分析下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **void** test1()  {  **char** string[10];  **char**\* str1 = "0123456789";  **strcpy**( string, str1 );  } |

# 参考答案

字符串str1需要11个字节才能存放下（包括末尾的’\0’），而string只有10个字节的空间，strcpy会导致数组越界；

分析下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **void** test2()  {  **char** string[10], str1[10];  **int** i;  **for**(i=0; i<10; i++)   {   str1  = 'a';   }  **strcpy**( string, str1 );  } |

# 参考答案

首先，代码根本不能通过编译。因为数组名str1为 char \*const类型的右值类型，根本不能赋值。

再者，即使想对数组的第一个元素赋值，也要使用 \*str1 = 'a';

其次，对字符数组赋值后，使用库函数strcpy进行拷贝操作，strcpy会从源地址一直往后拷贝，直到遇到'\0'为止。所以拷贝的长度是不定的。如果一直没有遇到'\0'导致越界访问非法内存，程序就崩了。

完美修改方案为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** test2()  {  **char** string[10], str1[10];  **int** i;  **for**(i=0; i<9; i++)      {          str1[i]  = 'a';      }      str1[9] = '\0';  **strcpy**( string, str1 );  } |

指出下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** test3(**char**\* str1)  {  **if**(str1 == NULL){  **return** ;   }  **char** string[10];  **if**( **strlen**( str1 ) <= 10 )   {  **strcpy**( string, str1 );   }  } |

# 参考答案

if(strlen(str1) <= 10)应改为if(strlen(str1) < 10)，因为strlen的结果未统计’\0’所占用的1个字节

写出完整版的strcpy函数

# 参考答案

如果编写一个标准strcpy函数的总分值为10，下面给出几个不同得分的答案：  
2分

|  |  |
| --- | --- |
| 1  2  3  4 | **void** **strcpy**( **char** \*strDest, **char** \*strSrc )  {  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  } |

4分

|  |  |
| --- | --- |
| 1  2  3  4  5 | **void** **strcpy**( **char** \*strDest, **const** **char** \*strSrc )  //将源字符串加const，表明其为输入参数，加2分  {  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  } |

7分

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **void** **strcpy**(**char** \*strDest, **const** **char** \*strSrc)  {   //对源地址和目的地址加非0断言，加3分  **assert**( (strDest != NULL) && (strSrc != NULL) );  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  } |

10分  
//为了实现链式操作，将目的地址返回，加3分！

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **char** \* **strcpy**( **char** \*strDest, **const** **char** \*strSrc )  {  **assert**( (strDest != NULL) && (strSrc != NULL) );  **char** \*address = strDest;  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  **return** address; |

|  |  |
| --- | --- |
| 1 | } |

检查下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** GetMemory( **char** \*p )  {   p = (**char** \*) **malloc**( 100 );  }  **void** Test( **void** )  {  **char** \*str = NULL;   GetMemory( str );  **strcpy**( str, "hello world" );  **printf**( str );  } |

# 参考答案

传入中GetMemory( char \*p )函数的形参为字符串指针，在函数内部修改形参并不能真正的改变传入形参的实参值，执行完

|  |  |
| --- | --- |
| 1  2 | **char** \*str = NULL;  GetMemory( str ); |

后的str仍然为NULL；

1：传入形参并不能真正改变形参的值，执行完之后为空；

2：在函数GetMemory中和Test中没有malloc对应的free，造成内存泄露

需要二级指针,才能改变

下面代码会出现什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **char** \*GetMemory( **void** )  {  **char** p[] = "hello world";  **return** p;  }  **void** Test( **void** )  {  **char** \*str = NULL;   str = GetMemory();  **printf**( str );  } |

# 参考答案

|  |  |
| --- | --- |
| 1  2 | **char** p[] = "hello world";  **return** p; |

的p[]数组为函数内的局部自动变量，在函数返回后，内存已经被释放。这是许多程序员常犯的错误，其根源在于不理解变量的生存期。

下面代码会出现什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** GetMemory( **char** \*\*p, **int** num )  {   \*p = (**char** \*) **malloc**( num );  }  **void** Test( **void** )  {  **char** \*str = NULL;   GetMemory( &str, 100 );  **strcpy**( str, "hello" );  **printf**( str );  } |

# 参考答案

1. 传入GetMemory的参数为字符串指针的指针，但是在GetMemory中执行申请内存及赋值语句

|  |  |
| --- | --- |
| 1 | \*p = (**char** \*) **malloc**( num ); |

后未判断内存是否申请成功，应加上：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **if** ( \*p == NULL )  {   ...//进行申请内存失败处理  }  同时应考虑num>0； |

2. 未释放堆内存 动态分配的内存在程序结束之前没有释放，应该调用free, 把malloc生成的内存释放掉

3. printf(str) 改为 printf("%s",str),否则可使用格式化 字符串攻击

下面代码会出现什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **void** Test( **void** )  {  **char** \*str = (**char** \*) **malloc**( 100 );  **strcpy**( str, "hello" );  **free**( str );   ... //省略的其它语句  } |

# 参考答案

在执行    
char \*str = (char \*) malloc(100);    
后未进行内存是否申请成功的判断；另外，在free(str)后未置str为空，导致可能变成一个“野”指针，应加上：    
str = NULL;    
试题6的Test函数中也未对malloc的内存进行释放。

看看下面的一段程序有什么错误?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | swap( **int**\* p1,**int**\* p2 )  {  **int** \*p;   \*p = \*p1;   \*p1 = \*p2;   \*p2 = \*p;  } |

# 参考答案

1.需要一个返回值void

2在swap函数中，p是一个“野”指针，有可能指向系统区，导致程序运行的崩溃。在VC++中DEBUG运行时提示错误“Access Violation”。该程序应该改为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **void** swap( **int**\* p1,**int**\* p2 )  {  **int** p;   p = \*p1;   \*p1 = \*p2;   \*p2 = p;  } |

分别给出BOOL，int，float，指针变量 与“零值”比较的 if 语句（假设变量名为var）

# 参考答案

【解答】

BOOL型变量：if(!var)

int型变量： if(var==0)

float型变量：

const float EPSINON = 0.00001;

if ((x >= - EPSINON) && (x <= EPSINON)

指针变量：　　if(var==NULL)

【剖析】

考查对0值判断的“内功”，BOOL型变量的0判断完全可以写成if(var==0)，而int型变量也可以写成if(!var)，指针变量的判断也可以写成if(!var)，上述写法虽然程序都能正确运行，但是未能清晰地表达程序的意思。

一般的，如果想让if判断一个变量的“真”、“假”，应直接使用if(var)、if(!var)，表明其为“逻辑”判断；如果用if判断一个数值型变量(short、int、long等)，应该用if(var==0)，表明是与0进行“数值”上的比较；而判断指针则适宜用if(var==NULL)，这是一种很好的编程习惯。

浮点型变量并不精确，所以不可将float变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。如果写成if (x == 0.0)，则判为错，得0分。

以下为Windows NT下的32位C++程序，请计算sizeof的值

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **void** Func ( **char** str[100] )  {  **sizeof**( str ) = ?  }  **void** \*p = **malloc**( 100 );  **sizeof** ( p ) = ? |

# 参考答案

sizeof( str ) = 4    
sizeof ( p ) = 4    
【剖析】    
Func ( char str[100] )函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。    
数组名的本质如下：    
（1）数组名指代一种数据结构，这种数据结构就是数组；    
例如：

|  |  |
| --- | --- |
| 1  2 | **char** str[10];  cout ＜＜ **sizeof**(str) ＜＜ endl; |

输出结果为10，str指代数据结构char[10]。    
（2）数组名可以转换为指向其指代实体的指针，而且是一个指针常量，不能作自增、自减等操作，不能被修改；    
char str[10];     
str++; //编译出错，提示str不是左值

（3）数组名作为函数形参时，沦为普通指针。

Windows NT 32位平台下，指针的长度（占用内存的大小）为4字节，故sizeof( str ) 、sizeof ( p ) 都为4。

写一个“标准”宏MIN，这个宏输入两个参数并返回较小的一个。另外，当你写下面的代码时会发生什么事？    
least = MIN(\*p++, b);

# 参考答案

解答：

|  |  |
| --- | --- |
| 1 | #define MIN(A,B) ((A) <= (B) ? (A) : (B)) |

MIN(\*p++, b)会产生宏的副作用    
剖析：    
这个面试题主要考查面试者对宏定义的使用，宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对“参数”进行的是一对一的替换。    
程序员对宏定义的使用要非常小心，特别要注意两个问题：    
（1）谨慎地将宏定义中的“参数”和整个宏用用括弧括起来。所以，严格地讲，下述解答：

|  |  |
| --- | --- |
| 1  2 | #define MIN(A,B) (A) <= (B) ? (A) : (B)  #define MIN(A,B) (A <= B ? A : B ) |

都应判0分；    
（2）防止宏的副作用。    
宏定义#define MIN(A,B) ((A) <= (B) ? (A) : (B))对MIN(\*p++, b)的作用结果是：

|  |  |
| --- | --- |
| 1 | ((\*p++) <= (b) ? (\*p++) : (b)) |

这个表达式会产生副作用，指针p会作2次++自增操作。

除此之外，另一个应该判0分的解答是：

|  |  |
| --- | --- |
| 1 | #define MIN(A,B) ((A) <= (B) ? (A) : (B)); |

这个解答在宏定义的后面加“;”，显示编写者对宏的概念模糊不清，只能被无情地判0分并被面试官淘汰。

为什么标准头文件都有类似以下的结构？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #ifndef \_\_INCvxWorksh  #define \_\_INCvxWorksh  #ifdef \_\_cplusplus  **extern** "C" {  #endif  /\*...\*/  #ifdef \_\_cplusplus  }  #endif  #endif /\* \_\_INCvxWorksh \*/ |

# 参考答案

头文件中的编译宏

|  |  |
| --- | --- |
| 1  2  3 | #ifndef　\_\_INCvxWorksh  #define　\_\_INCvxWorksh  #endif |

的作用是防止被重复引用。  
作为一种面向对象的语言，C++支持函数重载，而过程式语言C则不支持。函数被C++编译后在symbol库中的名字与C语言的不同。例如，假设某个函数的原型为：  
void foo(int x, int y);  
该函数被C编译器编译后在symbol库中的名字为\_foo，而C++编译器则会产生像\_foo\_int\_int之类的名字。\_foo\_int\_int这样的名字包含了函数名和函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。  
为了实现C和C++的混合编程，C++提供了C连接交换指定符号extern "C"来解决名字匹配问题，函数声明前加上extern "C"后，则编译器就会按照C语言的方式将该函数编译为\_foo，这样C语言中就可以调用C++的函数了

编写一个函数，作用是把一个char组成的字符串循环右移n个。比如原来是“abcdefghi”如果n=2，移位后应该是“hiabcdefg” 函数头是这样的：  
//pStr是指向以'\0'结尾的字符串的指针  
//steps是要求移动的n

|  |  |
| --- | --- |
| 1  2  3  4 | **void** LoopMove ( **char** \* pStr, **int** steps )  {   //请填充...  } |

# 参考答案

void LoopMove(char \*str, int steps)

{

int len = strlen(str);

char tmp[MAXSIZE];

strcpy(tmp, str+len-steps);

strcpy(tmp+steps, str);

\*(tmp+len)  = '/0';

strcpy(str, tmp);

