和%r12 的值与语句 3 执行前相同, 因为 external_func()会保存并恢复它们
- B. %rbx 和%r12 的值可能被 external_func()修改, 因为它们是调用者保存寄存器
- C. %rbx 和%r12 的值由编译器在调用 external_func()前显式保存到栈中
- D. %rbx 和%r12 的值必定被破坏, 因为 external_func()不保存调用者寄存器

10. 在 x86-64 Linux 系统中, 假设程序员编写了一个共享库 libexample.so, 其中包含以下代码:

```

// libexample.so
int global_var = 0x12345678; // 全局变量
static int local_var = 0x87654321; // 静态变量

```

```
void update_global() {  
    global_var += 0x100;  
}
```

```
void update_local() {  
    local_var += 0x100;  
}
```

该库被编译为位置无关代码 (PIC)，并与一个可执行文件 main 静态链接。在可执行文件的 .text 节中，某条指令需要访问 global_var 的地址，其机器码为：

48 8B 05 00 00 00 00 mov rax, QWORD PTR [rip+0x0]

(注：RIP 相对寻址模式)

该指令对应的重定位条目类型应为 ()

- A. R_X86_64_32 (绝对地址重定位)
- B. R_X86_64_PC32 (PC 相对地址重定位)
- C. R_X86_64_GOTPCREL (全局偏移表 (GOT) 相对偏移)
- D. R_X86_64_RELATIVE (运行时地址偏移修正)

得分

二、分析题（每题 5 分，共 20 分）

1. R[ax]=FFF4H, R[bx]=7FF3H, 执行指令“subw %bx, %ax”后, 寄存器的内容和标志寄存器标志位 ZF, CF, OF, SF 的值分别是 0x____、____、____、____和____。

2.

long loop(long a,long b)

```
{
    long result = ____;
    while(____)
    {
        if(____)
            result = ____;
        a=____;
    }
    return result;
}
```

}

使用 gcc -Og 得到下面的汇编代码,请你参考他们补全上方的 C 代码

loop:

```
        movl    $1, %eax
        jmp     .L2
.L3:
        addq    $1, %rdi
.L2:
        cmpq    %rsi, %rdi
        jge     .L5
        testb   $1, %dil
        je      .L3
        leaq    (%rdi,%rsi), %rdx
        imulq   %rdx, %rax
        jmp     .L3
.L5:
        Ret
```

3.假设下面的值存放在指定的内存单元和寄存器中，填写下面各条指令执行结果。(每次执行指令都是初始条件，不受上一条指令影响) M[addr]表示对内存中地址从 addr 的开始的 1 个字节的数据的引用

内存单元地址	值
0x100	0xff
0x104	0xab

寄存器	值
rax	0x104
rbx	0xff

指令	指令执行结果
subq \$0x4, %rax	rax= M[0x104]=
movq \$0x104,-4(%rax)	M[0x100]= M[0x104]=
addl 0x100, %ebx	ebx= M[0x100]=
salb \$2,%bl	bl =

3. 请简述为什么在开启 GCC -O3 优化后,foo 中 if-else 两个分支的代码会在实际执行条件判断前均被执行, 这样有什么好处

```
long foo(long x,long y )
```

```
{
    long res;
    if(x<y)
        res =y-x;
    else
        res = x-y;
    return res;
}
```

```
foo:
```

```
    movq    %rsi, %rdx
    movq    %rdi, %rax    rval = y-x
    subq    %rdi, %rdx
    subq    %rsi, %rax    eval = y-x
    cmpq    %rsi, %rdi    Compare x:y
    cmovl   %rdx, %rax    If >= , reval = eval
    Ret                                Return    reval
```

得分

三、综合应用题

一个 C 语言程序有两个源文件：main.c 和 sum.c,他们的内容如下图所示。
设在 x86-64/linux 平台上（采用小端法规则），用 GCC 编译驱动程序处理。

```
/*main.c*/
#include <stdio.h>

typedef struct {
    int id;
    int val;
} StructData;

int process_struct(StructData *arr, int len);

static StructData data[5] = {{1, 2}, {2, -5}, {3, 7}, {4, 12}, {5, -3}};

int global_product;

int main() {
    int n;
    scanf("%d", &n);

    int sum_val = process_struct(data, n);

    printf("sum = %d, product = %d\n", sum_val, global_product);
    return 0;
}
```

```
/*sum.c*/
typedef struct {
    int id;
    int val;
} StructData;

extern int global_product;

int process_struct(StructData *arr, int len) {
    int sum = 0;
    global_product = 1;
    for (int i = 0; i < len; i++) {
        sum += arr[i].val;
        global_product *= arr[i].val;
    }
}
```

```
return sum;
}
```

