**static关键字的作用**

1. 全局静态变量

在全局变量前加上关键字static

内存中位置：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，除非他被显示初始化）

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

1. 局部静态变量

在局部变量之前加上关键字static

内存中位置：静态存储区

初始化：未经初始化的局部静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

作用域：作用域仍为局部作用域，当定义它的函数或语句块结束的时候，作用域结束。但是当局部静态变量离开作用域之后，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

1. 静态函数

在函数返回值类型前加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，但静态函数只是声明在它的文件当中可见，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：不要在头文件中声明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去，否则cpp内部声明需加上static修饰

1. 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环隐藏原则，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。对于多个对象来说，静态数据成员只存储一处，供所有对象使用

1. 类的静态函数，

静态成员函数和静态数据成员一样，它们都属于类发静态成员，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。从中可以看出，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>)

**C++中static关键字的作用**

对于函数定义和代码块之外的变量声明，static修改标识符的链接属性，由默认的external变为internal，作用域和链接属性不变，这种变量在程序执行之前就创建，在程序执行的整个周期都存在

对于被static修饰的普通函数，其只能在定义它的源文件中使用，不能在其他源文件中被引用

对于被static修饰的类成员变量和成员函数，它们是属于类的，而不是某个对象，所有对象共享一个静态成员，静态成员通过<类名>::<静态成员>来使用

**C++和C的区别**

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、多态三种特性；C++相比C增加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

**C++中四种cast转换**

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能用于多态向上那转化，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原理。

向上转换：指的是子类像基类转换

向下转换：指的是基类像子类转换

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进行向下转换

1. 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，对转换结果不做任何保证
2. 为什么不使用C的强制转换：C的转化不够明确，不能进行错误检查，容易出错

**C/C++中指针和引用的区别**

1. 指针有自己的一块空间，而引用只是一个别名
2. 使用sizeof看一个指针大小时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，但没有const引用
6. 指针在使用中可以指向其他对象，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），而引用只有一级
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

给定三角形ABC和点P(x,y,z),判断点P是否在ABC内，并写出思路手写代码

（如果P在三角形ABC内，那ABP面积+BCP面积+ACP面积+ABC面积）

|  |
| --- |
| #include <iostream>  #include <math.h>  using namespace std;  #define ABS\_FLOAT\_0 0.0001  struct point\_float  {  float x;  float y;  };  float GetTriangleSquar(const point\_float pt0,const point\_float pt1,const point\_float pt2)  {  point\_float AB,BC;  AB.x=pt1.x-pt0.x;  AB.y=pt1.y-pt0.y;  BC.x=pt2.x-pt1.x;  BC.y=pt2.y-pt1.y;  return fabs((AB.x\*BC.y-AB.y\*BC.x))/2.0f;  }  bool IsInTriangle(const point\_float A,const point\_float B,const point\_float C,const point\_float D)  {  float SABC,SADB,SBDC,SADC;  SABC=GetTriangleSquar(A,B,C);  SADB=GetTriangleSquar(A,D,B);  SBDC=GetTriangleSquar(B,D,C);  SADC=GetTriangleSquar(A,D,C);  float SumSquer=SADB+SBDC+SADC;  if((-ABS\_FLOAT\_0<(SABC-SumSquer))&&((SABC-SumSquer)<ABS\_FLOAT\_0))  {  return true;  }  else  {  return false;  }  } |

**C++中smart pointer四个智能指针：shared\_ptr,unique\_ptr,weak\_ptr,auto\_ptr**

C++里面四个智能指针auto\_ptr,share+ptr,weak\_ptr,unique\_ptr其中后三个是c++11支持，并且第一个已经被11弃用

为什么要使用智能指针：

智能指针的作用是管理一个指针，因为存在以下情况：申请的空间在函数结束时忘记释放，造成内存泄露，使用智能指针可以很大程度上避免这个问题，因为智能指针就是一个类，当超出了类的作用域时，类就会自动调用析构函数，析构函数会自动释放资源，所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放。

1. auto\_ptr(c++98的方案，cpp11已经抛弃）

所有权模式

|  |
| --- |
| auto\_ptr<string>p1(new string(“i reigned lonely as acloud”));  auto\_ptr<string>p2;  P2=p1;//auto\_ptr不会报错 |