}

或:

void LoopMove(char \*str, int steps)

{

int len = strlen(str);

char tmp[MAXSIZE];

memcpy(tmp, str+len-steps, steps);

memcpy(str+steps, str, len-steps);

memcpy(str, tmp, steps);

}

编写类String的构造函数、析构函数和赋值函数，已知类String的原型为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **class** String  {  **public**:   String(**const** **char** \*str = NULL); // 普通构造函数   String(**const** String &other); // 拷贝构造函数   ~ String(**void**); // 析构函数   String & operator =(**const** String &other); // 赋值函数  **private**:  **char** \*m\_data; // 用于保存字符串  }; |

# 参考答案

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | //普通构造函数  String::String(**const** **char** \*str)  {  **if**(str==NULL)   {   m\_data = **new** **char**[1]; // 得分点：对空字符串自动申请存放结束标志'\0'的空   //加分点：对m\_data加NULL 判断   \*m\_data = '\0';   }  **else**   {  **int** length = **strlen**(str);   m\_data = **new** **char**[length+1];  **strcpy**(m\_data, str);   }  }  // String的析构函数  String::~String(**void**)  {  **delete** [] m\_data; // 或delete m\_data;  }  //拷贝构造函数  String::String(**const** String &other) 　　　// 得分点：输入参数为const型  {  **int** length = **strlen**(other.m\_data);   m\_data = **new** **char**[length+1];  **strcpy**(m\_data, other.m\_data);  }  //赋值函数  String & String::operator =(**const** String &other) // 得分点：输入参数为const型  {  **if**(**this** == &other) 　　//得分点：检查自赋值  **return** \***this**;  **delete** [] m\_data; 　　　　//得分点：释放原有的内存资源  **int** length = **strlen**( other.m\_data );   m\_data = **new** **char**[length+1];  **strcpy**( m\_data, other.m\_data );  **return** \***this**; 　　　　　　　　//得分点：返回本对象的引用  } |

剖析  
能够准确无误地编写出String类的构造函数、拷贝构造函数、赋值函数和析构函数的面试者至少已经具备了C++基本功的60%以上！  
在这个类中包括了指针类成员变量m\_data，当类中包括指针类成员变量时，一定要重载其拷贝构造函数、赋值函数和析构函数，这既是对C++程序员的基本要求，也是《Effective　C++》中特别强调的条款。  
仔细学习这个类，特别注意加注释的得分点和加分点的意义，这样就具备了60%以上的C++基本功！

请说出static和const关键字尽可能多的作用

# 参考答案

【解答】  
static关键字至少有下列n个作用：    
（1）函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；    
（2）在模块内的static全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；    
（3）在模块内的static函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；    
（4）在类中的static成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；    
（5）在类中的static成员函数属于整个类所拥有，这个函数不接收this指针，因而只能访问类的static成员变量。     
const关键字至少有下列n个作用：    
（1）欲阻止一个变量被改变，可以使用const关键字。在定义该const变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；    
（2）对指针来说，可以指定指针本身为const，也可以指定指针所指的数据为const，或二者同时指定为const；    
（3）在一个函数声明中，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；    
（4）对于类的成员函数，若指定其为const类型，则表明其是一个常函数，不能修改类的 成员变量；    
（5）对于类的成员函数，有时候必须指定其返回值为const类型，以使得其返回值不为“左值”。例如：    
const classA operator\*(const classA& a1,const classA& a2);    
operator\*的返回结果必须是一个const对象。如果不是，这样的变态代码也不会编译出错：    
classA a, b, c;    
(a \* b) = c; // 对a\*b的结果赋值    
操作(a \* b) = c显然不符合编程者的初衷，也没有任何意义。

写一个函数返回1+2+3+…+n的值（假定结果不会超过长整型变量的范围）

# 参考答案

【解答】

|  |  |
| --- | --- |
| 1  2  3  4 | **int** Sum( **int** n )  {  **return** ( (**long**)1 + n) \* n / 2;　　//或return (1l + n) \* n / 2;  } |

【剖析】  
对于这个题，只能说，也许最简单的答案就是最好的答案。下面的解答，或者基于下面的解答思路去优化，不管怎么“折腾”，其效率也不可能与直接return ( 1 l + n ) \* n / 2相比！

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **int** Sum( **int** n )  {  **long** sum = 0;  **for**( **int** i=1; i<=n; i++ )   {   sum += i;   }  **return** sum;  } |

说一下static关键字的作用

参考答案

参考回答:

1. 全局静态变量

在全局变量前加上关键字static，全局变量就定义成一个全局静态变量.

静态存储区，在整个程序运行期间一直存在。

初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）；

作用域：全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾。

2. 局部静态变量

在局部变量之前加上关键字static，局部变量就成为一个局部静态变量。

内存中的位置：静态存储区

初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）；

作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变；

3. 静态函数

在函数返回类型前加static，函数就定义为静态函数。函数的定义和声明在默认情况下都是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

函数的实现使用static修饰，那么这个函数只可在本cpp内使用，不会同其他cpp中的同名函数引起冲突；

warning：不要再头文件中声明static的全局函数，不要在cpp内声明非static的全局函数，如果你要在多个cpp中复用该函数，就把它的声明提到头文件里去，否则cpp内部声明需加上static修饰；

4. 类的静态成员

在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类的所有对象中共享的成员，而不是某个对象的成员。对多个对象来说，静态数据成员只存储一处，供所有对象共用

5. 类的静态函数

静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。因此，对静态成员的引用不需要用对象名。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员（这点非常重要）。如果静态成员函数中要引用非静态成员时，可通过对象来引用。从中可看出，调用静态成员函数使用如下格式：<类名>::<静态成员函数名>(<参数表>);

说一下C++和C的区别

# 参考答案

参考回答:

设计思想上：

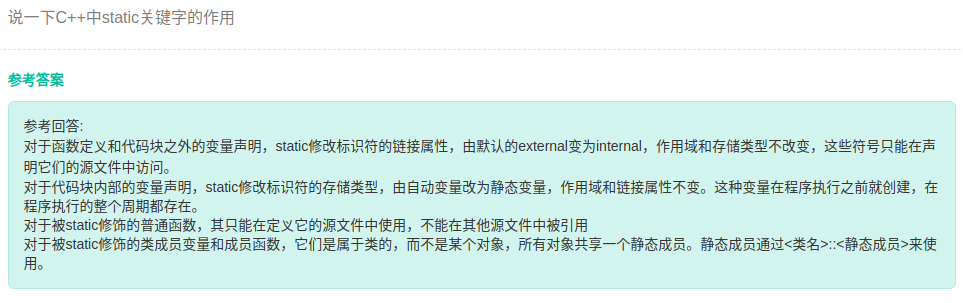
C++是面向对象的语言，而C是面向过程的结构化编程语言

语法上：

C++具有封装、继承和多态三种特性

C++相比C，增加多许多类型安全的功能，比如强制类型转换、

C++支持范式编程，比如模板类、函数模板等



说一说c++中四种cast转换

参考答案

参考回答:

C++中四种类型转换是：static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast

1、const\_cast

用于将const变量转为非const

2、static\_cast

用于各种隐式转换，比如非const转const，void\*转指针等, static\_cast能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

3、dynamic\_cast

用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。要深入了解内部转换的原理。

向上转换：指的是子类向基类的转换

向下转换：指的是基类向子类的转换

它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

4、reinterpret\_cast

几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；

5、为什么不使用C的强制转换？

C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

请说一下C/C++ 中指针和引用的区别？

参考回答:

1.指针有自己的一块空间，而引用只是一个别名；

2.使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；

3.指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象 的引用；

4.作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引 用的修改都会改变引用所指向的对象；

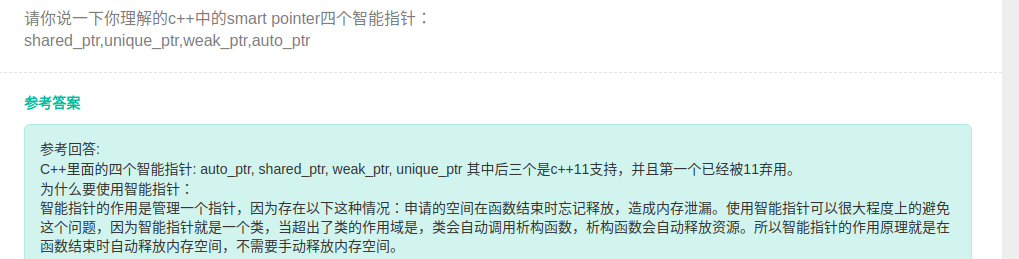
5.可以有const指针，但是没有const引用；

6.指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；

7.指针可以有多级指针（\*\*p），而引用至于一级；

8.指针和引用使用++运算符的意义不一样；

9.如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。



怎么判断一个数是二的倍数，怎么求一个数中有几个1，说一下你的思路并手写代码

参考答案

参考回答:

1、判断一个数是不是二的倍数，即判断该数二进制末位是不是0：

a % 2 == 0 或者a & 0x0001 == 0。

2、求一个数中1的位数，可以直接逐位除十取余判断：

int fun(long x)

{

int \_count = 0;

while(x)

{

if(x % 10 == 1)

++\_count;

x /= 10;

}

return \_count;

}

int main()

{

cout << fun(123321) << endl;

return 0;

}



请你回答一下野指针是什么？

# 参考答案

参考回答:

野指针就是指向一个已删除的对象或者未申请访问受限内存区域的指针

请你介绍一下C++中的智能指针

# 参考答案

参考回答:

智能指针主要用于管理在堆上分配的内存，它将普通的指针封装为一个栈对象。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。C++ 11中最常用的智能指针类型为shared\_ptr,它采用引用计数的方法，记录当前内存资源被多少个智能指针引用。该引用计数的内存在堆上分配。当新增一个时引用计数加1，当过期时引用计数减一。只有引用计数为0时，智能指针才会自动释放引用的内存资源。对shared\_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过make\_shared函数或者通过构造函数传入普通指针。并可以通过get函数获得普通指针

请你回答一下智能指针有没有内存泄露的情况

# 参考答案

参考回答:

当两个对象相互使用一个shared\_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄漏。

请你来说一下智能指针的内存泄漏如何解决

# 参考答案

参考回答:

为了解决循环引用导致的内存泄漏，引入了weak\_ptr弱指针，weak\_ptr的构造函数不会修改引用计数的值，从而不会对对象的内存进行管理，其类似一个普通指针，但不指向引用计数的共享内存，但是其可以检测到所管理的对象是否已经被释放，从而避免非法访问。

请你理解的c++中的引用和指针

# 参考答案

参考回答:

定义：

1、引用：

C++是C语言的继承，它可进行过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。引用就是C++对C语言的重要扩充。引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。引用的声明方法：类型标识符 &引用名=目标变量名；引用引入了对象的一个同义词。定义引用的表示方法与定义[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304" \t "/home/ts/Documents\\x/_blank)相似，只是用&代替了\*。

2、指针：

指针利用地址，它的值直接指向存在电脑存储器中另一个地方的值。由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元。因此，将地址形象化的称为“指针”。意思是通过它能找到以它为地址的内存单元。

区别：

1. 指针有自己的一块空间，而引用只是一个别名；2、使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；3、指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用；4、作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象；5、可以有const指针，但是没有const引用；6、指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；7、指针可以有多级指针（\*\*p），而引用至于一级；8、指针和引用使用++运算符的意义不一样；9、如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

