# 哈尔滨工业大学（深圳）

## 2025 年春《计算机系统》期末考试试题（A） 答案

题目来源：对应回忆版试题 题解：Chi. Ya.

本参考答案中使用到的部分题目的表述在原回忆版试题的基础上进行了调整。

一、选择题（共 20 分，每题 2 分）

1. 虚拟机是对 _____ 的抽象

A. I/O 设备

B. 处理器硬件

C. 操作系统

D. 整个计算机

虚拟机通过软件模拟硬件，把整台物理计算机：CPU、内存、I/O 设备都抽象出来，给上层软件提供一台假想的完整机器。因此它抽象整个计算机。

故答案为：$\boxed{D}$

2. 下列说法正确的是 _____

I. 浮点数不是实数

II. 浮点加法有结合律，乘法也有结合律

III. 浮点加法无结合律，乘法有结合律

IV. 浮点加法有结合律，乘法无结合律

V. 浮点数运算无分配率

A. I,II,V

B. I,III,V

C. I,IV,V

D. I,V

浮点数仅能近似实数，而不是实数。浮点数加法无结合律、乘法同样无结合律。

故答案为：$\boxed{D}$

3. 以下哪个条件可以判断补码加法 `s = x + y` 发生了负溢出？ _____

A. `x>0, y>0, s<=0`

B. `x>0, y<0, s>=0`

C. `x<0, y>0, s<=0`

D. `x<0, y<0, s>=0`

补码加法只有在两个操作数同号而结果异号时才溢出。

$x > 0, y > 0, s \leq 0$ ：正溢出。

$x < 0, y < 0, s \geq 0$ ：负溢出。

故答案为：$\boxed{D}$

4. `short t = -8196;`
   `int y = t;`
   y 的十六进制表示为 _____
   A. 0x00009FFC
   B. 0x0000DFFC
   C. 0xFFFF9FFC
   D. 0xFFFFDFFC
   -8196 的 16 位原码为 0x2004：取反加一得补码 0xDFFC。向 int 赋值时会符号扩展，高 16 位补 1。
   故答案为：D

5. 已知依赖 fun.o -> libx.a, liby.a; libx.a -> libz.a; libx.a 与 liby.a 相互独立，libz.a 与 liby.a 相互独立，则下面给出的编译命令中，会发生错误的是：_____
   A. `gcc -static -Og fun.o libx.a liby.a libz.a`
   B. `gcc -static -Og fun.o liby.a libz.a libx.a`
   C. `gcc -static -Og fun.o libx.a libz.a liby.a`
   D. `gcc -static -Og fun.o liby.a libx.a libz.a`
   静态链接器自左向右扫描库：当处理 libx.a 时，其中对 libz.a 的符号引用必须等到后面才能解析。这就是说 libx.a 不能在 libz.a 后；fun.o 不能在 libx.a 和 liby.a 后。
   故答案为：B

6. 由于 C 语言提供指针运算，因此 _____ 现象会导致潜在的性能下降和正确性下降
   A. 非法指针
   B. 函数调用
   C. 内存别名
   D. 缺页
   多指针指向同一内存区域（内存别名）让编译器难以判断数据依赖，既妨碍优化又可能因误写导致数据被意外修改，影响性能与正确性。
   故答案为：C

7. 以下 _____ 属于异步异常
   A. 计时器中断
   B. 缺页
   C. 系统调用
   D. 非法指令
   异步异常即中断来源于处理器外部事件，与当前指令无关；计时器中断正是典型异步中断。缺页、系统调用、非法指令都是同步异常。
   故答案为：A