此时不会报错，篇剥夺了p1的所有权，但是当程序运行时访问p1将会报错，所以auto\_ptr的缺点是：存在潜在的内存崩溃问题

1. unique\_ptr(替换auto\_ptr)

unique\_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象，它对于避免资源泄露（例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用

采用所有权模式，还是上面那个例子

|  |
| --- |
| unique\_ptr<string>p3(new string(“auto”));  unique\_ptr<string>p4;  P4=p3;//此时会报错 |

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题，因此，unique\_ptr比auto\_ptr更安全

另外unique\_ptr还有更聪明的地方：当程序试图将一个unique\_ptr赋值给另一个时，如果源unique\_ptr是个临时右值，编译器允许这么做，如果源unique\_ptr将存在一段时间，编译器将禁止这么做，比如

|  |
| --- |
| unique\_ptr<string> pu1(new string(“hello world”));  unique\_ptr<string> pu2;  Pu2=pu1; //#1 not allowed  Unique\_ptr<string> pu3;  Pu3=unique\_ptr<string>(new string(“you”)); //#2 allowed |

其中#1留下悬挂的unique\_ptr(pu1),这可能导致危害，而#2不会留下悬挂的unique\_ptr,因为它调用unique\_ptr的构造函数，该构造函数创建的临时对象在其所有权让给pu3后就会销毁，这种随情况而已的行为表明，unique\_ptr优于允许两种赋值的auto\_ptr

注：如果确定想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move(),让你能够将一个unique\_ptr赋给另一个，例如

|  |
| --- |
| Unique\_ptr<string> ps1,ps2;  ps1=demo(“hello”);  ps2=move(psl);  ps1=demo(“alexia”);  cout<<\*ps2<<\*ps1<<endl; |

1. shared\_ptr

Share\_prt实现共享式拥有概念，多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放，从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享，可以通过成员函数use\_count()来查看资源的所有者个数，除了可以通过new来构造，还可以通过传入auto\_ptr,unique\_ptr,weak\_ptr来构造，当我们调用release()时，当前指针会释放资源所有权，技术减一，当计数等于0时，资源会被释放

share\_ptr是为了解决auto\_ptr在对象所有权上的局限性（auto\_ptr是独占的），在使用引用计数的机制上提供了可以共享所有权的智能指针

成员函数：

use\_count返回引用计数的个数

unique返回是否是独占所有权（use\_count为1）

swap搅浑两个share\_ptr对象（即交换所拥有的对象）

reset放弃内部对象的所有权或拥有对象的变更，会引起计数的减少

get返回内部对象（指针），由于已经重载了（）方法，因此和直接使用对象是一样的，如share\_ptr<int> sp(new int(1));sp与sp.get()是等价的

1. weak\_ptr

weak\_ptr是一种不控制对象生命周期的智能指针，它指向一个share\_ptr管理的对象，进行该对象的内存管理的是那个强引用的share\_ptr，weak\_ptr只是提供了一个对管理对象的一个访问手段，weak\_ptr设计的目的是为了配合share\_ptr而引入的一种智能指针来协助share\_ptr工作，它只可以从一个share\_ptr或另一个weak\_ptr对象构造，它的构造和析构不会引起引用计数的增加和减少，weak\_ptr是用来解决share\_ptr是用来解决share\_ptr相互引用时的死锁问题，如果说两个share\_ptr相互引用，那么这两个指针的引用计数永远不可能下降为0，资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用次数，和share\_ptr之间可以相互转化，share\_ptr可以直接赋值给它，它可以通过调用lock函数来获得share\_ptr

|  |
| --- |
| class B;  class A  {  public:  share\_ptr<B> pb\_;  ~A()  {cout<<”A delete\n”;}  };  class B  {  public:  share\_ptr<A> pa\_;  ~B;  {cout<<”B delete\n”;}  };  void fun()  {  share\_ptr<B> pb(new B());  share\_ptr<A> pa(new A());  pb->pa\_=pa;  pa->pb\_=pb;  cout<<pb.use\_count()<<endl;  cout<<pa.use\_cout()<<endl;  }  ine main()  {  fun();  return 0;  } |

