1ARM指令和Thumb指令。（选择题）

解答：在ARM的体系结构中，可以工作在三种不同的状态，一是ARM状态，二是Thumb状态及Thumb-2状态，三是调试状态。而ARM状态和Thumb状态可以直接通过某些指令直接切换，都是在运行程序，只不过指令长度不一样而已。

ARM状态：arm处理器工作于32位指令的状态，所有指令均为32位；

Thumb状态：arm执行16位指令的状态，即16位状态；

thumb-2状态：这个状态是ARM7版本的ARM处理器所具有的新的状态，新的thumb-2内核技术兼有16位及32位指令，实现了更高的性能，更有效的功耗及更少地占用内存。总的来说，感觉这个状态除了兼有arm和thumb的优点外，还在这两种状态上有所提升，优化；

调试状态：处理器停机时进入调试状态。

也就是说：ARM状态，此时处理器执行32位的字对齐的ARM指令；Thumb状态，此时处理器执行16位的，半字对齐的THUMB指令。

ARM状态和Thumb状态切换程序：

从ARM到Thumb: LDR R0,=lable+1 BX R0（状态将寄存器的最低位设置为1，BX指令、R0指令将进入thumb状态）；

从ARM到Thumb: LDR R0,=lable BX R0（寄存器最低位设置为0，BX指令、R0指令将进入arm状态）。

当处理器进行异常处理时，则从异常向量地址开始执行，将自动进入ARM状态。

关于这个知识点还有几个注意点：

ARM处理器复位后开始执行代码时总是只处于ARM状态；

Cortex-M3只有Thumb-2状态和调试状态；

由于Thumb-2具有16位/32位指令功能，因此有了thumb-2就无需Thumb了。

另外，具有Thumb-2技术的ARM处理器也无需再ARM状态和Thumb-2状态间进行切换了，因为thumb-2具有32位指令功能。

2Linux的用户态与内核态的转换方法。（选择题）

解答：Linux下内核空间与用户空间进行通信的方式主要有syscall（system call）、procfs、ioctl和netlink等。

syscall：一般情况下，用户进程是不能访问内核的。它既不能访问内核所在的内存空间，也不能调用内核中的函数。Linux内核中设置了一组用于实现各种系统功能的子程序，用户可以通过调用他们访问linux内核的数据和函数，这些系统调用接口（SCI）称为系统调用；

procfs：是一种特殊的伪文件系统 ，是Linux内核信息的抽象文件接口，大量内核中的信息以及可调参数都被作为常规文件映射到一个目录树中，这样我们就可以简单直接的通过echo或cat这样的文件操作命令对系统信息进行查取；

netlink：用户态应用使用标准的 socket API 就可以使用 netlink 提供的强大功能；

ioctl：函数是文件结构中的一个属性分量，就是说如果你的驱动程序提供了对ioctl的支持，用户就可以在用户程序中使用ioctl函数控制设备的I/O通道。

3linux目录结构，选项是/usr、/tmp、/etc目录的作用。（选择题）

解答：linux目录图：

/usr：不是user的缩写，其实usr是Unix Software Resource的缩写， 也就是Unix操作系统软件资源所放置的目录，而不是用户的数据啦。这点要注意。 FHS建议所有软件开发者，应该将他们的数据合理的分别放置到这个目录下的次目录，而不要自行建立该软件自己独立的目录；

/tmp：这是让一般使用者或者是正在执行的程序暂时放置档案的地方。这个目录是任何人都能够存取的，所以你需要定期的清理一下。当然，重要资料不可放置在此目录啊。 因为FHS甚至建议在开机时，应该要将/tmp下的资料都删除；

/etc：系统主要的设定档几乎都放置在这个目录内，例如人员的帐号密码档、各种服务的启始档等等。 一般来说，这个目录下的各档案属性是可以让一般使用者查阅的，但是只有root有权力修改。 FHS建议不要放置可执行档(binary)在这个目录中。 比较重要的档案有：/etc/inittab, /etc/init.d/, /etc/modprobe.conf, /etc/X11/, /etc/fstab, /etc/sysconfig/等等。

4在32位系统中，有如下结构体，那么sizeof（fun）的数值是（）

#pragma pack(1)

struct fun{

int i;

double d;

char c;

};

解答：13。

可能是一般的内存对齐做习惯了，如果本题采用内存对齐的话，结果就是24（int 4 double char 7）。但是#pragma pack(1)让编译器将结构体数据强制按1来对齐。

每个特定平台上的编译器都有自己的默认“对齐系数”（32位机一般为4，64位机一般为8）。我们可以通过预编译命令#pragma pack(k)，k=1,2,4,8,16来改变这个系数，其中k就是需要指定的“对齐系数”。