8. 关于 fork() 的调用，错误的是 _____
   A. 调用一次，返回两次

B. 父子进程间运行顺序不可预测

C. 对子进程返回 0，对父进程返回子进程的进程号 pid

D. 子进程和父进程执行完全相同的代码直到进程结束

fork() 后父子从同一点开始执行，但随后可执行不同分支逻辑。一直执行完全相同的说法当然是不合适的。

故答案为：$\boxed{D}$

9. 关于虚拟地址空间的叙述，不正确的是 _____

A. DRAM 空间可以理解为虚拟地址的缓存

B. 每个程序都有独立的虚拟地址空间

C. 虚拟地址直接映射到物理地址的连续区域

D. 虚拟内存增强了进程的局部性

虚拟地址到物理地址并不要求一一连续映射。虚拟地址直接映射到物理地址的连续区域说法不正确。

故答案为：$\boxed{C}$

10. 关于动态库，以下说法不正确的是 _____

A. 可在加载时链接，即当可执行文件首次加载和运行时进行动态链接。

B. 当动态库更新但接口不变时，使用它的程序无需重新编译

C. 动态库可在编译期与装载期链接，不能在运行时链接

D. 即便有多个正在运行的程序使用同一动态库，系统也仅在内存中载入一份动态库。

动态库既可在装载时自动链接，也可在运行时用 dlopen/dlsym 手动加载。运行时不能链接是不正确的。

故答案为：$\boxed{C}$

二、填空题（共 10 分，每空 2 分）

1. 写出将 int 变量 x 的最高有效字节清零的 C 表达式：_____（限一个表达式）。

对 32 位整型清零最高有效字节，只要将 x 与掩码 0x00FFFFFF 按位与即可。

x = ((unsigned)x<<8)>>8 也可以。

故答案为：x &= 0x00FFFFFF 或x = ((unsigned)x<<8)>>8

2. 64 位系统中 struct node{ int a; char c1; char c2; short x; float *p;}，则 sizeof(node) 的值为 _____。

int 按 4 对齐，char 按 1 对齐，short 按 2 对齐与 float * 按 8 对齐，各元素占有字节如下：a[0:3]，c1[4:4]，c2[5:5]，x[6:7]，p[8:15]，整个结构体已经按最大元素对齐。因此就这么大。

故答案为：16

3. Unix 将 I/O 设备抽象为：_____。

Unix 将所有 I/O 设备（终端、磁盘、网络套接字等）统一抽象为文件，上层通过统一的文件

接口（open/read/write/close）操作。
故答案为：文件

4. 要将程序的标准输出 stdout（文件描述符为 1）重定向到 file.txt（设其文件描述符为 4），内核调用的函数是 _____（包括函数名和参数）。
要让文件描述符 1（stdout）指向已有的描述符 4，只需用系统调用 dup2(oldfd,newfd)：dup2(4, 1)；把 4 复制到 1，旧 1 即被关闭。

故答案为：dup2(4, 1)

5. 虚拟页面的状态有 _____、已缓存、未缓存。
未分配（not allocated）、已缓存（cached，在 DRAM）、未缓存（uncached，在磁盘）。
故答案为：未分配

三、简答题（共 15 分）

1. 简述缓冲区溢出的原理及防范方法。（5 分）
攻击原理：
（1）程序输入缓冲区写入特定的数据，例如在 gets 读入字符串时
（2）使位于栈中的缓冲区数据溢出，用特定的内容覆盖栈中的内容，例如函数返回地址等
（3）使得程序在读入字符串，结束函数 gets 从栈中读取返回地址时，错误地返回到特定的位置，执行特定的代码，达到攻击的目的。

防范方法：
（1）代码中避免溢出漏洞：例如使用限制字符串长度的库函数。
（2）随机栈偏移：程序启动后，在栈中分配随机数量的空间，将移动整个程序使用的栈空间地址。
（3）限制可执行代码的区域
（4）进行栈破坏检查——金丝雀

2. 由以下 C 语言代码 `hello.c` 生成可执行文件需要经历哪几个步骤？描述每个步骤发生了什么变化。（5 分）

```
#include <stdio.h>
int main(void) {
    printf("hello world\n");
}
```

从源文件 `hello.c` 生成最终的可执行文件典型要经历四个主要步骤。

1. 预处理：`hello.c` → `hello.i`

展开宏与 `stdio.h` 头文件；删除注释；得到纯 C 代码文本。

2. 编译：`hello.i` → `hello.s`

语法/语义分析、中间表示、优化、生成与 ISA 相关的汇编文本。

3. 汇编：`hello.s` → `hello.o`

汇编器把每条汇编指令翻译成机器指令，填入各节，构造重定位表等，形成目标文件。

4. 链接：`hello.o`，`printf.o` → `hello(executable)`

解析符号、地址重定位，合并节表，生成可执行文件。

3. 列举三种常见的程序优化方法，并任选一种结合代码示例分析。（5 分）

代码移动是一种把不依赖循环迭代变量且在循环体内重复计算的表达式提前到循环外部的技术，从而减少冗余计算次数。例如：

```
1 double sum = 0;
2 double c = cos(theta);          /* 迭代前先计算 */
3 for (int i = 0; i < n; ++i) {
4     sum += c * a[i];            /* 只留下与 i 相关的部分 */
5 }
```