可以看到fun函数pa，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa,pb析构时两个资源引用计数会减一，但是两者引用计数还是1，导致跳出函数时资源没有被释放（AB的析构函数没有被调用），如果把其中一个改为weak\_ptr就可以了，我们把类A里面的share\_ptr pb\_;改为weak\_ptr pb\_;运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放

注意的是我们不能通过weak\_ptr直接访问对象的方法，比如B对象中有一个方法print(),我们不能这样访问，pa->pb\_->print(),英文pb\_是一个weak\_ptr应该先把它转化为share\_ptr，如：share\_ptr p=pa->pb\_.lock();p->print();

**智能指针存在内存泄露情况**

当间隔对象相互使用一个share\_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄露，例如：

|  |
| --- |
| class Parent{  private:  std::share\_ptr<child> childptr;  public:  void setchild(std::share\_ptr<child> child){  this->childptr=child;  }  void doSomething(){  if (this->childptr.use\_count()){  }  }  ~parent(){}  }; |
| class child{  privatr:  std::share\_ptr<Parent> Parentptr;  public:  void setParent(std::share\_ptr<parent> parent){  this->Parentptr=parent;  }  void dosomething(){  if (this->Parentptr.use\_count())[  }  }  ~child{}  }; |
| int main(){  std::weak\_ptr<Parent> wpp;  std::weak\_ptr<child> wpc;  {  std::share\_ptr<Parent> p(new Parent);  std::share\_ptr<child> c(new child);  p->setchild(c);  c->setParent(p);  wpp=p;  wpc=c;  std::cout<<p.use\_count()<<std::endl; //2  std::cout<<c.use\_count()<<std::endl; //2  }  std::cout<<wpp.use\_count()<<std::endl; //1  std::cout<<wpc.use\_count()<<std::endl;//1  return 0;  } |

上述代码中，parent有一个share\_ptr类型的成员指向孩子，而child也有一个share\_ptr类型的成员指向父亲，然后在创建孩子和父亲对象时也使用了智能指针c和p，随后将c和p分别又赋值给child的智能指针成员paarent和parent的智能指针成员child，从而形成了一个循环引用

**如何解决智能指针内存泄露**

为了解决循环引用导致的内存泄露，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，其类似的一个普通指针，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。

**指针和数组的区别**

|  |  |
| --- | --- |
| 指针 | 数组 |
| 保存数据的地址 | 保存数据 |
| 间接访问数据，首先获得指针的内容，然后将其作为地址，从该地址中提取数据 | 直接访问数据 |
| 通常用于动态数据结构 | 通常用于固定数目且数据类型相同的元素 |
| 通过Malloc分配内存，free释放内存 | 隐式的分配和删除 |
| 通常指向匿名数据，操作匿名函数 | 自身即为数据名 |

**什么是野指针**

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

**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？**

将可能会被继承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孩子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄露

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

**fork函数**

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”的方式很是常见

在早期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按中。但从内核角度来说，逐页的复制方式时十分耗时的，现代的Unix系统采取了更多优化，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

**C++中析构函数的作用**

析构函数和构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕，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(),以区别与构造函数，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之前，释放掉申请的内存空间，避免内存泄露，

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

**静态函数和虚函数的区别**

静态函数在编译的时候已经确定运行时机，虚函数在运行的时候动态绑定，虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销

**重载和覆盖**

重载：两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中

重写：子类继承父类，父类中的函数是虚函数，在子类中重新定义了这个虚函数，这种情况是重写

**strcpy和strlen**

strcpy是字符串拷贝函数，原型：char \*strcpy(char \*dest,const char \*src);

从src逐字节拷贝到dest，直到遇到’\0’结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞，安全版本是strncpy函数，strlen函数是计算字符串长度的函数，返回从开始到’\0’之间的字符个数

**理解虚函数和多态**

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候已经确定，动态多态是用虚函数机制实现的，在运行期间动态绑定，举个例子，一个父类类型的指针指向一个子类对象的时候，使用父类的指针去调用子类中重写了父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数

