测试题2 答案

1.B

根据程序中的代码，我们可以分析得出答案。

首先，定义了一个整型数组 `a`，并初始化为 `{ 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 }`。

然后，定义了一个指向数组 `a` 中第 6 个元素（即 `a[5]`）的指针 `p`。

接下来，通过 `p[7]` 访问了 `p` 所指向的位置向后偏移 7 个元素的值。也就是说，`p[7]` 相当于 `\*(p + 7)`。

由于 `p` 指向 `a[5]`，因此 `p + 7` 实际上是指向 `a[12]` 的位置。

所以，`b` 的值是 `a[12]`，即数组 `a` 中的第 13 个元素。

因此，答案是 B.13。

2.C

该程序中，定义了一个大小为 `MAX\_DATA\_SIZE` 的 unsigned char 类型的数组 `data`，并初始化为 `{ 1,2,3,4,5,6,7,8,9,0 }`。

然后，使用 `memcpy` 函数将 `data` 数组中从第 3 个元素开始的一半（也就是 5 个元素）复制到了数组中从第 1 个元素开始的位置上，即 `&data[2]`。

由于 `memcpy` 函数是按字节进行复制的，因此最终 `data` 数组中的数据如下所示：

{ 1,2,1,2,3,4,5,6,7,8 }

其中，`data[0]` 和 `data[1]` 分别为 1 和 2，而 `data[2]` 和 `data[3]` 则分别为 1 和 2，因为它们是从原数组的第 3 个元素开始复制的。而后面的元素则是原数组中对应的元素。

因此，最终 `data` 数组中的数据是 `{ 1,2,1,2,3,4,5,6,7,8,0 }`。

3.C

选项C错误。互斥锁的解锁过程需要按照加锁的顺序进行解锁，即先加锁的线程需要后解锁，后加锁的线程需要先解锁。如果解锁的顺序与加锁的顺序不一致，就可能导致死锁或者其他的并发问题。

因此，互斥锁的加锁和解锁必须由同一线程分别对应使用，并且互斥锁只能为0/1，用于线程的互斥。

4.B

通常在 Linux 中，查找某个文件中的字符串使用 `grep` 指令。`grep` 指令可以在文件中搜索指定的字符串，并将包含该字符串的行打印出来。

例如，要在文件 `example.txt` 中查找字符串 `hello`，可以使用以下命令：

```

grep "hello" example.txt

```

该命令会在 `example.txt` 文件中搜索字符串 `hello`，并将包含该字符串的行打印出来。

5.D

这种制约关系称之为\*\*同步\*\*。

在多进程或多线程的环境中，当多个进程或线程需要按照特定的顺序来访问某个共享资源时，需要使用同步机制来确保它们的执行顺序和访问顺序是正确的。

同步机制可以通过各种方式实现，例如使用互斥锁、条件变量、信号量等。它们可以协调进程或线程之间的执行顺序，以避免竞态条件和数据不一致等问题。同步机制的目的是保证多个进程或线程按照既定的顺序正确地进行访问和操作共享资源。因此，选项D "同步" 是正确的答案。

6.C

运行进程时间片用完，进程运行出错，运行进程阻塞（也就是等待某一事件发生）都会使操作系统选择新进程，但有新进程进入就绪状态不会影响其他进程状态变化。

1. A

A. 进程是操作系统分配资源的基本单位。

进程是操作系统中的一个重要概念，是程序在执行过程中的实体。每个进程都有自己的地址空间、寄存器集合和状态。操作系统通过进程来管理系统资源，如 CPU、内存、文件等。

进程是操作系统分配资源的基本单位，它可以被看作是一个独立的执行单元，具有独立的内存空间、代码段、数据段和堆栈等。操作系统为每个进程分配资源，并负责调度和管理它们的执行。

程序是指令的集合，是静态的，需要操作系统加载到内存中才能运行。线程是进程中的一个执行单元，是 CPU 调度的最小单位。指令是计算机执行的最基本单位，是 CPU 执行的基本操作。