移动前若在循环内每次都调用 `cos(theta)`，则需要 $n$ 次三角函数计算；移动后只需一次。这样既节省指令数，也降低流水线停顿概率，能显著提高 CPU 利用率。

循环展开通过一次循环体执行多次核心操作来降低分支开销并增加指令级并行度。例如把步长为 1 的循环展开为步长 4：

```
1 for (int i = 0; i < n; i += 4) {
2     y[i  ] += a * x[i  ];
3     y[i+1] += a * x[i+1];
4     y[i+2] += a * x[i+2];
5     y[i+3] += a * x[i+3];
6 }
```

展开后，迭代次数减少为 $\lceil n/4 \rceil$，分支预测负担减轻；同时，多条独立指令可并行发射，占满流水线执行端口。

减少过程调用把短小函数直接嵌入调用点，避免存取栈帧与返回跳转。例如：

```
static inline int add(int x, int y) { return x + y; }
/* 调用处可被编译器替换为 x + y，无额外开销 */
```

当函数体远小于保存/恢复寄存器和跳转耗时时，内联能显著降低指令条数并暴露额外优化机会（如常量传播）。

消除不必要内存引用利用寄存器保存临时值，或把多次访问的全局、结构字段提取到局部变量：

```
void vecsum(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = 0; //temp var

    for (i = 0; i < length; i++) {
        acc = acc + data[i];
    }
    *dest = acc;
}
```

提高并行性——累积把串行的操作拆成若干独立累加器，降低循环内依赖链长度并提升指令级并行度：

```
double s0=0,s1=0,s2=0,s3=0;
for (int i = 0; i < n; i += 4) {
    s0 += a[i  ];  s1 += a[i+1];
    s2 += a[i+2];  s3 += a[i+3];
}
double sum = s0 + s1 + s2 + s3;
```

四条累加链可在不同 ALU 上并行执行，隐藏流水线级联依赖的加法延迟；最后一次合并开销相对较小。

重新结合变换（Reassociation）通过改变运算结合顺序来暴露并行机会或减少循环递归依赖：

```
/* 原串行：((((a+b)+c)+d)+e) */
double t1 = a + b;    double t2 = c + d;
double sum = (t1 + t2) + e;    /* 深度降低并行度提高 */
```

深度由 $n-1$ 降至 $\lceil \log_2 n \rceil$，关键路径缩短。

利用向量指令（Vectorization/SIMD）将标量运算替换为一次处理多数据元素的向量指令，以提升数据级并行性：

```
#include <immintrin.h>
for (int i = 0; i < n; i += 8) {
    __m256 vx = _mm256_loadu_ps(&x[i]);    /* 加载 8 个 float */
    __m256 vy = _mm256_loadu_ps(&y[i]);
    vy = _mm256_fmadd_ps(vx, alpha, vy);  /* vy = alpha*vx + vy */
    _mm256_storeu_ps(&y[i], vy);
}
```

AVX 指令一次完成 8 个浮点乘加，理论吞吐比标量代码高 8 倍；结合对齐、预取和展开可逼近峰值带宽。

四、计算题（共 10 分）

给出 C 语言变量 `float a = -2.1`

1. 给出 `a` 的十六进制表示

2. 再给定 `float b = 1.05;` 两数的二进制表示有多少个位不同?

对 `float a = -2.1f` 和 `float b = 1.05f` 取 IEEE-754 单精度二进制表示（1 位符号 + 8 位阶码 + 23 位尾数）

$$\texttt{a} = -2.1 = -1.\underbrace{00001100110011001100110}_{\text{23 位尾数向偶舍入}}01100\ldots \times 2^1$$

$$\Longrightarrow S = 1,\ E = 127 + 1 = 128 = 1000\,0000_2,\ M = 00001100110011001100110$$

$$\texttt{a} = 1100\ 0000\ 0000\ 0110\ 0110\ 0110\ 0110\ 0110$$

$$= \boxed{\texttt{0xC0066666}}$$

$$\texttt{b} = 1.05 = +1.\underbrace{00001100110011001100110}_{\text{23 位尾数向偶舍入}}01100\ldots \times 2^0$$

$$\Longrightarrow S = 0,\ E = 127 = 0111\ 1111_2,\ M = 00001100110011001100110$$

$$\texttt{b} = 0011\ 1111\ 1000\ 0110\ 0110\ 0110\ 0110\ 0110$$

对比可知有 9 位不同。

五、程序分析题（共 30 分）

1. 分析以下 Y86-64 命令

```
0x006  irmovq $10,  %rsp
_____  call .Label
_____  addq %rdx,  %rax
_____  ret
```