虚函数实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段（.text）中当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到虚函数表中的地址替换为重新写的函数地址，使用了虚函数，会增加访问内存开销，降低效率。

**++i和i++的实现**

**++i实现**

|  |
| --- |
| int& int::operator++()  {  \*this+=1;  return \*this;  } |

**i++实现**

|  |
| --- |
| const int int::operator(int)  {  int oldValue=\*this;  ++(\*this);  return oldValue;  } |

**写一个函数，在main函数执行前先运行**

|  |
| --- |
| \_attribute ((constructor))void before()  {  printf(“before main\n”);  } |

**智能指针share\_ptr的实现**

核心要理解引用计数，什么时候销毁底层指针，还有赋值，拷贝构造时候的引用计数的变化，析构的时候要判断底层指针的引用计数为0才能真正释放底层指针的内存

|  |
| --- |
| template <typename T>  clatrss SmartPtr  {  private:  T \*ptr;//底层真实的指针  int \*use\_count;//保存当前对象被多少指针引用计数  public:  SmartPtr(T, \*p);//smartPtr <int> p(new int(2));  SmartPtr(const SmartPtr<T>&orig);//SmartPtr<int>q(p);  SmartPtr<T> &operator=(const SmartPtr<T> &rhs);//q=p  ~SmartPtr();  T operator\*();//为了能把智能指针当成普通指针操作定义解引用操作  T \*operator->();//定义取成员操作  T\* operator+(int i);//定义指针加一个常数  int operator-(SmartPtr<T>&t1,SmartPtr<T>&t2);//定义两个指针相减 |
| void getcount() {return \*use\_count}};  template <typename T> int SmartPtr<T>::operator-(SmarrPtr<T> &t1,SmartPtr<T> &t2){return t1.ptr-t2.ptr;}  template <typename T> SmartPtr<T>::SmartPtr(T \*P){ptr=p;try{use\_count=new int(1);}catch(...){ |
| delete ptr;//申请失败释放真实指针和引用计数的内存 |
| ptr=nullptr; delete use\_count; use\_count=nullptr;}} template <typename T> |
| SmartPtr <T>::SmartPtr(const SmartPtr<T> &orig)//复制构造函数  {  use\_count=orig.use\_count;//引用计数保存在一块内存，所有的SmartPtr对象的引用计数都指向这里  this->ptr=orig.ptr;  ++（\*use\_count);//当前对象的引用计数加一 |
| }template <typename T> SmartPtr<T> & SmartPtr<T>::operator=(const SmartPtr<T> &rhs){  //重载=运算符，例如SmartPtr<int>p,q;p=q;这个语句中，首先给q指向的对象的引用计数加1，因为p重新指向了q所指的对象，所有p需要先给原来的对象的引用计数减1，如果减一后为0，先释放掉p原来指向的内存，然后将q指向的对象的引用计数加1后赋值给P |
| ++\*(rhs.use\_count);if((--\*(use\_count))==0){delete ptr;ptr=nullptr;delete use\_count;use\_count=nullptr;}  ptr=rhs.ptr;\*use\_count=\*(rhs.use\_count);return \*this;}template <typename T> SmartPtr<T>::~SmartPtr(){getcount(); |
| if(--(\*use\_count)==0)//SmartPtr的对象会在其生命周期结束的时候调用其析构函数，在析构函数中检测当前对象的引用计数是不是只有正在结束生命周期的这个SmartPtr引用，如果是，就释放掉，如果不是，就还有其他的SmartPtr引用当前对象，就等待其他的SmartPtr对象在其生命周期结束的时候调用析构函数被释放掉 |
| {  getcount();  delete ptr;  ptr=nullptr;  delete use\_count;  use\_count=nullptr;  }  }  template (typename T>  T SmartPtr<T>::operator\*()  {  return \*ptr;  }  template <typename T>  T\* SmartPtr<T>::operator+(int i)  {  T \*temp+ptr+i;  return temp;  }  } |

**C++怎么定义常量的，常量存放在内存那个位置**

常量在C++里的定义就是一个top\_level const加上对象类型，常量定义必须初始化，对于局部对象，常量存放在栈区，对于全局变量，常量存放在全局、静态存储区，对于字面值常量，常量存放在常量存储区

const修饰成员函数的目的

const修饰的成员函数调用不会对对象做出任何改变，事实上，如果确认不会对对象做更改，就应该为函数加上const限定，这样无论const对象还是普通对象都可以调用该函数

