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：

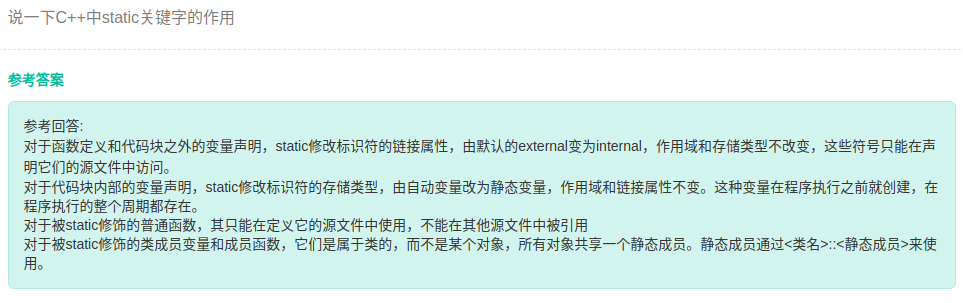
C++是面向对象的语言，而C是面向过程的结构化编程语言

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；

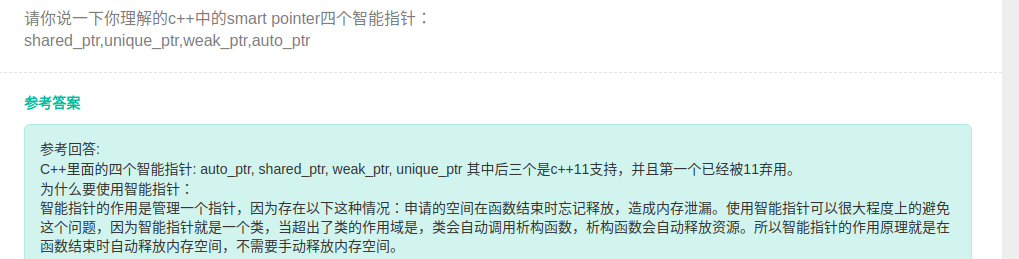
5.可以有const指针，但是没有const引用；

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

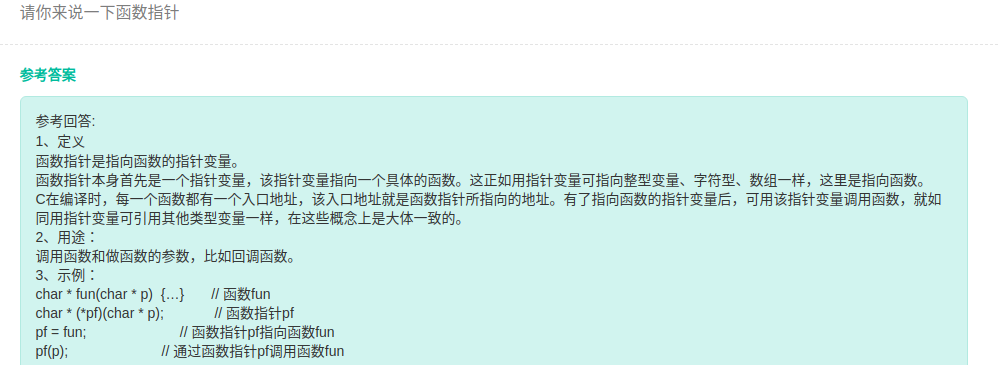
考点:虚函数 析构函数

# 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);

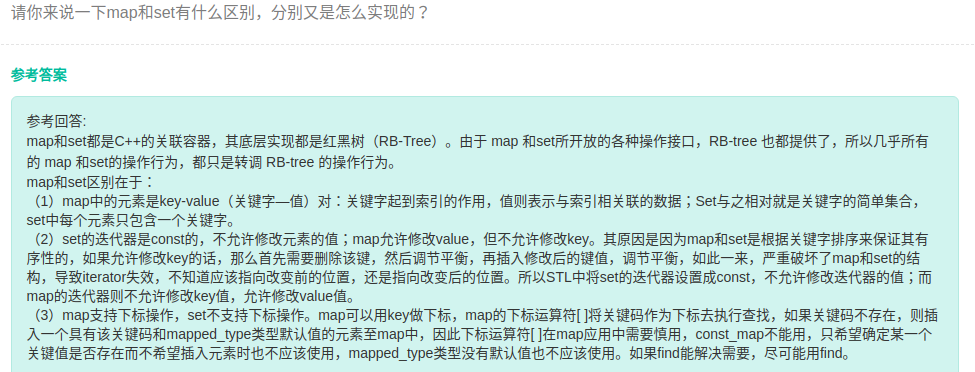
成功调用fork( )会创建一个新的进程，它几乎与调用fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork( )调用会返回0。在父进程中fork( )返回子进程的pid。如果出现错误，fork( )返回一个负值。 最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。 在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程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：