(1) 在表中填写空缺的指令地址。(3 分)

| 地址 | 指令 | 字节数 |
|---|---|---|
| 0x006 | irmovq $10,%rsp | 10 |
| 0x010 | call Label | 9 |
| 0x019 | addq %rdx,%rax | 2 |
| 0x01b | ret | 1 |

(2) 第一条指令和第二条指令之间是否有冒险。如有，给出具体的处理冒险的解决方案。(4 分)

irmovq 在 W 阶段写寄存器%rsp；紧接的 call 在 E 阶段要读%rsp 计算新栈顶，因此存在数据冒险。

解决方案（任一）

- 插入 2 个泡泡；
- 增加前递通路：在判断到寄存器同名后把 irmovq 的 E.valE 直接转发到 call 的 E.srcA 也可。

(3) 补充 ret 的 Y86-64 微指令。(8 分)

| 指令 | ret |
|---|---|
| 取指 | icode : ifun $\leftarrow M_1$[PC]<br><br><br><br>valP $\leftarrow$ PC + 1 |
| 译码 | valA $\leftarrow$ R[%rsp]<br>valB $\leftarrow$ R[%rsp] |
| 执行 | valE $\leftarrow$ valB + 8 |
| 访存 | valM $\leftarrow M_8$[valA] |
| 写回 | R[%rsp] $\leftarrow$ valE |
| 更新 PC | PC $\leftarrow$ valM |

2. 分析以下 x86-64 指令（4 分）

```
.varx
    int 123, -2345   # 123 的地址是 varx, -2345 在 varx+4

movq $varx, %rax
movl 4(%rax), %edx
movl $-1, %rax
```

(1) 执行 `movl 4(%rax), %edx` 后 `%rdx` =_____（long long int 十进制表示）

(2) 执行 `movl $-1, %rax` 后 `%rax` =_____（long long int 十进制表示）

`movl 4(%rax), %edx` 把 `-2345` 读入 32 位 `%edx`，随后硬件把高 32 位清零：

$$\texttt{\%rdx} = \texttt{0x00000000FFFFF6D7} = 2^{32} - 2345 = \boxed{4294964951}$$

`movl $-1, %rax` 把低 32 位设为 `0xFFFFFFFF`，同样清零高 32 位：

$$\texttt{\%rax} = \texttt{0x00000000FFFFFFFF} = 2^{32} - 1 = \boxed{4294967295}$$

3. 分析以下 x86-64 机器上的代码（6 分）

```
struct rec {
    int a[4];
    int i;
    struct rec* next;
}
void fun(struct rec* r, int val) {
    while (r) {
          int i = r->i;
          a[i] = val;
          r = r->next;
    }
}
```

下面是实现上述循环的汇编代码，代码中存在错误，在错误的指令右边写出正确的汇编指令（注意：无需添加寄存器）。

```
jmp .L12
.L11:
    movslq 16(%esi), %rax
    movl %esi, (%edi, %eax, 4)
    movq 20(%edi), %edi
.L12:
    testq %edi, %edi
    je .L11
```

| 原始指令 | 正确指令 (理由) |
|---|---|
| `movslq 16(%esi), %rax` | `movslq 16(%edi), %rax` （第一参数保存在%edi） |
| `movq 20(%edi), %edi` | `movq 24(%rdi), %rdi` （next 位于偏移24字节） |
| `je .L11` | `jne .L11` （ZF=0 表示r $\neq 0$，循环继续） |

4. （5 分）

```
int main() {
    int status, count = 0;
    int pid = fork();
    if (pid == 0) {
          printf("%d\n", --count);
          exit(0);
    }
    else {
          printf("%d\n", ++count);
          wait(&status);
          printf("%d\n", ++count);
    }
    printf("%d\n", ++count);
}
```

(1) 共有几种输出可能?（1 分）

2

(2) 写出所有可能的输出序列。（4 分）

| | |
|---|---|
| −1 | 1 |
| 1 | −1 |
| 2 | 2 |
| 3 | 3 |

说明：

- 子进程 `printf()` 先运行则先打印 `-1`。
- 父进程 `printf()` 先运行，则先打印 `1`。
- 只有子进程死了导致 `wait` 返回后父进程才能打印 `2, 3`。