因此，选项 A "进程" 是操作系统分配资源的基本单位。

8.B

该题主要考核 swith语句中没有break的情况下代码的运行

初始值为i=10

case 10:执行后： i++ ：i=11

case 11:执行后： i++ ：i=12

default:执行后： i++ : i=13

9.B

B. int a[2][3]={{0,1,2},{3,4,5}}; 是能对二维数组a进行完整初始化的语句。

选项 A 中的语句 ((0,1,2),(3,4,5)) 不是合法的二维数组初始化语句，应该使用大括号 {} 而不是小括号 ()。

选项 C 中的语句 {{0,1},{2,3},{4,5}} 是一个 3 行 2 列的二维数组，与 a 的大小不匹配，不能用于对 a 进行初始化。

选项 D 中的语句 int a[2][3]={0,1,2,3,4}; 只提供了 5 个元素的初始化值，而 a 是一个 2 行 3 列的数组，所以无法对 a 进行完整的初始化。

因此，选项 B int a[2][3]={{0,1,2},{3,4,5}}; 是能对二维数组a进行完整初始化的语句。

1. A

2\*3\*3+2+2\*3

11.B

根据给定的代码和信息，我们可以得出以下结论：

test\_p 是一个 int 类型的指针，它被赋值为 test 的地址，即 0x12345678。这意味着 test\_p 指向了 test 数组的起始位置。

test 数组是一个 char 类型的数组，每个元素占用一个字节。根据 C 语言的内存对齐规则，int 类型的指针在指针运算时会按照 int 类型的大小进行偏移。

因此，表达式 test\_p[1] 将会使 test\_p 偏移一个 int 类型的大小（4 字节），即指向 test 数组中的下一个 int 类型的位置。

由于 test 数组的起始地址是 0x12345678，偏移一个 int 类型的大小后，指针地址值将变为 0x1234567C。

因此，选项 B. 0x1234567C 是表达式 test\_p[1] 的指针地址值。

12.A

根据 C 语言中的运算符优先级，乘法运算符 `\*` 的优先级高于取模运算符 `%`，加法运算符 `+` 的优先级低于取模运算符 `%`。

因此，表达式 `4 \* f % 3 + 1` 的计算顺序为：

1. 计算 `4 \* f`，结果为 68。

2. 计算 `68 % 3`，结果为 2。

3. 计算 `2 + 1`，结果为 3。

最终将计算结果 3 赋值给变量 n。

因此，选项 A. 3 是程序运行后，变量 n 的值。

13.C

正确答案是 C. `flag&=~4`。

根据题目描述，我们需要将 flag 变量的第 2 位（从右往左数，下同）置为 0。可以使用按位与运算符 `&` 和按位取反运算符 `~` 来实现这个操作。具体步骤如下：

1. 构造一个只有第 2 位为 0，其他位为 1 的掩码，可以使用按位取反运算符 `~` 对值 4 进行取反得到：`~4`。

2. 将掩码与 flag 变量进行按位与运算，即 `flag & ~4`。

3. 按位与运算的结果是一个新的整数，它的第 2 位和 flag 变量的第 2 位相同，其他位都为 0。将这个新的整数赋值给 flag 变量即可。

因此，正确的操作是 `flag &= ~4`，它可以将 flag 变量的第 2 位置为 0。选项 A 中的 `flag |= 4` 操作会将 flag 变量的第 2 位置为 1；选项 B 中的 `flag ^= 4` 操作会将 flag 变量的第 2 位取反；选项 D 中的 `flag |= ~4` 操作会将 flag 变量的第 2 位及其后面的所有位都置为 1。