请你回答一下为什么析构函数必须是虚函数？为什么C++默认的析构函数不是虚函数

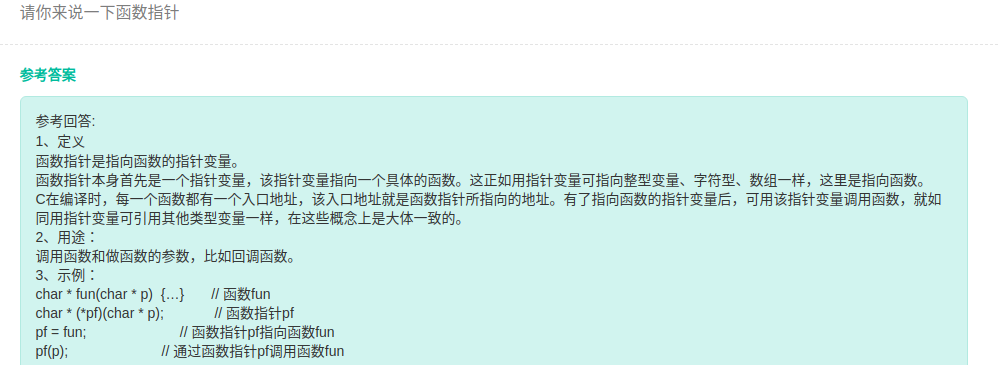
考点:虚函数 析构函数

# 参考答案

参考回答:

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄漏。

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是只有当需要当作父类时，设置为虚函数。



请你来说一下fork函数

# 参考答案

参考回答:

Fork：创建一个和当前进程映像一样的进程可以通过fork( )系统调用：

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

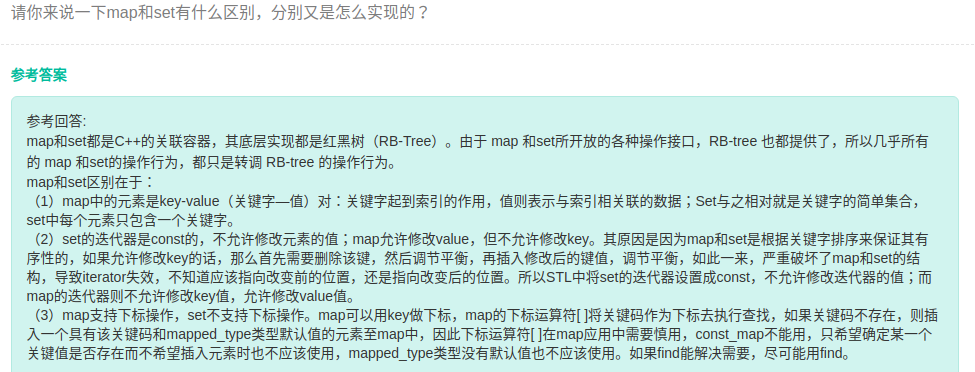
成功调用fork( )会创建一个新的进程，它几乎与调用fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork( )调用会返回0。在父进程中fork( )返回子进程的pid。如果出现错误，fork( )返回一个负值。 最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。 在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的Unix系统采取了更多的优化，例如Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

请你来说一下C++中析构函数的作用

# 参考答案

参考回答:

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。 析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。 如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。 如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。 类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。



41>

请你说一下进程与线程的概念，以及为什么要有进程线程，其中有什么区别，他们各自又是怎么同步的

**参考答案**

参考回答:

基本概念：

进程是对运行时程序的封装，是系统进行资源调度和分配的的基本单位，实现了操作系统的并发；

线程是进程的子任务，是CPU调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发；线程是操作系统可识别的最小执行和调度单位。每个线程都独自占用一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。每个线程完成不同的任务，但是共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件队列和其他内核资源。

区别：

1.一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程依赖于进程而存在。

2.进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。（资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。）

3.进程是资源分配的最小单位，线程是CPU调度的最小单位；

4.系统开销： 由于在创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I／o设备等。因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。类似地，在进行进程切换时，涉及到整个当前进程CPU环境的保存以及新被调度运行的进程的CPU环境的设置。而线程切换只须保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，进程切换的开销也远大于线程切换的开销。

5.通信：由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现，也变得比较容易。进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。在有的系统中，线程的切换、同步和通信都无须操作系统内核的干预

6.进程编程调试简单可靠性高，但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。

7.进程间不会相互影响 ；线程一个线程挂掉将导致整个进程挂掉

8.进程适应于多核、多机分布；线程适用于多核

进程间通信的方式：

进程间通信主要包括管道、系统IPC（包括消息队列、信号量、信号、共享内存等）、以及套接字socket。

1.管道：

管道主要包括无名管道和命名管道:管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信

1.1 普通管道PIPE：

1)它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端

2)它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）

3)它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

1.2 命名管道FIFO：

1)FIFO可以在无关的进程之间交换数据

2)FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2. 系统IPC：

2.1 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标记。 (消息队列克服了信号传递信息少，管道只能承载无格式字节流以及缓冲区大小受限等特点)具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息；

特点：

1)消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

2)消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

3)消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

2.2 信号量semaphore

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器，可以用来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

特点：

1)信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

2)信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

3)每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

4)支持信号量组。

2.3 信号signal

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

2.4 共享内存（Shared Memory）

它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等

特点：

1)共享内存是最快的一种IPC，因为进程是直接对内存进行存取

2)因为多个进程可以同时操作，所以需要进行同步

3)信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问

3.套接字SOCKET：

socket也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机之间的进程通信。

线程间通信的方式:

临界区：通过多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问；

互斥量Synchronized/Lock：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问

信号量Semphare：为控制具有有限数量的用户资源而设计的，它允许多个线程在同一时刻去访问同一个资源，但一般需要限制同一时刻访问此资源的最大线程数目。

事件(信号)，Wait/Notify：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作

请你说一说Linux虚拟地址空间

**参考答案**

参考回答:

为了防止不同进程同一时刻在物理内存中运行而对物理内存的争夺和践踏，采用了虚拟内存。

虚拟内存技术使得不同进程在运行过程中，它所看到的是自己独自占有了当前系统的4G内存。所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。 事实上，在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，具体就是初始化进程控制表中内存相关的链表，实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射就好（叫做存储器映射），等到运行到对应的程序时，才会通过缺页异常，来拷贝数据。还有进程运行过程中，要动态分配内存，比如malloc时，也只是分配了虚拟内存，即为这块虚拟内存对应的页表项做相应设置，当进程真正访问到此数据时，才引发缺页异常。

请求分页系统、请求分段系统和请求段页式系统都是针对虚拟内存的，通过请求实现内存与外存的信息置换。

虚拟内存的好处：

1.扩大地址空间；

2.内存保护：每个进程运行在各自的虚拟内存地址空间，互相不能干扰对方。虚存还对特定的内存地址提供写保护，可以防止代码或数据被恶意篡改。

3.公平内存分配。采用了虚存之后，每个进程都相当于有同样大小的虚存空间。

4.当进程通信时，可采用虚存共享的方式实现。

5.当不同的进程使用同样的代码时，比如库文件中的代码，物理内存中可以只存储一份这样的代码，不同的进程只需要把自己的虚拟内存映射过去就可以了，节省内存

6.虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中。当一个程序等待它的一部分读入内存时，可以把CPU交给另一个进程使用。在内存中可以保留多个进程，系统并发度提高

7.在程序需要分配连续的内存空间的时候，只需要在虚拟内存空间分配连续空间，而不需要实际物理内存的连续空间，可以利用碎片

虚拟内存的代价：

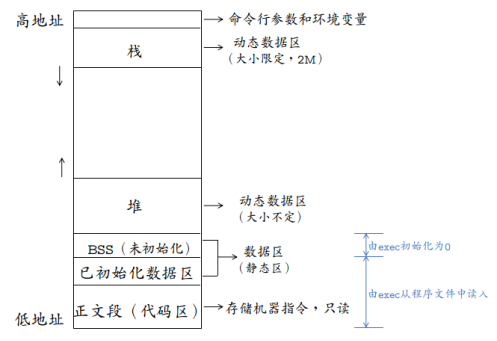
1.虚存的管理需要建立很多数据结构，这些数据结构要占用额外的内存

2.虚拟地址到物理地址的转换，增加了指令的执行时间。

3.页面的换入换出需要磁盘I/O，这是很耗时的

4.如果一页中只有一部分数据，会浪费内存。

请你说一说操作系统中的程序的内存结构



一个程序本质上都是由BSS段、data段、text段三个组成的。可以看到一个可执行程序在存储（没有调入内存）时分为代码段、数据区和未初始化数据区三部分。

BSS段（未初始化数据区）：通常用来存放程序中未初始化的全局变量和静态变量的一块内存区域。BSS段属于静态分配，程序结束后静态变量资源由系统自动释放。

数据段：存放程序中已初始化的全局变量的一块内存区域。数据段也属于静态内存分配

代码段：存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域属于只读。在代码段中，也有可能包含一些只读的常数变量

text段和data段在编译时已经分配了空间，而BSS段并不占用可执行文件的大小，它是由链接器来获取内存的。

bss段（未进行初始化的数据）的内容并不存放在磁盘上的程序文件中。其原因是内核在程序开始运行前将它们设置为0。需要存放在程序文件中的只有正文段和初始化数据段。

data段（已经初始化的数据）则为数据分配空间，数据保存到目标文件中。

数据段包含经过初始化的全局变量以及它们的值。BSS段的大小从可执行文件中得到，然后链接器得到这个大小的内存块，紧跟在数据段的后面。当这个内存进入程序的地址空间后全部清零。包含数据段和BSS段的整个区段此时通常称为数据区。

可执行程序在运行时又多出两个区域：栈区和堆区。

栈区：由编译器自动释放，存放函数的参数值、局部变量等。每当一个函数被调用时，该函数的返回类型和一些调用的信息被存放到栈中。然后这个被调用的函数再为他的自动变量和临时变量在栈上分配空间。每调用一个函数一个新的栈就会被使用。栈区是从高地址位向低地址位增长的，是一块连续的内存区域，最大容量是由系统预先定义好的，申请的栈空间超过这个界限时会提示溢出，用户能从栈中获取的空间较小。

堆区：用于动态分配内存，位于BSS和栈中间的地址区域。由程序员申请分配和释放。堆是从低地址位向高地址位增长，采用链式存储结构。频繁的malloc/free造成内存空间的不连续，产生碎片。当申请堆空间时库函数是按照一定的算法搜索可用的足够大的空间。因此堆的效率比栈要低的多。

请你说一说操作系统中的缺页中断

**参考答案**

参考回答:

malloc()和mmap()等内存分配函数，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页异常。

缺页中断：在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。每当所要访问的页面不在内存是，会产生一次缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。

缺页本身是一种中断，与一般的中断一样，需要经过4个处理步骤：

1、保护CPU现场

2、分析中断原因

3、转入缺页中断处理程序进行处理

4、恢复CPU现场，继续执行

但是缺页中断是由于所要访问的页面不存在于内存时，由硬件所产生的一种特殊的中断，因此，与一般的中断存在区别：

1、在指令执行期间产生和处理缺页中断信号

2、一条指令在执行期间，可能产生多次缺页中断

3、缺页中断返回是，执行产生中断的一条指令，而一般的中断返回是，执行下一条指令。

请你回答一下fork和vfork的区别

**参考答案**

参考回答:

fork的基础知识：

fork:创建一个和当前进程映像一样的进程可以通过fork( )系统调用：

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

成功调用fork( )会创建一个新的进程，它几乎与调用fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork( )调用会返回0。在父进程中fork( )返回子进程的pid。如果出现错误，fork( )返回一个负值。