Main.c 和 sum.c 的可重定位目标文件名分别是 main.o 和 sum.o，生成的可执行文件名为 sum。使用“objdump-d sum”得到可执行目标文件 sum 的反汇编部分结果如下。

```
0000000000001159 <main>:
    1159: 55                push    %rbp
    115a: 48 89 e5          mov     %rsp,%rbp
    115d: 48 83 ec 10       sub     $0x10,%rsp
    1161: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
    1168: 00 00
    116a: 48 89 45 f8       mov     %rax,-0x8(%rbp)
    116e: 31 c0            xor     %eax,%eax
    1170: 48 8d 45 f0       lea     -0x10(%rbp),%rax
    1174: 48 8d 15 89 0e 00 00 lea     0xe89(%rip),%rdx          # 2004
<_IO_stdin_used+0x4>
    117b: 48 89 c6          mov     %rax,%rsi
    117e: 48 89 d7          mov     %rdx,%rdi
    1181: b8 00 00 00 00    mov     $0x0,%eax
    1186: e8 c5 fe ff ff    call    1050 <__isoc23_scanf@plt>
    118b: 8b 45 f0          mov     -0x10(%rbp),%eax
    118e: 48 8d 15 ab 2e 00 00 lea     0x2eab(%rip),%rdx          # 4040 <data>
    1195: 89 c6            mov     %eax,%esi
    1197: 48 89 d7          mov     %rdx,%rdi
    119a: e8 _ _ _ _ _      call    11dc <process_struct>
    119f: 89 45 f4          mov     %eax,-0xc(%rbp)
    11a2: 8b 15 c4 2e 00 00 mov     0x2ec4(%rip),%edx          # 406c
<global_product>
    11a8: 8b 45 f4          mov     -0xc(%rbp),%eax
    11ab: 48 8d 0d 55 0e 00 00 lea     0xe55(%rip),%rcx          # 2007
<_IO_stdin_used+0x7>
    11b2: 89 c6            mov     %eax,%esi
    11b4: 48 89 cf          mov     %rcx,%rdi
    11b7: b8 00 00 00 00    mov     $0x0,%eax
    11bc: e8 7f fe ff ff    call    1040 <printf@plt>
    11c1: b8 00 00 00 00    mov     $0x0,%eax
    11c6: 48 8b 55 f8       mov     -0x8(%rbp),%rdx
    11ca: 64 48 2b 14 25 28 00 sub     %fs:0x28,%rdx
    11d1: 00 00
    11d3: 74 05            je      11da <main+0x81>
    11d5: e8 56 fe ff ff    call    1030 <__stack_chk_fail@plt>
    11da: c9              leave
    11db: c3              ret

00000000000011dc <process_struct>:
```

11dc:	55	push	%rbp	
11dd:	48 89 e5	mov	%rsp,%rbp	
11e0:	48 89 7d e8	mov	%rdi,-0x18(%rbp)	
11e4:	89 75 e4	mov	%esi,-0x1c(%rbp)	
11e7:	c7 45 f8 00 00 00 00	movl	\$0x0,-0x8(%rbp)	
11ee:	c7 05 74 2e 00 00 01	movl	\$0x1,0x2e74(%rip)	# 406c <global_product>
11f5:	00 00 00			
11f8:	c7 45 fc 00 00 00 00	movl	\$0x0,-0x4(%rbp)	
11ff:	eb 44	jmp	1245 <process_struct+0x69>	
1201:	8b 45 fc	mov	-0x4(%rbp),%eax	
1204:	48 98	cltq		
1206:	48 8d 14 c5 00 00 00	lea	0x0(,%rax,8),%rdx	
120d:	00			
120e:	48 8b 45 e8	mov	-0x18(%rbp),%rax	
1212:	48 01 d0	add	%rdx,%rax	
1215:	8b 40 04	mov	0x4(%rax),%eax	
1218:	01 45 f8	add	%eax,-0x8(%rbp)	
121b:	8b 45 fc	mov	-0x4(%rbp),%eax	
121e:	48 98	cltq		
1220:	48 8d 14 c5 00 00 00	lea	0x0(,%rax,8),%rdx	
1227:	00			
1228:	48 8b 45 e8	mov	-0x18(%rbp),%rax	
122c:	48 01 d0	add	%rdx,%rax	
122f:	8b 50 04	mov	0x4(%rax),%edx	
1232:	8b 05 34 2e 00 00	mov	0x2e34(%rip),%eax	# 406c <global_product>
1238:	0f af c2	imul	%edx,%eax	
123b:	89 05 2b 2e 00 00	mov	%eax,0x2e2b(%rip)	# 406c <global_product>
1241:	83 45 fc 01	addl	\$0x1,-0x4(%rbp)	
1245:	8b 45 fc	mov	-0x4(%rbp),%eax	
1248:	3b 45 e4	cmp	-0x1c(%rbp),%eax	
124b:	7c b4	j1	1201 <process_struct+0x25>	
124d:	8b 45 f8	mov	-0x8(%rbp),%eax	
1250:	5d	pop	%rbp	
1251:	c3	ret		

采用 gdb 调试程序，当 CPU 准备运行地址 119a 处的 call 指令，此时部分调试信息如下：

(gdb) i r rsp rdi rsi (查看寄存器内容)。

rsp 0x7ffffffd770 rdi 0x555555558040 rsi 0x4

基于给出的代码以及给出调试信息，回答下列问题或完成下列任务。

1. 写出从 C 语言源程序到生成带有调试信息的可重定位文件 sum.o 的命令行 (不进行优化)。

2.填写下表各标识符的情况,说明每个标识符是否出现在 main.o 的符号表(.symtab 节)中,如果是的话,进一步说明定义该符号的模块是 main.o 还是 sum.o、该符号的类型是全局、外部还是局部符号。

标识符	是否在 main.o 的符号表中?	定义模块	符号类型
StructData			
process_struct			
data			
global_product			

3 在 main.o 的.bss 节可以看到哪些符号?