**隐式类型转换**

首先对于内置类型，低精度的变量给高精度的变量赋值会发生隐式类型转换，其次，对于只存在单个函数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象

**C++函数栈空间的最大值**

默认是1M，不过可以调整

**说一说extern”C”**

C++调用C函数需要extern C，因为C语言没有函数重载

**new/delete与malloc/free的区别是什么**

首先new/delete是C++的关键字，而malloc/free是C语言的库函数，后者使用必须指明申请内存空间的大小，对于类类型的对象，后者不会调用构造函数和析构函数

**new和malloc的区别**

1.new分配内存按照数据类型进行分配,malloc分配内存按照指定的大小分配

2.new返回的是指定对象的指针，而malloc返回的是void\*,因此malloc的返回值一般都需要进行类型转化

3.new不仅分配一段内存，还会调用构造函数，malloc不会

4.new分配到内存要用delete销毁，malloc要用free来销毁，delete销毁的时候会调用对象的析构函数，而free不会

5.new是一个操作符可以重载，malloc是一个库函数

6.malloc分配的内存不够的时候，可以用realloc扩容，new没有这样的操作

7.new如果分配失败会抛出bad\_malloc的异常，而malloc失败会返回NULL

8.申请数组时，new[]一次分配所有内存，多次调用构造函数，搭配使用delete[],delete[]多次调用析构函数，销毁数组中的每个对象，而malloc则只能sizeof(int)\* n

**你了解的RTTI**

运行时类型检查，在C++层面主要体现在dynamic\_cast和typeid，VS中虚函数表的-1位置存放了指向type\_info的指针，对于存在虚函数的类型，typeid和dynamic\_cast都会区查询type\_info

**虚函数表具体是怎样实现运行时多态**

子类若重写父类虚函数，虚函数表中，该函数的地址会被替换，对于存在虚函数的类的对象，在VS中，对象的对象模型的头部存放指向虚函数表的指针，通过该机制实现多态

**C语言怎样进行函数调用的**

每一个函数调用都会分配函数栈，在栈内进行函数执行过程，调用前，先把返回地址压栈，然后把当前函数的esp指针压栈

**C语言参数压栈顺序**（从右到左）

**C++如何处理返回值**

生成一个临时变量，把他的引用作为函数参数传入函数内

**C++中拷贝赋值函数的形参能否进行值传递**

不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数。。如此循环。将无法完成拷贝，栈也会满

**说一说select**

select在使用前，先将需要监控的描述符对应的bit位置1，然后将其传给select，当有任何事件发生时，select将会返回所有描述符，需要在应用程序自己去遍历检查哪个描述符上有事件发生，效率很低，并且其不断在内核态和用户态进行描述符的拷贝，开销很大

**fork,wait,exec函数**

父进程产生子进程使用fork拷贝出来一个父进程的副本，此时只拷贝了父进程的页表，两个进程都读同一块内存，当有进程写的时候使用写实拷贝机制分配内存，exec函数可以加载一个GLF文件去替换父进程，从此父进程和子进程就可以运行不同的程序了，fork从父进程返回子进程的pid，从子进程返回0.调用了wait的父进程将会发生阻塞，直到有子进程状态改变，执行成功返回0，错误返回-1，exec执行成功则子进程从新的程序开始运行，无返回值，执行失败返回-1

**strcpy和strlen**

strcpy是字符串拷贝函数，原型：char \*strcpy(char \*dest,const char \*src);

从src逐字节拷贝到dest，直到遇到’\0’结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞，安全版本是strncpy函数

strlen函数是计算字符串长度的函数，返回从开始到’\0’之间的字符个数

**map和set有什么区别，分别又是怎么实现的**

map和set都是C++的关联容器，其底层实现都是红黑树（RB-Tree），由于map和set所开放是各种操作接口，RB-tree也都提供了，所以几乎所有的map和set区别在于