最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的Unix系统采取了更多的优化，例如Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

vfork的基础知识：

在实现写时复制之前，Unix的设计者们就一直很关注在fork后立刻执行exec所造成的地址空间的浪费。BSD的开发者们在3.0的BSD系统中引入了vfork( )系统调用。

#include <sys/types.h>

#include <unistd.h>

pid\_t vfork(void);

除了子进程必须要立刻执行一次对exec的系统调用，或者调用\_exit( )退出，对vfork( )的成功调用所产生的结果和fork( )是一样的。vfork( )会挂起父进程直到子进程终止或者运行了一个新的可执行文件的映像。通过这样的方式，vfork( )避免了地址空间的按页复制。在这个过程中，父进程和子进程共享相同的地址空间和页表项。实际上vfork( )只完成了一件事：复制内部的内核数据结构。因此，子进程也就不能修改地址空间中的任何内存。

vfork( )是一个历史遗留产物，Linux本不应该实现它。需要注意的是，即使增加了写时复制，vfork( )也要比fork( )快，因为它没有进行页表项的复制。然而，写时复制的出现减少了对于替换fork( )争论。实际上，直到2.2.0内核，vfork( )只是一个封装过的fork( )。因为对vfork( )的需求要小于fork( )，所以vfork( )的这种实现方式是可行的。

补充知识点：写时复制

Linux采用了写时复制的方法，以减少fork时对父进程空间进程整体复制带来的开销。

写时复制是一种采取了惰性优化方法来避免复制时的系统开销。它的前提很简单：如果有多个进程要读取它们自己的那部门资源的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”，就存在着这样的幻觉：每个进程好像独占那个资源。从而就避免了复制带来的负担。如果一个进程要修改自己的那份资源“副本”，那么就会复制那份资源，并把复制的那份提供给进程。不过其中的复制对进程来说是透明的。这个进程就可以修改复制后的资源了，同时其他的进程仍然共享那份没有修改过的资源。所以这就是名称的由来：在写入时进行复制。

写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。

在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在fork( )调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。

写时复制在内核中的实现非常简单。与内核页相关的数据结构可以被标记为只读和写时复制。如果有进程试图修改一个页，就会产生一个缺页中断。内核处理缺页中断的方式就是对该页进行一次透明复制。这时会清除页面的COW属性，表示着它不再被共享。

现代的计算机系统结构中都在内存管理单元（MMU）提供了硬件级别的写时复制支持，所以实现是很容易的。

在调用fork( )时，写时复制是有很大优势的。因为大量的fork之后都会跟着执行exec，那么复制整个父进程地址空间中的内容到子进程的地址空间完全是在浪费时间：如果子进程立刻执行一个新的二进制可执行文件的映像，它先前的地址空间就会被交换出去。写时复制可以对这种情况进行优化。

fork和vfork的区别：

1. fork( )的子进程拷贝父进程的数据段和代码段；vfork( )的子进程与父进程共享数据段

2. fork( )的父子进程的执行次序不确定；vfork( )保证子进程先运行，在调用exec或exit之前与父进程数据是共享的，在它调用exec或exit之后父进程才可能被调度运行。

3. vfork( )保证子进程先运行，在它调用exec或exit之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

4.当需要改变共享数据段中变量的值，则拷贝父进程。

请你说一说并发(concurrency)和并行(parallelism)

**参考答案**

参考回答:

并发（concurrency）：指宏观上看起来两个程序在同时运行，比如说在单核cpu上的多任务。但是从微观上看两个程序的指令是交织着运行的，你的指令之间穿插着我的指令，我的指令之间穿插着你的，在单个周期内只运行了一个指令。这种并发并不能提高计算机的性能，只能提高效率。

并行（parallelism）：指严格物理意义上的同时运行，比如多核cpu，两个程序分别运行在两个核上，两者之间互不影响，单个周期内每个程序都运行了自己的指令，也就是运行了两条指令。这样说来并行的确提高了计算机的效率。所以现在的cpu都是往多核方面发展。

请你说一说有了进程，为什么还要有线程？

**参考答案**

参考回答:

线程产生的原因：

进程可以使多个程序能并发执行，以提高资源的利用率和系统的吞吐量；但是其具有一些缺点：

进程在同一时间只能干一件事

进程在执行的过程中如果阻塞，整个进程就会挂起，即使进程中有些工作不依赖于等待的资源，仍然不会执行。

因此，操作系统引入了比进程粒度更小的线程，作为并发执行的基本单位，从而减少程序在并发执行时所付出的时空开销，提高并发性。和进程相比，线程的优势如下：

从资源上来讲，线程是一种非常"节俭"的多任务操作方式。在linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。

从切换效率上来讲，运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的30倍左右。（

从通信机制上来讲，线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进城下的线程之间贡献数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。

除以上优点外，多线程程序作为一种多任务、并发的工作方式，还有如下优点：

1、使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

2、改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序才会利于理解和修改。

请问单核机器上写多线程程序，是否需要考虑加锁，为什么？

**参考答案**

参考回答:

在单核机器上写多线程程序，仍然需要线程锁。因为线程锁通常用来实现线程的同步和通信。在单核机器上的多线程程序，仍然存在线程同步的问题。因为在抢占式操作系统中，通常为每个线程分配一个时间片，当某个线程时间片耗尽时，操作系统会将其挂起，然后运行另一个线程。如果这两个线程共享某些数据，不使用线程锁的前提下，可能会导致共享数据修改引起冲突。

请问线程需要保存哪些上下文，SP、PC、EAX这些寄存器是干嘛用的

**参考答案**

参考回答:

线程在切换的过程中需要保存当前线程Id、线程状态、堆栈、寄存器状态等信息。其中寄存器主要包括SP PC EAX等寄存器，其主要功能如下：

SP:堆栈指针，指向当前栈的栈顶地址

PC:程序计数器，存储下一条将要执行的指令

EAX:累加寄存器，用于加法乘法的缺省寄存器

请你说一说线程间的同步方式，最好说出具体的系统调用

**参考答案**

参考回答:

信号量

信号量是一种特殊的变量，可用于线程同步。它只取自然数值，并且只支持两种操作：

P(SV):如果信号量SV大于0，将它减一；如果SV值为0，则挂起该线程。

V(SV)：如果有其他进程因为等待SV而挂起，则唤醒，然后将SV+1；否则直接将SV+1。

其系统调用为：

sem\_wait（sem\_t \*sem）：以原子操作的方式将信号量减1，如果信号量值为0，则sem\_wait将被阻塞，直到这个信号量具有非0值。

sem\_post（sem\_t \*sem)：以原子操作将信号量值+1。当信号量大于0时，其他正在调用sem\_wait等待信号量的线程将被唤醒。

互斥量

互斥量又称互斥锁，主要用于线程互斥，不能保证按序访问，可以和条件锁一起实现同步。当进入临界区      时，需要获得互斥锁并且加锁；当离开临界区时，需要对互斥锁解锁，以唤醒其他等待该互斥锁的线程。其主要的系统调用如下：

pthread\_mutex\_init:初始化互斥锁

pthread\_mutex\_destroy：销毁互斥锁

pthread\_mutex\_lock：以原子操作的方式给一个互斥锁加锁，如果目标互斥锁已经被上锁，pthread\_mutex\_lock调用将阻塞，直到该互斥锁的占有者将其解锁。

pthread\_mutex\_unlock:以一个原子操作的方式给一个互斥锁解锁。

条件变量

条件变量，又称条件锁，用于在线程之间同步共享数据的值。条件变量提供一种线程间通信机制：当某个共享数据达到某个值时，唤醒等待这个共享数据的一个/多个线程。即，当某个共享变量等于某个值时，调用 signal/broadcast。此时操作共享变量时需要加锁。其主要的系统调用如下：

pthread\_cond\_init:初始化条件变量

pthread\_cond\_destroy：销毁条件变量

pthread\_cond\_signal：唤醒一个等待目标条件变量的线程。哪个线程被唤醒取决于调度策略和优先级。

pthread\_cond\_wait：等待目标条件变量。需要一个加锁的互斥锁确保操作的原子性。该函数中在进入wait状态前首先进行解锁，然后接收到信号后会再加锁，保证该线程对共享资源正确访问。

请你说一下多线程和多进程的不同

**参考答案**

参考回答:

进程是资源分配的最小单位，而线程时CPU调度的最小单位。多线程之间共享同一个进程的地址空间，线程间通信简单，同步复杂，线程创建、销毁和切换简单，速度快，占用内存少，适用于多核分布式系统，但是线程间会相互影响，一个线程意外终止会导致同一个进程的其他线程也终止，程序可靠性弱。而多进程间拥有各自独立的运行地址空间，进程间不会相互影响，程序可靠性强，但是进程创建、销毁和切换复杂，速度慢，占用内存多，进程间通信复杂，但是同步简单，适用于多核、多机分布。

请你说一说进程和线程的区别

**参考答案**

参考回答:

1）进程是cpu资源分配的最小单位，线程是cpu调度的最小单位。

2）进程有独立的系统资源，而同一进程内的线程共享进程的大部分系统资源,包括堆、代码段、数据段，每个线程只拥有一些在运行中必不可少的私有属性，比如tcb,线程Id,栈、寄存器。

3）一个进程崩溃，不会对其他进程产生影响；而一个线程崩溃，会让同一进程内的其他线程也死掉。

4）进程在创建、切换和销毁时开销比较大，而线程比较小。进程创建的时候需要分配系统资源，而销毁的的时候需要释放系统资源。进程切换需要分两步：切换页目录、刷新TLB以使用新的地址空间；切换内核栈和硬件上下文（寄存器）；而同一进程的线程间逻辑地址空间是一样的，不需要切换页目录、刷新TLB。

5）进程间通信比较复杂，而同一进程的线程由于共享代码段和数据段，所以通信比较容易。

游戏服务器应该为每个用户开辟一个线程还是一个进程，为什么？

**参考答案**

参考回答:

游戏服务器应该为每个用户开辟一个进程。因为同一进程间的线程会相互影响，一个线程死掉会影响其他线程，从而导致进程崩溃。因此为了保证不同用户之间不会相互影响，应该为每个用户开辟一个进程

请你说一说OS缺页置换算法

**参考答案**

参考回答:

当访问一个内存中不存在的页，并且内存已满，则需要从内存中调出一个页或将数据送至磁盘对换区，替换一个页，这种现象叫做缺页置换。当前操作系统最常采用的缺页置换算法如下：

先进先出(FIFO)算法：置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。

最近最少使用（LRU）算法: 置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。

当前最常采用的就是LRU算法。

请你说一说死锁发生的条件以及如何解决死锁

**参考答案**

参考回答:

死锁是指两个或两个以上进程在执行过程中，因争夺资源而造成的下相互等待的现象。死锁发生的四个必要条件如下：

互斥条件：进程对所分配到的资源不允许其他进程访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源；

请求和保持条件：进程获得一定的资源后，又对其他资源发出请求，但是该资源可能被其他进程占有，此时请求阻塞，但该进程不会释放自己已经占有的资源

不可剥夺条件：进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用后自己释放

环路等待条件：进程发生死锁后，必然存在一个进程-资源之间的环形链

解决死锁的方法即破坏上述四个条件之一，主要方法如下：

资源一次性分配，从而剥夺请求和保持条件

可剥夺资源：即当进程新的资源未得到满足时，释放已占有的资源，从而破坏不可剥夺的条件

资源有序分配法：系统给每类资源赋予一个序号，每个进程按编号递增的请求资源，释放则相反，从而破坏环路等待的条件

请你说一说操作系统中的结构体对齐，字节对齐

**参考答案**

参考回答:

1、原因：

1）平台原因（移植原因）：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2）性能原因：数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

2、规则

1）数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。

2）结构(或联合)的整体对齐规则：在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

3）结构体作为成员：如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储。

3、定义结构体对齐

可以通过预编译命令#pragma pack(n)，n=1,2,4,8,16来改变这一系数，其中的n就是指定的“对齐系数”。

4、举例

#pragma pack(2)

struct AA {

int a;       //长度4 > 2 按2对齐；偏移量为0；存放位置区间[0,3]

char b;  //长度1 < 2 按1对齐；偏移量为4；存放位置区间[4]

short c;     //长度2 = 2 按2对齐；偏移量要提升到2的倍数6；存放位置区间[6,7]

char d;  //长度1 < 2 按1对齐；偏移量为7；存放位置区间[8]；共九个字节

};

#pragma pack()

请问进程间怎么通信

**参考答案**

参考回答:

进程间通信主要包括管道、系统IPC（包括消息队列、信号量、信号、共享内存等）、以及套接字socket。

1.管道：

管道主要包括无名管道和命名管道:管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信

1.1 普通管道PIPE：

1）它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端

2）它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）

3）它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

1.2 命名管道FIFO：

1）FIFO可以在无关的进程之间交换数据

2）FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2. 系统IPC：

2.1 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标记。 (消息队列克服了信号传递信息少，管道只能承载无格式字节流以及缓冲区大小受限等特点)具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息；

特点：

1）消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

2）消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

3）消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

2.2 信号量semaphore

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器，可以用来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

特点：

1）信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

2）信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

3）每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

4）支持信号量组。

2.3 信号signal

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

2.4 共享内存（Shared Memory）

它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等

特点：

1）共享内存是最快的一种IPC，因为进程是直接对内存进行存取

2）因为多个进程可以同时操作，所以需要进行同步

3）信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问

3. 套接字SOCKET：

socket也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机之间的进程通信。

请你说一下虚拟内存置换的方式

**参考答案**

参考回答:

比较常见的内存替换算法有：FIFO，LRU，LFU，LRU-K，2Q。

1、FIFO（先进先出淘汰算法）

思想：最近刚访问的，将来访问的可能性比较大。

实现：使用一个队列，新加入的页面放入队尾，每次淘汰队首的页面，即最先进入的数据，最先被淘汰。

弊端：无法体现页面冷热信息

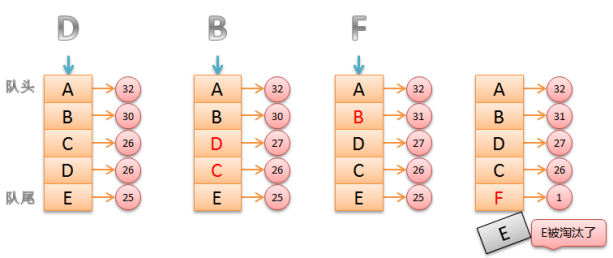
2、LFU（最不经常访问淘汰算法）

思想：如果数据过去被访问多次，那么将来被访问的频率也更高。

实现：每个数据块一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序。每次淘汰队尾数据块。

开销：排序开销。

弊端：缓存颠簸。



3、LRU（最近最少使用替换算法）

思想：如果数据最近被访问过，那么将来被访问的几率也更高。

实现：使用一个栈，新页面或者命中的页面则将该页面移动到栈底，每次替换栈顶的缓存页面。

优点：LRU算法对热点数据命中率是很高的。

缺陷：

1）缓存颠簸，当缓存（1，2，3）满了，之后数据访问（0，3，2，1，0，3，2，1。。。）。

2）缓存污染，突然大量偶发性的数据访问，会让内存中存放大量冷数据。

4、LRU-K（LRU-2、LRU-3）

思想：最久未使用K次淘汰算法。

LRU-K中的K代表最近使用的次数，因此LRU可以认为是LRU-1。LRU-K的主要目的是为了解决LRU算法“缓存污染”的问题，其核心思想是将“最近使用过1次”的判断标准扩展为“最近使用过K次”。

相比LRU，LRU-K需要多维护一个队列，用于记录所有缓存数据被访问的历史。只有当数据的访问次数达到K次的时候，才将数据放入缓存。当需要淘汰数据时，LRU-K会淘汰第K次访问时间距当前时间最大的数据。

实现：

1）数据第一次被访问，加入到访问历史列表；

2）如果数据在访问历史列表里后没有达到K次访问，则按照一定规则（FIFO，LRU）淘汰；

3）当访问历史队列中的数据访问次数达到K次后，将数据索引从历史队列删除，将数据移到缓存队列中，并缓存此数据，缓存队列重新按照时间排序；

4）缓存数据队列中被再次访问后，重新排序；

5）需要淘汰数据时，淘汰缓存队列中排在末尾的数据，即：淘汰“倒数第K次访问离现在最久”的数据。

针对问题：

LRU-K的主要目的是为了解决LRU算法“缓存污染”的问题，其核心思想是将“最近使用过1次”的判断标准扩展为“最近使用过K次”。

5、2Q

类似LRU-2。使用一个FIFO队列和一个LRU队列。

实现：

1）新访问的数据插入到FIFO队列；

2）如果数据在FIFO队列中一直没有被再次访问，则最终按照FIFO规则淘汰；

3）如果数据在FIFO队列中被再次访问，则将数据移到LRU队列头部；

4）如果数据在LRU队列再次被访问，则将数据移到LRU队列头部；

5）LRU队列淘汰末尾的数据。

针对问题：LRU的缓存污染

弊端：

当FIFO容量为2时，访问负载是：ABCABCABC会退化为FIFO，用不到LRU。

请你说一下多线程，线程同步的几种方式

**参考答案**

参考回答:

概念：

进程是对运行时程序的封装，是系统进行资源调度和分配的的基本单位，实现了操作系统的并发；

线程是进程的子任务，是CPU调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发；线程是操作系统可识别的最小执行和调度单位。每个线程都独自占用一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。每个线程完成不同的任务，但是共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件队列和其他内核资源。

线程间通信的方式:

1、临界区：

通过多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问；

2、互斥量 Synchronized/Lock：

采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问

3、信号量 Semphare：

为控制具有有限数量的用户资源而设计的，它允许多个线程在同一时刻去访问同一个资源，但一般需要限制同一时刻访问此资源的最大线程数目。

4、事件(信号)，Wait/Notify：

通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作

请你讲述一下互斥锁（mutex）机制，以及互斥锁和读写锁的区别

**参考答案**

参考回答:

1、互斥锁和读写锁区别：

互斥锁：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。

读写锁：rwlock，分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。 注意：写锁会阻塞其它读写锁。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

互斥锁和读写锁的区别：

1）读写锁区分读者和写者，而互斥锁不区分

2）互斥锁同一时间只允许一个线程访问该对象，无论读写；读写锁同一时间内只允许一个写者，但是允许多个读者同时读对象。

2、Linux的4种锁机制：

互斥锁：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒

读写锁：rwlock，分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。 注意：写锁会阻塞其它读写锁。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

自旋锁：spinlock，在任何时刻同样只能有一个线程访问对象。但是当获取锁操作失败时，不会进入睡眠，而是会在原地自旋，直到锁被释放。这样节省了线程从睡眠状态到被唤醒期间的消耗，在加锁时间短暂的环境下会极大的提高效率。但如果加锁时间过长，则会非常浪费CPU资源。

RCU：即read-copy-update，在修改数据时，首先需要读取数据，然后生成一个副本，对副本进行修改。修改完成后，再将老数据update成新的数据。使用RCU时，读者几乎不需要同步开销，既不需要获得锁，也不使用原子指令，不会导致锁竞争，因此就不用考虑死锁问题了。而对于写者的同步开销较大，它需要复制被修改的数据，还必须使用锁机制同步并行其它写者的修改操作。在有大量读操作，少量写操作的情况下效率非常高。

请回答一下进程和线程的区别

**参考答案**

参考回答:

1、一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程依赖于进程而存在。

2、进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。（资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。）

3、进程是资源分配的最小单位，线程是CPU调度的最小单位。

4、系统开销： 由于在创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I／o设备等。因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。类似地，在进行进程切换时，涉及到整个当前进程CPU环境的保存以及新被调度运行的进程的CPU环境的设置。而线程切换只须保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，进程切换的开销也远大于线程切换的开销。

5、通信：由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现，也变得比较容易。进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。在有的系统中，线程的切换、同步和通信都无须操作系统内核的干预 。

6、进程编程调试简单可靠性高，但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。

7、进程间不会相互影响 ；线程一个线程挂掉将导致整个进程挂掉。

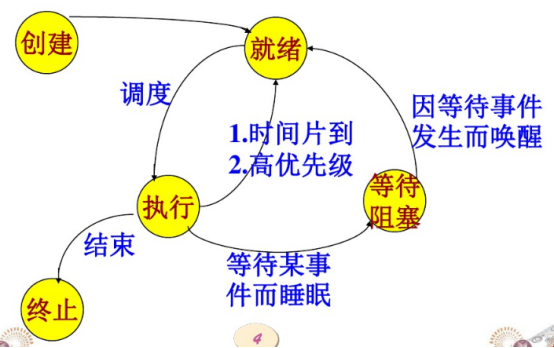
8、进程适应于多核、多机分布；线程适用于多核。

请你说一说进程状态转换图，动态就绪，静态就绪，动态阻塞，静态阻塞

**参考答案**

参考回答:

1、进程的五种基本状态：



1）创建状态：进程正在被创建

2）就绪状态：进程被加入到就绪队列中等待CPU调度运行

3）执行状态：进程正在被运行

4）等待阻塞状态：进程因为某种原因，比如等待I/O，等待设备，而暂时不能运行。

5）终止状态：进程运行完毕

2、交换技术

当多个进程竞争内存资源时，会造成内存资源紧张，并且，如果此时没有就绪进程，处理机会空闲，I/0速度比处理机速度慢得多，可能出现全部进程阻塞等待I/O。

针对以上问题，提出了两种解决方法：

1）交换技术：换出一部分进程到外存，腾出内存空间。

2）虚拟存储技术：每个进程只能装入一部分程序和数据。

在交换技术上，将内存暂时不能运行的进程，或者暂时不用的数据和程序，换出到外存，来腾出足够的内存空间，把已经具备运行条件的进程，或进程所需的数据和程序换入到内存。

从而出现了进程的挂起状态：进程被交换到外存，进程状态就成为了挂起状态。

3、活动阻塞，静止阻塞，活动就绪，静止就绪

1）活动阻塞：进程在内存，但是由于某种原因被阻塞了。

2）静止阻塞：进程在外存，同时被某种原因阻塞了。

3）活动就绪：进程在内存，处于就绪状态，只要给CPU和调度就可以直接运行。

4）静止就绪：进程在外存，处于就绪状态，只要调度到内存，给CPU和调度就可以运行。

从而出现了：

活动就绪 ——  静止就绪        （内存不够，调到外存）

活动阻塞 ——  静止阻塞        （内存不够，调到外存）

执行     ——  静止就绪         （时间片用完）

请你回答一下软链接和硬链接区别

**参考答案**

参考回答:

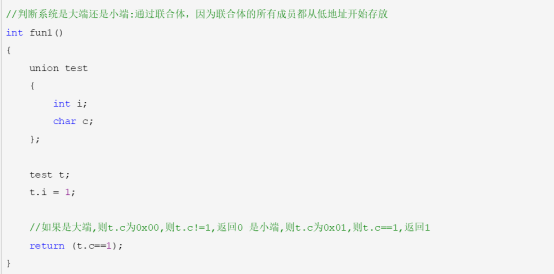
为了解决文件共享问题，Linux引入了软链接和硬链接。除了为Linux解决文件共享使用，还带来了隐藏文件路径、增加权限安全及节省存储等好处。若1个inode号对应多个文件名，则为硬链接，即硬链接就是同一个文件使用了不同的别名,使用ln创建。若文件用户数据块中存放的内容是另一个文件的路径名指向，则该文件是软连接。软连接是一个普通文件，有自己独立的inode,但是其数据块内容比较特殊。

请问什么是大端小端以及如何判断大端小端

**参考答案**

参考回答:

大端是指低字节存储在高地址；小端存储是指低字节存储在低地址。我们可以根据联合体来判断该系统是大端还是小端。因为联合体变量总是从低地址存储。



请你回答一下静态变量什么时候初始化

# 参考答案

参考回答:

静态变量存储在虚拟地址空间的数据段和bss段，C语言中其在代码执行之前初始化，属于编译期初始化。而C++中由于引入对象，对象生成必须调用构造函数，因此C++规定全局或局部静态对象当且仅当对象首次用到时进行构造

请你说一说用户态和内核态区别

# 参考答案

参考回答:

用户态和内核态是操作系统的两种运行级别，两者最大的区别就是特权级不同。用户态拥有最低的特权级，内核态拥有较高的特权级。运行在用户态的程序不能直接访问操作系统内核数据结构和程序。内核态和用户态之间的转换方式主要包括：系统调用，异常和中断。

92>

请你说一下TCP怎么保证可靠性，并且简述一下TCP建立连接和断开连接的过程

# 参考答案

参考回答:

TCP保证可靠性：

（1）序列号、确认应答、超时重传

数据到达接收方，接收方需要发出一个确认应答，表示已经收到该数据段，并且确认序号会说明了它下一次需要接收的数据序列号。如果发送发迟迟未收到确认应答，那么可能是发送的数据丢失，也可能是确认应答丢失，这时发送方在等待一定时间后会进行重传。这个时间一般是2\*RTT(报文段往返时间）+一个偏差值。

（2）窗口控制与高速重发控制/快速重传（重复确认应答）

TCP会利用窗口控制来提高传输速度，意思是在一个窗口大小内，不用一定要等到应答才能发送下一段数据，窗口大小就是无需等待确认而可以继续发送数据的最大值。如果不使用窗口控制，每一个没收到确认应答的数据都要重发。

使用窗口控制，如果数据段1001-2000丢失，后面数据每次传输，确认应答都会不停地发送序号为1001的应答，表示我要接收1001开始的数据，发送端如果收到3次相同应答，就会立刻进行重发；但还有种情况有可能是数据都收到了，但是有的应答丢失了，这种情况不会进行重发，因为发送端知道，如果是数据段丢失，接收端不会放过它的，会疯狂向它提醒......

（3）拥塞控制

如果把窗口定的很大，发送端连续发送大量的数据，可能会造成网络的拥堵（大家都在用网，你在这狂发，吞吐量就那么大，当然会堵），甚至造成网络的瘫痪。所以TCP在为了防止这种情况而进行了拥塞控制。

慢启动：定义拥塞窗口，一开始将该窗口大小设为1，之后每次收到确认应答（经过一个rtt），将拥塞窗口大小\*2。

拥塞避免：设置慢启动阈值，一般开始都设为65536。拥塞避免是指当拥塞窗口大小达到这个阈值，拥塞窗口的值不再指数上升，而是加法增加（每次确认应答/每个rtt，拥塞窗口大小+1），以此来避免拥塞。

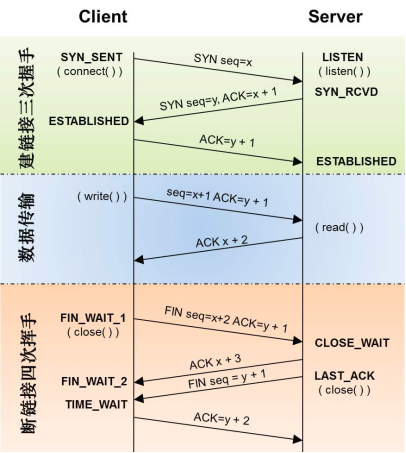
将报文段的超时重传看做拥塞，则一旦发生超时重传，我们需要先将阈值设为当前窗口大小的一半，并且将窗口大小设为初值1，然后重新进入慢启动过程。

快速重传：在遇到3次重复确认应答（高速重发控制）时，代表收到了3个报文段，但是这之前的1个段丢失了，便对它进行立即重传。

然后，先将阈值设为当前窗口大小的一半，然后将拥塞窗口大小设为慢启动阈值+3的大小。

这样可以达到：在TCP通信时，网络吞吐量呈现逐渐的上升，并且随着拥堵来降低吞吐量，再进入慢慢上升的过程，网络不会轻易的发生瘫痪。

TCP建立连接和断开连接的过程：



三次握手：

1. Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

2. Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

3. Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

四次挥手：

由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭。

1.数据传输结束后，客户端的应用进程发出连接释放报文段，并停止发送数据，客户端进入FIN\_WAIT\_1状态，此时客户端依然可以接收服务器发送来的数据。

2.服务器接收到FIN后，发送一个ACK给客户端，确认序号为收到的序号+1，服务器进入CLOSE\_WAIT状态。客户端收到后进入FIN\_WAIT\_2状态。

3.当服务器没有数据要发送时，服务器发送一个FIN报文，此时服务器进入LAST\_ACK状态，等待客户端的确认

4.客户端收到服务器的FIN报文后，给服务器发送一个ACK报文，确认序列号为收到的序号+1。此时客户端进入TIME\_WAIT状态，等待2MSL（MSL：报文段最大生存时间），然后关闭连接。

请回答一下HTTP和HTTPS的区别，以及HTTPS有什么缺点？

# 参考答案

参考回答:

HTTP协议和HTTPS协议区别如下：

1）HTTP协议是以明文的方式在网络中传输数据，而HTTPS协议传输的数据则是经过TLS加密后的，HTTPS具有更高的安全性

2）HTTPS在TCP三次握手阶段之后，还需要进行SSL 的handshake，协商加密使用的对称加密密钥

3）HTTPS协议需要服务端申请证书，浏览器端安装对应的根证书

4）HTTP协议端口是80，HTTPS协议端口是443

HTTPS优点：

HTTPS传输数据过程中使用密钥进行加密，所以安全性更高

HTTPS协议可以认证用户和服务器，确保数据发送到正确的用户和服务器

HTTPS缺点：

HTTPS握手阶段延时较高：由于在进行HTTP会话之前还需要进行SSL握手，因此HTTPS协议握手阶段延时增加

HTTPS部署成本高：一方面HTTPS协议需要使用证书来验证自身的安全性，所以需要购买CA证书；另一方面由于采用HTTPS协议需要进行加解密的计算，占用CPU资源较多，需要的服务器配置或数目高

请你说一说HTTP返回码

**参考答案**

参考回答:

HTTP协议的响应报文由状态行、响应头部和响应包体组成，其响应状态码总体描述如下：

1xx：指示信息--表示请求已接收，继续处理。

2xx：成功--表示请求已被成功接收、理解、接受。

3xx：重定向--要完成请求必须进行更进一步的操作。

4xx：客户端错误--请求有语法错误或请求无法实现。

5xx：服务器端错误--服务器未能实现合法的请求。

常见状态代码、状态描述的详细说明如下。

200 OK：客户端请求成功。

206 partial content服务器已经正确处理部分GET请求，实现断点续传或同时分片下载，该请求必须包含Range请求头来指示客户端期望得到的范围

300 multiple choices（可选重定向）:被请求的资源有一系列可供选择的反馈信息，由浏览器/用户自行选择其中一个。

301  moved permanently（永久重定向）：该资源已被永久移动到新位置，将来任何对该资源的访问都要使用本响应返回的若干个URI之一。

302 move temporarily(临时重定向)：请求的资源现在临时从不同的URI中获得，

304：not modified :如果客户端发送一个待条件的GET请求并且该请求以经被允许，而文档内容未被改变，则返回304,该响应不包含包体（即可直接使用缓存）。

403 Forbidden：服务器收到请求，但是拒绝提供服务。

t Found：请求资源不存在，举个例子：输入了错误的URL。

请你说一说IP地址作用，以及MAC地址作用

**参考答案**

参考回答:

MAC地址是一个硬件地址，用来定义网络设备的位置，主要由数据链路层负责。而IP地址是IP协议提供的一种统一的地址格式，为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。

请介绍一下操作系统中的中断

**参考答案**

参考回答:

中断是指CPU对系统发生的某个事件做出的一种反应，CPU暂停正在执行的程序，保存现场后自动去执行相应的处理程序，处理完该事件后再返回中断处继续执行原来的程序。中断一般三类，一种是由CPU外部引起的，如I/O中断、时钟中断，一种是来自CPU内部事件或程序执行中引起的中断，例如程序非法操作，地址越界、浮点溢出），最后一种是在程序中使用了系统调用引起的。而中断处理一般分为中断响应和中断处理两个步骤，中断响应由硬件实施，中断处理主要由软件实施。

请回答OSI七层模型和TCP/IP四层模型，每层列举2个协议

# 参考答案

参考回答:

OSI七层模型及其包含的协议如下:

物理层: 通过媒介传输比特,确定机械及电气规范,传输单位为bit，主要包括的协议为：IEE802.3 CLOCK RJ45

数据链路层: 将比特组装成帧和点到点的传递,传输单位为帧,主要包括的协议为MAC VLAN PPP

网络层：负责数据包从源到宿的传递和网际互连，传输单位为包,主要包括的协议为IP ARP ICMP

传输层：提供端到端的可靠报文传递和错误恢复，传输单位为报文,主要包括的协议为TCP UDP

会话层：建立、管理和终止会话，传输单位为SPDU，主要包括的协议为RPC NFS

表示层: 对数据进行翻译、加密和压缩,传输单位为PPDU，主要包括的协议为JPEG ASII

应用层: 允许访问OSI环境的手段,传输单位为APDU，主要包括的协议为FTP HTTP DNS

TCP/IP 4层模型包括：

网络接口层：MAC VLAN

网络层:IP ARP ICMP

传输层:TCP UDP

应用层:HTTP DNS SMTP

请你说一说TCP的三次握手和四次挥手的过程及原因

**参考答案**

参考回答:

TCP的三次握手过程如下：

C-> SYN -> S

S->SYN/ACK->C

C->ACK->S

三次握手的原因：三次握手可以防止已经失效的连接请求报文突然又传输到服务器端导致的服务器资源浪费。例如，客户端先发送了一个SYN，但是由于网络阻塞，该SYN数据包在某个节点长期滞留。然后客户端又重传SYN数据包并正确建立TCP连接，然后传输完数据后关闭该连接。该连接释放后失效的SYN数据包才到达服务器端。在二次握手的前提下，服务器端会认为这是客户端发起的又一次请求，然后发送SYN ，并且在服务器端创建socket套接字，一直等待客户端发送数据。但是由于客户端并没有发起新的请求，所以会丢弃服务端的SYN 。此时服务器会一直等待客户端发送数据从而造成资源浪费。

TCP的四次挥手过程如下：

C->FIN->S

S->ACK->C

S->FIN->C

C->ACK->S

四次挥手的原因：由于连接的关闭控制权在应用层，所以被动关闭的一方在接收到FIN包时，TCP协议栈会直接发送一个ACK确认包，优先关闭一端的通信。然后通知应用层，由应用层决定什么时候发送FIN包。应用层可以使用系统调用函数read==0来判断对端是否关闭连接。

搜索baidu，会用到计算机网络中的什么层？每层是干什么的

# 参考答案

参考回答:

浏览器中输入URL

浏览器要将URL解析为IP地址，解析域名就要用到DNS协议，首先主机会查询DNS的缓存，如果没有就给本地DNS发送查询请求。DNS查询分为两种方式，一种是递归查询，一种是迭代查询。如果是迭代查询，本地的DNS服务器，向根域名服务器发送查询请求，根域名服务器告知该域名的一级域名服务器，然后本地服务器给该一级域名服务器发送查询请求，然后依次类推直到查询到该域名的IP地址。DNS服务器是基于UDP的，因此会用到UDP协议。

得到IP地址后，浏览器就要与服务器建立一个http连接。因此要用到http协议，http协议报文格式上面已经提到。http生成一个get请求报文，将该报文传给TCP层处理，所以还会用到TCP协议。如果采用https还会使用https协议先对http数据进行加密。TCP层如果有需要先将HTTP数据包分片，分片依据路径MTU和MSS。TCP的数据包然后会发送给IP层，用到IP协议。IP层通过路由选路，一跳一跳发送到目的地址。当然在一个网段内的寻址是通过以太网协议实现(也可以是其他物理层协议，比如PPP，SLIP)，以太网协议需要直到目的IP地址的物理地址，有需要ARP协议。

其中：

1、DNS协议，http协议，https协议属于应用层

应用层是体系结构中的最高层。应用层确定进程之间通信的性质以满足用户的需要。这里的进程就是指正在运行的程序。应用层不仅要提供应用进程所需要的信息交换和远地操作，而且还要作为互相作用的应用进程的用户代理，来完成一些为进行语义上有意义的信息交换所必须的功能。应用层直接为用户的应用进程提供服务。

2、TCP/UDP属于传输层

传输层的任务就是负责主机中两个进程之间的通信。因特网的传输层可使用两种不同协议：即面向连接的传输控制协议[TCP](https://www.baidu.com/s?wd=TCP&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，和无连接的用户数据报协议[UDP](https://www.baidu.com/s?wd=UDP&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)。面向连接的服务能够提供可靠的交付，但无连接服务则不保证提供可靠的交付，它只是“尽最大努力交付”。这两种服务方式都很有用，备有其优缺点。在分组交换网内的各个交换结点机都没有传输层。

3、IP协议，ARP协议属于网络层

网络层负责为分组交换网上的不同主机提供通信。在发送数据时，网络层将运输层产生的报文段或用户数据报封装成分组或包进行传送。在[TCP](https://www.baidu.com/s?wd=TCP&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)/IP体系中，分组也叫作IP数据报，或简称为数据报。网络层的另一个任务就是要选择合适的路由，使源主机运输层所传下来的分组能够交付到目的主机。  
4、数据链路层

当发送数据时，数据链路层的任务是将在网络层交下来的IP数据报组装成帧，在两个相邻结点间的链路上传送以帧为单位的数据。每一帧包括数据和必要的控制信息（如同步信息、地址信息、差错控制、以及[流量控制](https://www.baidu.com/s?wd=%E6%B5%81%E9%87%8F%E6%8E%A7%E5%88%B6&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)信息等）。控制信息使接收端能够知道—个帧从哪个比特开始和到哪个比特结束。控制信息还使接收端能够检测到所收到的帧中有无差错。  
5、物理层

物理层的任务就是透明地传送比特流。在物理层上所传数据的单位是比特。传递信息所利用的一些物理媒体，如[双绞线](https://www.baidu.com/s?wd=%E5%8F%8C%E7%BB%9E%E7%BA%BF&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)、同轴电缆、光缆等，并不在物理层之内而是在物理层的下面。因此也有人把物理媒体当做第0层。

请你说一说TCP拥塞控制？以及达到什么情况的时候开始减慢增长的速度？

# 参考答案

参考回答:

拥塞控制是防止过多的数据注入网络，使得网络中的路由器或者链路过载。流量控制是点对点的通信量控制，而拥塞控制是全局的网络流量整体性的控制。发送双方都有一个拥塞窗口——cwnd。

1、慢开始

最开始发送方的拥塞窗口为1，由小到大逐渐增大发送窗口和拥塞窗口。每经过一个传输轮次，拥塞窗口cwnd加倍。当cwnd超过慢开始门限，则使用拥塞避免算法，避免cwnd增长过大。

2、拥塞避免

每经过一个往返时间RTT，cwnd就增长1。

在慢开始和拥塞避免的过程中，一旦发现网络拥塞，就把慢开始门限设为当前值的一半，并且重新设置cwnd为1，重新慢启动。（乘法减小，加法增大）

3、快重传

接收方每次收到一个失序的报文段后就立即发出重复确认，发送方只要连续收到三个重复确认就立即重传（尽早重传未被确认的报文段）。

4、快恢复

当发送方连续收到了三个重复确认，就乘法减半（慢开始门限减半），将当前的cwnd设置为慢开始门限，并且采用拥塞避免算法（连续收到了三个重复请求，说明当前网络可能没有拥塞）。

采用快恢复算法时，慢开始只在建立连接和网络超时才使用。

达到什么情况的时候开始减慢增长的速度？

采用慢开始和拥塞避免算法的时候

1. 一旦cwnd>慢开始门限，就采用拥塞避免算法，减慢增长速度

2. 一旦出现丢包的情况，就重新进行慢开始，减慢增长速度

采用快恢复和快重传算法的时候

1. 一旦cwnd>慢开始门限，就采用拥塞避免算法，减慢增长速度

2. 一旦发送方连续收到了三个重复确认，就采用拥塞避免算法，减慢增长速度

请你说一下数据库事务以及四个特性

# 参考答案

参考回答:

事务（Transaction）是由一系列对系统中数据进行访问与更新的操作所组成的一个程序执行逻辑单元。事务是DBMS中最基础的单位，事务不可分割。

事务具有4个基本特征，分别是：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Duration），简称ACID。

1. 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，[删删删]因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

2. 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

3. 隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。

这指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。

不同的隔离级别：

Read Uncommitted（读取未提交[添加中文释义]内容）：最低的隔离级别，什么都不需要做，一个事务可以读到另一个事务未提交的结果。所有的并发事务问题都会发生。

Read Committed（读取提交内容）：只有在事务提交后，其更新结果才会被其他事务看见。可以解决脏读问题。

Repeated Read（可重复读）：在一个事务中，对于同一份数据的读取结果总是相同的，无论是否有其他事务对这份数据进行操作，以及这个事务是否提交。可以解决脏读、不可重复读。

Serialization（可串行化）：事务串行化执行，隔离级别最高，牺牲了系统的并发性。可以解决并发事务的所有问题。

4. 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

请你说一说数据库的三大范式

# 参考答案

参考回答:

第一范式：当关系模式R的所有属性都不能再分解为更基本的数据单位时，称R是满足第一范式，即属性不可分

第二范式：如果关系模式R满足第一范式，并且R得所有非主属性都完全依赖于R的每一个候选关键属性，称R满足第二范式

第三范式：设R是一个满足第一范式条件的关系模式，X是R的任意属性集，如果X非传递依赖于R的任意一个候选关键字，称R满足第三范式，即非主属性不传递依赖于键码

请你介绍一下数据库的ACID特性

# 参考答案

参考回答:

1)原子性：事务被视为不可分割的最小单元，事物的所有操作要不成功，要不失败回滚，而回滚可以通过日志来实现，日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作。

2)一致性：数据库在事务执行前后都保持一致性状态，在一致性状态下，所有事务对一个数据的读取结果都是相同的。

3)隔离性：一个事务所做的修改在最终提交以前，对其他事务是可不见的。

4)持久性：一旦事务提交，则其所做的修改将会永远保存到数据库中。

请你说一说mysql的四种隔离状态

**参考答案**

参考回答:

Mysql主要包含四种隔离状态：

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 是 | 是 | 是 |
| 不可重复读（read-committed） | 否 | 是 | 是 |
| 可重复读（repeatable-read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

请问SQL优化方法有哪些

# 参考答案

参考回答:

通过建立索引对查询进行优化

对查询进行优化，应尽量避免全表扫描

请你回答一下mongodb和redis的区别

# 参考答案

参考回答:

内存管理机制上：Redis 数据全部存在内存，定期写入磁盘，当内存不够时，可以选择指定的 LRU 算法删除数据。MongoDB 数据存在内存，由 linux系统 mmap 实现，当内存不够时，只将热点数据放入内存，其他数据存在磁盘。

支持的数据结构上：Redis 支持的数据结构丰富，包括hash、set、list等。

MongoDB 数据结构比较单一，但是支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富

请问什么是单向链表，如何判断两个单向链表是否相交

# 参考答案

参考回答:

考察点：数据结构，算法

公司：百度

1、单向链表

单向链表（单链表）是链表的一种，其特点是链表的链接方向是单向的，对链表的访问要通过顺序读取从头部开始；链表是使用指针进行构造的列表；又称为结点列表，因为链表是由一个个结点组装起来的；其中每个结点都有指针成员变量指向列表中的下一个结点。

列表是由结点构成，head指针指向第一个成为表头结点，而终止于最后一个指向nuLL的指针。



2、判断两个链表是否相交

1）方法1：

链表相交之后，后面的部分节点全部共用，可以用2个指针分别从这两个链表头部走到尾部，最后判断尾部指针的地址信息是否一样，若一样则代表链表相交！

2）方法2：

可以把其中一个链表的所有节点地址信息存到数组中，然后把另一个链表的每一个节点地址信息遍历数组，若相等，则跳出循环，说明链表相交。进一步优化则是进行hash排序，建立hash表。

请你手写链表反转

# 参考答案

参考回答:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | struct ListNode {  int val;  struct ListNode \*next;  ListNode(int x) :val(x), next(NULL) {}  }  ListNode\* ReverseList(ListNode\* pHead)  {  if(!pHead||!pHead->next)return pHead;  ListNode \*pre=nullptr;  ListNode \*p=pHead;  ListNode \*next=pHead->next;  while(p)  {  p->next=pre;  pre=p;  p=next;  if(next)  next=next->next;  }  return pre;  } |

请你说一说哈希冲突的解决方法

**参考答案**

参考回答:

考察点：hash冲突，数据结构

公司：腾讯

1、开放定址

开放地址法有个非常关键的特征，就是所有输入的元素全部存放在哈希表里，也就是说，位桶的实现是不需要任何的链表来实现的，换句话说，也就是这个哈希表的装载因子不会超过1。它的实现是在插入一个元素的时候，先通过哈希函数进行判断，若是发生哈希冲突，就以当前地址为基准，根据再寻址的方法（探查序列），去寻找下一个地址，若发生冲突再去寻找，直至找到一个为空的地址为止。所以这种方法又称为再散列法。

有几种常用的探查序列的方法：

①线性探查

dii=1，2，3，…，m-1；这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

②二次探查

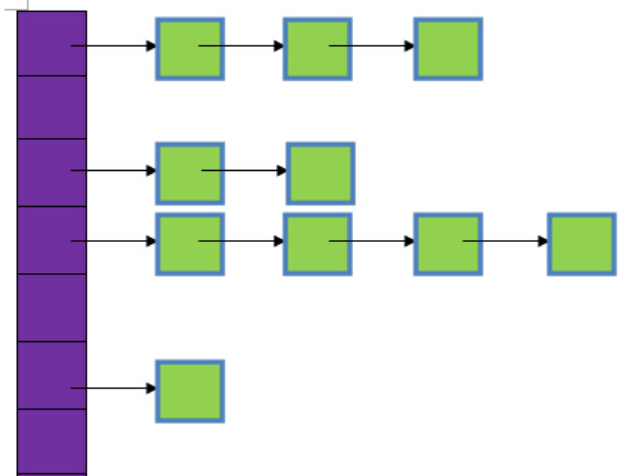
di=12，-12，22，-22，…，k2，-k2    ( k<=m/2 )；这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

③ 伪随机探测

di=伪随机数序列；具体实现时，应建立一个伪随机数发生器，（如i=(i+p) % m），生成一个位随机序列，并给定一个随机数做起点，每次去加上这个伪随机数++就可以了。

2、链地址

每个位桶实现的时候，采用链表或者树的数据结构来去存取发生哈希冲突的输入域的关键字，也就是被哈希函数映射到同一个位桶上的关键字。



紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中，即链接在桶后。

3、公共溢出区

建立一个公共溢出区域，把hash冲突的元素都放在该溢出区里。查找时，如果发现hash表中对应桶里存在其他元素，还需要在公共溢出区里再次进行查找。

4、再hash

再散列法其实很简单，就是再使用哈希函数去散列一个输入的时候，输出是同一个位置就再次散列，直至不发生冲突位置。

缺点：每次冲突都要重新散列，计算时间增加。

请你介绍一下快排算法；以及什么是稳定性排序，快排是稳定性的吗；快排算法最差情况推导公式

# 参考答案

参考回答:

1、快排算法

根据哨兵元素，用两个指针指向待排序数组的首尾，首指针从前往后移动找到比哨兵元素大的，尾指针从后往前移动找到比哨兵元素小的，交换两个元素，直到两个指针相遇，这是一趟排序，经常这趟排序后，比哨兵元素大的在右边，小的在左边。经过多趟排序后，整个数组有序。

稳定性：不稳定

平均时间复杂度：O(nlogn)

2、稳定排序

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。

快排算法是不稳定的排序算法。例如：

待排序数组:int a[] ={1, 2, 2, 3, 4, 5, 6};

若选择a[2]（即数组中的第二个2）为枢轴，而把大于等于比较子的数均放置在大数数组中，则a[1]（即数组中的第一个2）会到pivot的右边， 那么数组中的两个2非原序。

若选择a[1]为比较子，而把小于等于比较子的数均放置在小数数组中，则数组中的两个2顺序也非原序。

3、快排最差情况推倒

在快速排序的早期版本中呢，最左面或者是最右面的那个元素被选为枢轴，那最坏的情况就会在下面的情况下发生啦：

1）数组已经是正序排过序的。 （每次最右边的那个元素被选为枢轴）

2）数组已经是倒序排过序的。 （每次最左边的那个元素被选为枢轴）

3）所有的元素都相同（1、2的特殊情况）

因为这些案例在用例中十分常见，所以这个问题可以通过要么选择一个随机的枢轴，或者选择一个分区中间的下标作为枢轴，或者（特别是对于相比更长的分区）选择分区的第一个、中间、最后一个元素的中值作为枢轴。有了这些修改，那快排的最差的情况就不那么容易出现了，但是如果输入的数组最大（或者最小元素）被选为枢轴，那最坏的情况就又来了。

快速排序，在最坏情况退化为冒泡排序，需要比较O(n2)次（n(n - 1)/2次）。

请问稳定排序哪几种？

# 参考答案

参考回答:

基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序

请问快排的时间复杂度最差是多少？什么时候时间最差

# 参考答案

参考回答:

O（N2),元素本来倒序排列用时最多

请你说一说小根堆特点

# 参考答案

参考回答:

堆是一棵完全二叉树（如果一共有h层，那么1~h-1层均满，在h层可能会连续缺失若干个右叶子）。

1）小根堆

若根节点存在左子女则根节点的值小于左子女的值；若根节点存在右子女则根节点的值小于右子女的值。

2）大根堆

若根节点存在左子女则根节点的值大于左子女的值；若根节点存在右子女则根节点的值大于右子女的值。

手写代码：二叉树序列化反序列化

# 参考答案

参考回答:

> 序列化：必须保存一个中序遍历结果，然后外加一个前序或者后序遍历结果

>反序列化：根据两次遍历生成的结果恢复二叉树，代码如下(前序和中序)：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | ```TreeNode\* helper(vector<int>pre,int startPre,int endPre,vector<int>in,int startIn,int endIn)  {  if(startPre>endPre||startIn>endIn)  return nullptr;  TreeNode \* root=new TreeNode(pre[startPre]);  for(int i=startIn;i<=endIn;++i)  {  if(in[i]==pre[startPre])  {  root->left=helper(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);  root->right=helper(pre,i-startIn+startPre+1,endPre,in,i+1,endIn);  break;  }  }  return root;  }  TreeNode\* reConstructBinaryTree(vector<int> pre,vector<int> vin)  {  TreeNode \*root=helper(pre,0,pre.size()-1,vin,0,vin.size()-1);  return root;  } |

请你实现二叉树的层序遍历并输出

# 参考答案

参考回答:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | void layerTrace(BTreeNode \*T)  {  if(T== nullptr)return;  BTreeNode \*p=T;  queue<BTreeNode\*>q;  q.push(p);  while(!q.empty())  {  p=q.front();  q.pop();  cout<<<<p->data;  if(p->left!= nullptr)q.push(p->left);  if(p->right!= nullptr)q.push(p->right);  }  }`` |

请你说一说红黑树的性质还有左右旋转

# 参考答案

参考回答:

考察点：算法

公司：京东，阿里巴巴

1）平衡二叉树（AVL树）：

红黑树是在AVL树的基础上提出来的。

平衡二叉树又称为AVL树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1。

AVL树中所有结点为根的树的左右子树高度之差的绝对值不超过1。

将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

2）红黑树：

红黑树是在AVL树的基础上发展而来的。红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑

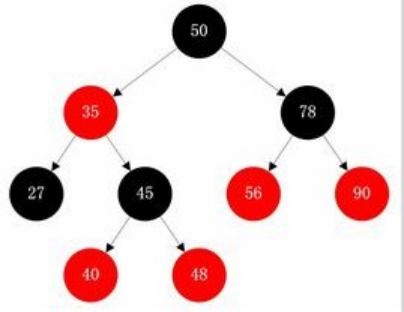
2. 根节点是黑的;

3. 每个叶节点（叶节点即树尾端NULL指针或NULL节点）都是黑的;

4. 如果一个节点是红色的，则它的子节点必须是黑色的。

5. 对于任意节点而言，其到叶子点树NULL指针的每条路径都包含相同数目的黑节点;

从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。恢复红黑属性需要少量(O(log n))的颜色变更(这在实践中是非常快速的)并且不超过三次树旋转(对于插入是两次)。这允许插入和删除保持为 O(log n) 次，



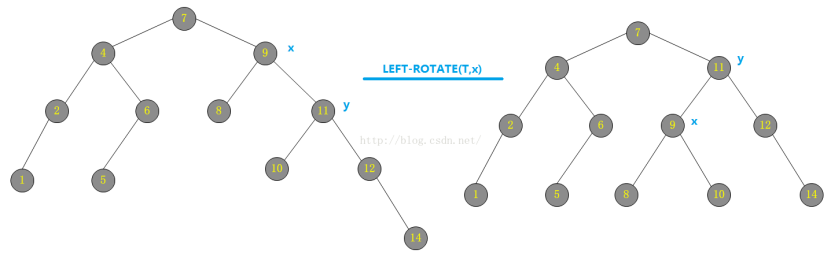
3）红黑树较AVL树的优点：

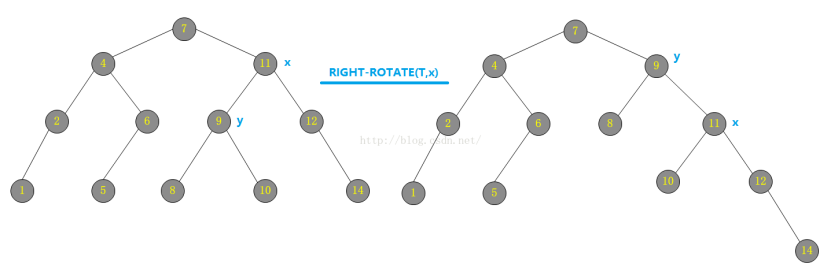
AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

所以红黑树在查找，插入删除的性能都是O(logn)，且性能稳定，所以STL里面很多结构包括map底层实现都是使用的红黑树。

4）红黑树旋转：

旋转：红黑树的旋转是一种能保持二叉搜索树性质的搜索树局部操作。有左旋和右旋两种旋转，通过改变树中某些结点的颜色以及指针结构来保持对红黑树进行插入和删除操作后的红黑性质。

左旋：对某个结点x做左旋操作时，假设其右孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的左孩子，y的左孩子成为x的右孩子。

右旋：对某个结点x做右旋操作时，假设其左孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的右孩子，y的右孩子成为x的左孩子。

请你说一说C++两种map

# 参考答案

参考回答:

unordered\_map（哈希表)和map（红黑树）

请你说一说inner join和left join

# 参考答案

参考回答:

left join(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录 right join(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录  
inner join(等值连接) 只返回两个表中联结字段相等的行

请你说一说数据库事务隔离

# 参考答案

参考回答:

同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。

请你讲述一下Socket编程的send() recv() accept() socket()函数？

# 参考答案

参考回答:

send函数用来向TCP连接的另一端发送数据。客户程序一般用send函数向服务器发送请求，而服务器则通常用send函数来向客户程序发送应答,send的作用是将要发送的数据拷贝到缓冲区，协议负责传输。

recv函数用来从TCP连接的另一端接收数据，当应用程序调用recv函数时，recv先等待s的发送缓冲中的数据被协议传送完毕，然后从缓冲区中读取接收到的内容给应用层。  
accept函数用了接收一个连接，内核维护了半连接队列和一个已完成连接队列，当队列为空的时候，accept函数阻塞，不为空的时候accept函数从上边取下来一个已完成连接，返回一个文件描述符。