4.程序调试执行时, 输入的 n 的值是多少? 程序屏幕输出的结果是什么?

4. 计算机按字节编址, 0x555555558048 开始的连续 32 字节地址单元中存放的内容是什么?
 (gdb)x/4xg 555555558048 (按 64 位一个字节显示)
 0x555555558048: _____
 0x555555558058: _____

6. 119a 处 call 指令的操作码是 0xe8,其余字节是偏移量, 偏移量采用补码表示, 转移的目标地址采用相对寻址方式。重定位表如下所示。

readelf -r main.o

重定位节 '.rel.text' at offset 0x308 contains 8 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
00000000001e	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
00000000002e	000800000004	R_X86_64_PLT32	0000000000000000	__isoc23_scanf - 4
000000000038	000300000002	R_X86_64_PC32	0000000000000000	.data - 4
000000000042	000900000004	R_X86_64_PLT32	0000000000000000	process_struct - 4
00000000004b	000600000002	R_X86_64_PC32	0000000000000000	global_product - 4
000000000055	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 1
000000000064	000a00000004	R_X86_64_PLT32	0000000000000000	printf - 4
00000000007d	000b00000004	R_X86_64_PLT32	0000000000000000	__stack_chk_fail- 4

重定位节 '.rel.eh_frame' at offset 0x3c8 contains 1 entry:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0

分析该 call 指令的机器指令有几个字节?写出该 call 指令的完整机器指令。

119a: e8 _ _ _ _ call 11dc <process_struct>

7.分析程序运行过程中 main 函数的栈帧空间大小的变化

8. 11ca 处的指令有什么作用？

9. 哪几条指令实现了 `global_product *= arr[i].val` 功能

得分

四、实验题

甲同学希望通过重定位实验学习链接相关知识，实验要求，程序运行后输出他的学号(123456789)

文件 `phase1.o` 的反汇编结果如下

`phase1.o:` `file format elf64-x86-64`

Disassembly of section `.text`:

```
0000000000000000 <do_phase>:
  0:  f3 0f 1e fa          endbr64
  4:  55                   push    %rbp
  5:  48 89 e5             mov     %rsp,%rbp
  8:  b8 00 00 00 00      mov     $0x0,%eax
                        9: R_X86_64_32 .data+0x4c
  d:  48 89 c7             mov     %rax,%rdi
 10:  e8 00 00 00 00      call   15 <do_phase+0x15>
                        11: R_X86_64_PLT32 puts-0x4
 15:  90                   nop
 16:  5d                   pop     %rbp
 17:  c3                   ret
```

下面是 `readelf -S phase1.o` 的结果

There are 15 section headers, starting at offset 0x3a0:

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000		0 0 0
[1]	.text	PROGBITS	0000000000000000	00000040
	0000000000000018	0000000000000000	AX	0 0 1
[2]	.rela.text	RELA	0000000000000000	000002a8
	0000000000000030	0000000000000018	I	12 1 8
[3]	.data	PROGBITS	0000000000000000	00000060
	00000000000000a0	0000000000000000	WA	0 0 32
[4]	.rela.data	RELA	0000000000000000	000002d8
	0000000000000030	0000000000000018	I	12 3 8
[5]	.bss	NOBITS	0000000000000000	00000100
	0000000000000000	0000000000000000	WA	0 0 1
[6]	.rodata	PROGBITS	0000000000000000	00000100
	0000000000000002	0000000000000000	A	0 0 1

下面是 hexedit phase1.o 跳转到相关地址后的结果

(site)表示行首第一个字节的起始地址.

```
(00000090)  6D 4C 72 4F  72 68 52 42  4E 4B 36 6F  6C 46 4C 41  4C 6A 33 31  36 55
4D  39      46  5A  50  58      59  73  42  77      38  44  62  70
mLrOrhRBNK6oIFLALj316UM9FZPXyBw8Dbp
(000000B4)  47 4D 4C 65  70 4D 6D 7A  51 09 4B 44  41 43 41 78  4C 70 77 35  51 31
4B  09      32  63  30  78      67  31  52  71      77  46  72  4A
GMLePmMzQ.KDACAxLpw5Q1K.2c0xg1RqwFrJ
(000000D8)  53 59 35 4A  43 47 56 45  38 32 43 41  4C 44 55 45  63 71 20 46  35 56
6A 54  66 00 00 00  00 00 00 00  00 00 00 00  SY5JCGVE82CALDUEcq F5VJTf.....
```

(1) 需要修改的字符串的起始位置在哪里(整个文件的开始设为地址 0x0，给出相对于这个位置的地址)

(2) 一个函数在其函数体中递归调用自己，而此时它自身尚未“完成定义”，但链接却能成功——为什么？