14.B

选项 B. `char test[10]; test="test";` 是错误的赋值方式。

在 C 语言中，字符数组是一种特殊的数组，用于存储字符串。正确的赋值方式是使用字符串相关的函数或者逐个赋值字符。

A. `char test[10]; strcpy(test, "test");` 是正确的赋值方式，使用了字符串拷贝函数 `strcpy` 将字符串 "test" 拷贝到字符数组 test 中。

B. `char test[10]; test = "test";` 是错误的赋值方式。在 C 语言中，字符数组名是一个常量指针，不能直接赋值为字符串常量。这里应该使用字符串拷贝函数 `strcpy` 来将字符串拷贝到字符数组中。

C. `char test[] = {'t', 'e', 's', 't'};` 是正确的赋值方式，通过逐个赋值字符的方式将字符数组 test 初始化为字符串 "test"。

D. `char test[10] = {'t', 'e', 's', 't'};` 是正确的赋值方式，通过逐个赋值字符的方式将字符数组 test 的前四个元素初始化为字符串 "test"，其余元素会被自动初始化为零值。

因此，选项 B. `char test[10]; test="test";` 是错误的赋值方式。

15.ABC

**解答：**

可用于中断下半段处理的处理机制有 A. tasklet、B. 工作队列和 C. 软中断。

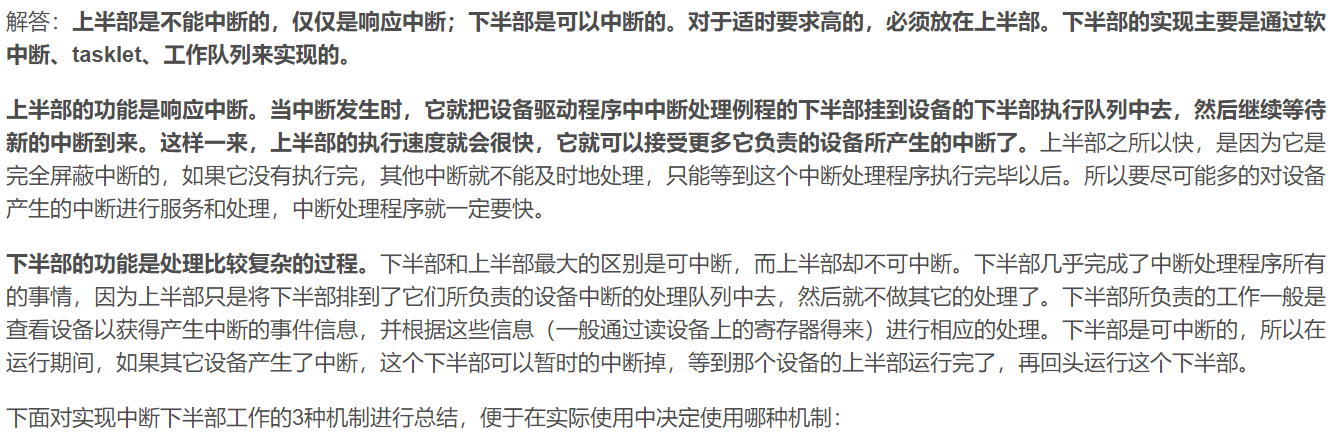
A. Tasklet 是一种轻量级的中断处理机制，用于在中断上下文中延迟执行工作。它可以在中断处理程序的上下文中调度，并在下一个调度点执行。Tasklet 通常用于处理短期且不需要睡眠的工作。

B. 工作队列（Work Queue）是一种异步执行工作的机制，用于将工作推迟到下半段处理。它可以在内核中的任何上下文中调度，并在适当的时候执行。工作队列可以用于处理较长时间的工作或需要睡眠的工作。

C. 软中断（Softirq）是一种在软件层面上模拟硬件中断的机制，用于在中断上下文中执行延迟的工作。软中断可以由内核的不同部分触发，并在适当的时候执行。软中断通常用于高优先级的任务，如网络数据包处理或定时器处理。

D. 双向链表（双向链表）不是一种用于中断下半段处理的处理机制，而是一种数据结构，用于在内核中组织和管理数据。

**解答(补充)：**





1. **填空题（共5空，共计10分）**
2. C语言中-15在计算机内存中8位二进制存放形式为\_\_\_**11110001**\_\_\_.和-15在计算机内存中存放方式一样的8位二进制数的十进制为\_\_\_\_241\_\_\_\_\_\_\_.
3. 在C语言中，如下程序输出结果为

int a = -2<<5;

printf("a = %d\n", a); \_\_\_\_\_\_\_-64\_\_\_\_\_\_\_\_

printf("a = %u\n", a); \_\_\_\_ 4294967232\_\_\_\_\_\_

1. 假设有一个寄存器a,要把a的第3bit变为0,其他bit位不变，该如何操作？ \_\_a&=~4\_\_\_\_\_
2. **问答题（共计2题，每题5分，共10分）**

**1、**请简要描述#include “” 和 #include <>的区别

在 C 语言中，`#include` 是一个预处理指令，用于将其他文件中的代码包含到当前文件中。在使用 `#include` 指令时，需要指定被包含文件的名称。

`#include` 指令有两种形式：`#include "filename"` 和 `#include <filename>`。

- `#include "filename"`：这种形式的 `#include` 指令会先在当前文件所在目录下查找指定的文件，如果找不到，则会在编译器指定的标准头文件目录中查找。通常用于包含用户自己编写的头文件。（2分）