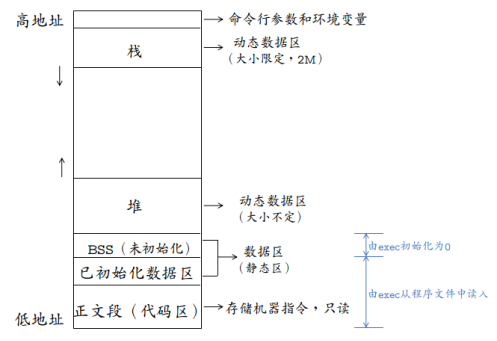
1.虚存的管理需要建立很多数据结构，这些数据结构要占用额外的内存

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

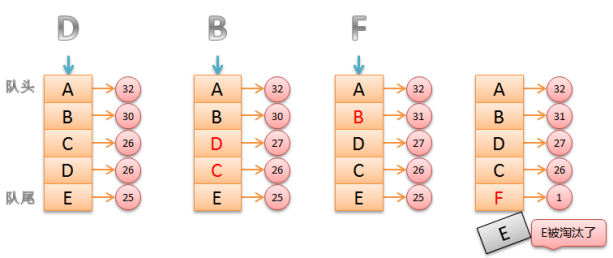
2、LFU（最不经常访问淘汰算法）

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。

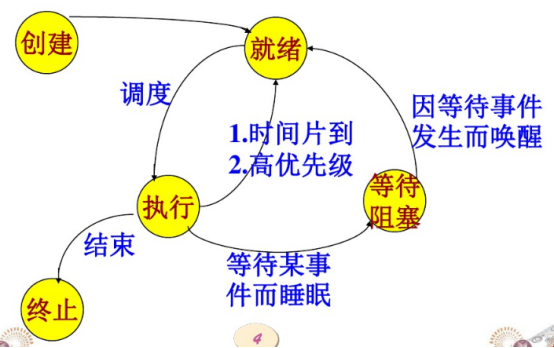
8、进程适应于多核、多机分布；线程适用于多核。

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:

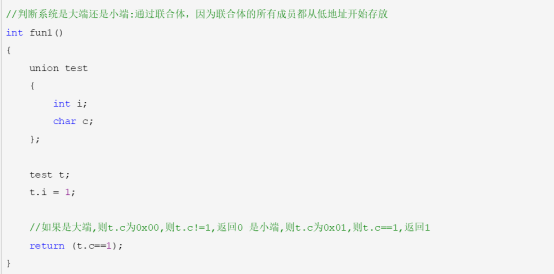
为了解决文件共享问题，Linux引入了软链接和硬链接。除了为Linux解决文件共享使用，还带来了隐藏文件路径、增加权限安全及节省存储等好处。若1个inode号对应多个文件名，则为硬链接，即硬链接就是同一个文件使用了不同的别名,使用ln创建。若文件用户数据块中存放的内容是另一个文件的路径名指向，则该文件是软连接。软连接是一个普通文件，有自己独立的inode,但是其数据块内容比较特殊。

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>