只需牢记：

第一个数据成员放在offset为0的地方，对齐按照对齐系数和自身占用字节数中，二者比较小的那个进行对齐；

在数据成员完成各自对齐以后，struct或者union本身也要进行对齐，对齐将按照对齐系数和struct或者union中最大数据成员长度中比较小的那个进行；

5下面四个选项是四个整数在内存中的存储情况，请选择其中最大的一个。（选择题）

四个整数在内存的存储情况

A B C D

Big-endian

低地址 高地址

12 34 56 78

Big-endian

低地址 高地址

56 78 12 34

Little-endian

低地址 高地址

34 56 78 12

Little-endian

低地址 高地址

78 12 34 56

解答：大端小端问题：

所谓的大端模式（BE big-endian），是指数据的低位保存在内存的高地址中，而数据的高位，保存在内存的低地址中（低对高，高对高）；

所谓的小端模式（LE little-endian），是指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中（低对低，高对高）。

那么A：12345678、B：56781234、C：12785634、D：56341278。

6下面这段程序的运行结果？（填空题）

int main() {

int a[10] = { 0,1,2,3,4,5,6,7,8,9 };

memcpy(a + 3, a, 5);

for (int i = 0; i<10; i++){

printf("%d ", a[i]);

}

return 0;

}

解答：0 1 2 0 1 5 6 7 8 9

首先看一下内存复制函数memcpy()函数的定义：

void \* memcpy ( void \* destination, const void \* source, size\_t num );

将source指向的地址处的 num 个字节 拷贝到 destination 指向的地址处。注意，是字节。

因为memcpy的最后一个参数是需要拷贝的字节的数目！一个int类型占据4个字节！这样的话，本题5字节，实际上只能移动2个数字（往大的去）。如果要想达到将a地址开始的5个元素拷贝到a+3地址处，需要这么写：

memcpy(a + 3, a, 5\*sizeof(int));

7C语言编译过程中，volatile关键字和extern关键字分别在哪个阶段起作用？（填空题）

解答：volatile应该是在编译阶段，extern在链接阶段。

volatile关键字的作用是防止变量被编译器优化，而优化是处于编译阶段，所以volatile关键字是在编译阶段起作用。

8无锁可以提高整个程序的性能，但是CPU需要对此提供支持，请以x86/ARM为例简述。

解答：无锁编程具体使用和考虑到的技术方法包括：原子操作（atomic operations）, 内存栅栏（memory barriers）, 内存顺序冲突（memory order）， 指令序列一致性（sequential consistency）和顺ABA现象等等。

9已知两个已经按从小到大排列的数组，将它们中的所有数字组合成一个新的数组，要求新数组也是按照从小到大的顺序

void fun (int a[],int n,int b[],int m,int \*c)

{

int i,j;

for(i=j=0;i<n&&j<m;)

{

\*c++=a[i]<b[j]?a[i++]:b[j++];

}

while(i<n) \*c++=a[i++];

while(j<m) \*c++=b[j++];

10已知循环缓冲区是一个可以无限循环读写的缓冲区，当缓冲区满了还继续写的话就会覆盖我们还没读取到的数据。下面定义了一个循环缓冲区并初始化，请编写它的Write函数：

//循环缓冲区

typedef struct RingBuf {

char \*Buf;

unsigned int Size;

unsigned int RdId;

unsigned int WrId;

}RingBuf;

//采用循环队列的数据结构

int Init(RingBuf \*ringBuf,unsigned int size) {

if (ringBuf == NULL)

return -1;

char \*buf = malloc(sizeof(char) \* size);

if (!buf)

return -1;

ringBuf->Buf = buf;

ringBuf->Size = size;

ringBuf->RdId = 0;

ringBuf->WrId = 0;

return 0;

}

//当缓冲区满了还继续写的话就会覆盖我们还没读取到的数据

int write(RingBuf \*ringBuf, char data) {

if (ringBuf == NULL)

return -1;

ringBuf->Buf[ringBuf->WrId] = data;

ringBuf->WrId = (ringBuf->WrId + 1) % ringBuf->Size;

return 0;

}

int main() {

RingBuf ring;

Init(&ring, 10);

unsigned int i;

for (i = 0; i < 15; ++i)

write(&ring, i+'a');

for (i = 0; i < ring.Size; ++i)

printf("%c ", ring.Buf[i]);

free(ring.Buf);

return 0;

}

11我们平时在64位机上的指针大小也是4个字节，主要是为了兼容32位的机器，如果真的想让指针在64位系统机器占用8个字节，需要做某些设置如果面试的时候，你能把这个过程说出来给面试官说清楚，并且能讲提到虽然是64位系统，但是因为需要兼容32位程序，指针的大小还是 4字节的