（1）map中的元素是key-value(关键字-值）对：关键字起到索引的作用，值则表示与索引相关联的数据，set与之相对就是关键字的简单集合，set中每个元素只包含一个关键字

（2）set的迭代器是const的不允许修改元素的值，map 允许修改value,但不允许修改key，其原因是因为map和set是根据关键字排序来保证其有序性的，如果允许修改key的话，那么首先需要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了map和set的结构，导致iterator失效，不知道应该指向改变前的位置，还是改变后的位置。所以STL中将set的迭代器设置成const，不允许修改迭代器的值，而map的迭代器则不允许修改key值，允许修改value值

（3）map支持下标操作，set不支持下标操作，map可以用key做下标，map是下标运算符[]将关键码作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和mapped\_type类型默认值的元素至map中，因此下标运算符[]在map应用中需要慎用，const\_map不能用，只希望确定某一个关键值是否存在而不希望插入元素时也不应该使用，mapped\_type类型没有默认值也不该使用，如果find能解决需要，尽可能用fine

**介绍一下STL的allocaotr**

STL的分配器用于封装STL容器在内存管理上的底层细节，在C++中，其内存分配和释放如下：

new运算分两个阶段：（1）调用：：operator new配置内存（2）调用对象构造函数构造对象内容

delete运算分两个阶段：（1）调用对象析构函数（2）掉员工：：operator delete释放内存

为了精密分工，STL allocator将两个阶段操作区分开来：内存配置有alloc::allocate()负责，内存释放由alloc::deallocate()负责，对象构造由::construct()负责，对象析构由::destroy()负责

同时为了提升内存管理的效率，减少申请小内存造成的内存碎片问题，SGI STL采用了两级配置器，当分配的空间大小超过128B时，会使用第一级空间配置器，当分配的空间大小小于128B时，将使用第二级空间配置器，第一级空间配置器直接使用malloc(),realloc(),free()函数进行内存空间的分配和释放，而第二级空间配置器采用了内存池技术，通过空闲链表来管理内存

**STL迭代器删除元素**

这个主要考察的是迭代器失效的问题。1.对于序列容器vector,deque来说，使用erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器；2.对于关联容器map set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用ERASE之前，记录下一个元素的迭代器即可，3.对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator,因此上面两种正确的方法都可以使用

**STL有什么基本组成**

STL主要由：以下几部分组成

容器迭代器仿函数算法分配器配接器

它们之间的关系：分配器给容器分配存储空间，算法通过迭代器获取容器中的内容，仿函数可以协助算法完成各种操作，配接器用来套接适配仿函数

**STL中map与unordered\_map**

1.Map映射，map的所有元素都是pair,同时拥有实值（value)和键值（key)。pair的第一元素被视为键值，第二元素被视为实值，所有元素都会根据元素的键值自动被排序，不允许键值重复

底层实现：红黑树

适用场景：有序键值对不重复映射

2.Multimap

多重映射。multimap的所有元素都是pair，同时拥有实值（value）和键值（key），Pair的第一元素被视为键值，第二元素被视为实值，所有元素都会根据元素的键值自动被排序，允许键值重复