- `#include <filename>`：这种形式的 `#include` 指令只会在编译器指定的标准头文件目录中查找指定的文件，不会在当前文件所在目录下查找。通常用于包含标准库的头文件。（2分）

总的来说，两种形式的 `#include` 指令都可以用于包含头文件，但是它们的搜索路径不同。（1分）使用 `#include "filename"` 可以先在当前文件所在目录下查找头文件，然后再在标准头文件目录中查找；而使用 `#include <filename>` 只会在标准头文件目录中查找头文件。因此，如果要包含用户自己编写的头文件，应该使用 `#include "filename"` 形式的指令。如果要包含标准库的头文件，应该使用 `#include <filename>` 形式的指令。

2、请比较标准IO和系统IO的区别

标准 I/O（Standard I/O）和系统 I/O（System I/O）是在编程中用于进行输入和输出操作的两种不同的方法。

1. 缓冲方式：（1分）

- 标准 I/O：使用缓冲区进行输入和输出操作。它将数据存储在内存缓冲区中，当缓冲区满时才进行实际的 I/O 操作。这种方式可以提高效率，减少系统调用的次数。

- 系统 I/O：直接进行输入和输出操作，不使用缓冲区。每次进行 I/O 操作时都会立即发送或接收数据。这种方式可以保证数据的实时性，但可能会增加系统调用的次数。

2. 数据传输方式：（1分）

- 标准 I/O：以流（stream）的方式进行数据传输，可以按照字符或行来读取或写入数据。

- 系统 I/O：以文件描述符（file descriptor）的方式进行数据传输，使用底层的系统调用函数（如 read() 和 write()）直接读取或写入二进制数据块。

3. 执行效率：（1分）

- 标准 I/O：由于使用了缓冲区，可以减少系统调用的次数，从而提高执行效率。适用于大量的、频繁的 I/O 操作。

- 系统 I/O：没有缓冲区的开销，每次 I/O 操作都是直接的系统调用，适用于对实时性要求较高的场景。

4. 可移植性：（1分）

- 标准 I/O：提供了一套统一的接口，不依赖于底层操作系统的具体实现，因此具有较好的可移植性。

- 系统 I/O：直接使用底层的系统调用函数，对于不同的操作系统可能存在差异，不太具有可移植性。

在实际编程中，可以根据具体需求选择使用标准 I/O 或系统 I/O。标准 I/O 适用于大多数常规的 I/O 操作，而系统 I/O 则适用于对实时性要求较高或需要直接控制底层的二进制数据传输的情况。（1分）

**四．编程题（共计2题，每题10分，共20分）**

(全对10分 语法问题8分 逻辑问题5分 语法问题+逻辑问题3分 空0分)

用两个栈来实现一个队列，使用n个元素来完成 n 次在队列尾部插入整数(push)和n次在队列头部删除整数(pop)的功能。 队列中的元素为int类型。保证操作合法，即保证pop操作时队列内已有元素。



要使用两个栈来实现一个队列，可以按照以下步骤进行操作：

定义两个栈，一个用于存储插入的元素，称为插入栈（push\_stack），另一个用于删除元素，称为删除栈（pop\_stack）。

当执行插入操作（push）时，将元素直接压入插入栈（push\_stack）。

当执行删除操作（pop）时，首先检查删除栈（pop\_stack）是否为空。如果不为空，则直接从删除栈的栈顶弹出一个元素并返回。如果删除栈为空，则将插入栈中的所有元素依次弹出并压入删除栈，然后再从删除栈的栈顶弹出一个元素并返回。

输入两个递增的链表，单个链表的长度为n，合并这两个链表并使新链表中的节点仍然是递增排序的

要合并两个递增的链表，并保持合并后的链表仍然是递增排序的，可以按照以下步骤进行操作：

创建一个新的链表，用于存储合并后的结果。

初始化三个指针，分别指向两个输入链表的头节点和合并后链表的当前节点。

比较两个输入链表的当前节点的值，将较小的节点添加到合并后的链表中，并将对应链表的指针向后移动一个节点。

重复步骤3，直到其中一个输入链表的指针到达链表末尾。

将另一个输入链表剩余的节点直接添加到合并后的链表的末尾。

返回合并后的链表。