六、综合分析题（共 15 分，每空 1 分）

某计算机系统的情况如下：

1. 内存是按字节寻址，字长为 2 字节（非 4 字节），采用小端模式存储；
2. 虚拟地址长度：24 位；
3. 使用一级页表，页面大小是 2048 字节；
4. 物理地址的位数是 19 位；
5.TLB（翻译后备缓冲器）是 4 路组相连，共 16 个条目；
6.L1 d-Cache 是物理寻址、直接映射（每组一行）、行大小 8 字节，共 8 个组；
7.TLB、Cache 的当前数值如表 1、2 所示。

表 1: TLB 数值表

| 组 | 标记位 | PPN | 有效位 | 标记位 | PPN | 有效位 | 标记位 | PPN | 有效位 | 标记位 | PPN | 有效位 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0CD | 09 | 1 | AA1 | 00 | 1 | 3E0 | 62 | 1 | C4C | 48 | 1 |
| 1 | 312 | 45 | 0 | 010 | 75 | 1 | 987 | 3A | 1 | D39 | 3F | 0 |
| 2 | 038 | E3 | 0 | 0A7 | 13 | 0 | 18B | 52 | 1 | 49B | 11 | 0 |
| 3 | 6C0 | 42 | 0 | 075 | 50 | 0 | 013 | 39 | 1 | 0F2 | 0D | 0 |

表 2: L1 d-Cache 的数值

| 索引 | 标记位 | 有效位 | 块 0 | 块 1 | 块 2 | 块 3 | 块 4 | 块 5 | 块 6 | 块 7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 35E | 1 | 42 | A0 | 75 | 50 | 42 | 00 | 05 | 50 |
| 1 | 27B | 1 | 08 | E3 | 00 | A7 | 13 | 00 | 88 | 52 |
| 2 | C54 | 1 | 3F | 75 | AB | 11 | 25 | 78 | 9A | 00 |
| 4 | A32 | 1 | 97 | 3A | 91 | D3 | 3F | 12 | 86 | 22 |
| 5 | C30 | 1 | 30 | 62 | 15 | 4C | 48 | A1 | 12 | 5C |
| 6 | B26 | 1 | 01 | 25 | 3E | 62 | 1F | C4 | 85 | 12 |
| 7 | 01A | 1 | 98 | 3A | 12 | D3 | 3F | 3C | 4D | 5E |

(1) VPN 的位数是 _____ 位，页表项 PTE 的总数量是 _____ 个。

(2) TLB 中，组索引是 _____ 位，标记 Tag 的位数是 _____ 位。

(3) 在 L1 d-Cache 中，块偏移的位数是 _____ 位，组索引的位数是 _____ 位，标记的位数是 _____ 位。

(4) CPU 从虚拟地址 0x7C0515 读取一个字的数值，将虚拟地址翻译成物理地址，并获取数值的过程中，写出下表中各参数对应的数值（8 分，每空 1 分）。

| 参数 | 数值 |
|---|---|
| 虚拟地址 | 0x7C0515 |
| 虚拟页号 VPN | _____ |
| TLB 索引 | _____ |
| TLB 标签 Tag | _____ |
| TLB 命中? (Y/N) | _____ |
| 物理页号 PPN | _____ |
| 物理地址 | _____ |
| Cache 命中? (Y/N) | _____ |
| 读取的数值 | _____ |

1. 1 页 2048B，是 $2^{11}$B，因此 VPN 的位数是 $24-11=13$ 位，PTE 的总数量是 $2^{13}=8192$。

2. TLB 中，组索引是 $\log_2(4)=2$ 位，Tag 的位数是 $13-2=11$ 位。

3. L1 d-Cache 中，块偏移位数是 $\log_2(8)=3$ 位，组索引位数是 $\log_2(8)=3$ 位，Tag 位数是 $19-3-3=13$ 位。

4. CPU 访问虚拟地址 `0x7C0515` 时：

| 参数 | 数值 |
|---|---|
| 虚拟地址 | `0x7C0515` |
| 虚拟页号 VPN | `0xF80` |
| TLB 索引 | `0x0` |
| TLB 标签 Tag | `0x3E0` |
| TLB 命中? | Y |
| 物理页号 PPN | `0x62` |
| 物理地址 | `0x31515` |
| Cache 命中? | Y |
| 读取的数值 | `0x9A78` |