底层实现：红黑树

适用场景：有序键值对可重复映射

**详细说vector和list的区别，应用**

1.vector：连续存储的容器，动态数组，在堆上分配空间

底层实现：数组

两倍容量增长：vector增加（插入）新元素时，如果未超过当时的容量，则还有剩余的空间，那么直接添加到最后（插入指定位置），然后调整迭代器

如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器会失效

性能：

访问：O（1）

插入：在最后插入（空间够）：很快

在最后插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝

在中间插入（空间够）：内存拷贝

在中间插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝

删除：在最后删除：很快

在中间删除：内存拷贝

适用场景：经常随机访问，且不经常对非尾节点进行插入删除

2.list

动态链表，在堆上分配空间，每插入一个元素都会分配空间，每删除一个元素都会释放空间

底层：双向链表

性能：

访问：随机访问性能很差，只能快速访问头尾节点

插入：很快，一般是常数开销

删除：很快，一般是常数开销

适用场景：经常插入删除大量数据

区别

1.vector底层实现是数组，list是双向链表

2.vector支持随机访问，list不支持

3.vector是顺序内存，list不是

4.vector在中间节点进行插入删除会导致内存拷贝，list不会

5.vector一次性分配好内存，不够才进行二倍扩容，list每次插入新节点都会进行内存申请

6.vector随机访问性能好，插入删除性能差，list随机访问性能差，插入删除性能好

应用

vector拥有一段连续的内存空间，因此支持随机访问，如果需要高效的随机访问，而不在乎插入和删除的效率，用vector

list拥有一段不连续的内存空间，如果需要高效的插入和删除，而不关心随机访问，用list

**STL中迭代器的作用，有指针为什么还要迭代器**

1.迭代器

iterator（迭代器）模式又称cursor(游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示，或者这样说可能更容易理解：literator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素

由于iterator模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如STL的list、vector、stack等容器类及ostream\_iterator等扩展iterator

2.迭代器和指针的区别

迭代器不是指针，是类模板，表现的像指针，它只是模拟了指针的一些功能，通过重载了指针的一些操作符，->/\*/++、--等，迭代器封装了指针，是一个“可遍历STL（standard template library)容器内全部或部分元素”的对象，本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，它可以根据不同类型的数据结构来实现不同的++、--等操作

迭代器返回的是对象引用，而不是对象的值，所以cout只能输出迭代器使用\*取值后的值而不能直接输出其自身

3.迭代器产生原因

iterator类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部结构而达到循环遍历集合的效果

**epoll的原理**

调用顺序

int epoll\_create(int size);

int epoll\_ctl(int epfd,int fd,struct epoll\_event\*event);

int epoll\_wait(int epfd,struct epoll\_event\*event,int maxevents,int timeout);

首先创建一个epoll对象，然后使用epoll\_ctl对这个对象进行操作，把需要监控的描述添加进去，这些描述如将会以epoll\_event结构体的形式组成一颗红黑树，接着阻塞在epoll\_wait,进入大循环，当某个fd上由事件发生时，内核将会把其对应的结构体放入一个链表中，返回有事件发生的链表

**STL迭代器是怎么删除元素的**

这个主要考察的是迭代器失效的问题

1.对于序列容器vector，deque来说，使用erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器

2.对于关联容器mao set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可

3.对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator，因此上面两种正确的方法都可以使用

**STL中MAP数据存放形式**

红黑树，unordered map底层结构是哈希表

n个整数的无序数组，找到每个元素后面比它大的第一个数，要求事件复杂度为O（N）

|  |
| --- |
| vector <int> findMax(vector<int>num)  {  if(num.size()==0) return num;  vector <int>res(num.size());  int i=0;  stack<int> s;  while(i<num.size())  {  if(s.empty()||num[s.top()]>=num[i])  {  s.push(i++);  }  else  {  res[s.top()]=num[i];  s.pop();  }  }  while(!s.empty())  {  res[s.top()]=INT\_MAX;  s.pop();  }  for(int i=0;i<res.size();i++)  cout<<res[i]<<endl;  return res;  } |

**STL里resize和reserve的区别**

resize():改变当前容器内含有元素的数量（size()),eg:vectoc<int> v;v.resize(len),v的size变为len，如果原来v的size小于len，那么容器新增（len-size)个元素，元素的值默认为0，当v.push\_back(3),之后，则是3放在了v的末尾，即下标为len，此时容器是size为len+1；

reserve():改变当前容器的最大容量（capacity），它不会生成元素，只是确定这个容器允许放入多少对象，如果reserve(len)的值大于当前的capacity(),那么会重新分配一块能存len个对象的空间，然后把之前v.size()个对象通过copy construtor复制过来，销毁之前的内存

测试代码如下

|  |
| --- |
| #include <iostream>  #include <vector>  using namespace std;  int main(){  vector <int> a;  a.reserve(100);  a.resize(50);  cout<<a.size()<<” “<<a.capacity()<<endl; //50 100  a.resize(150);  cout<<a.size()<<” “<<a.capacity()<<endl; //150 200  a.reserve(50);  cout<<a.size()<<” “<<a.capacity()<<endl; //150 200  a.resize(50);  cout<<a.size()<<” “<<a.capacity()<<endl; //50 200  } |

**说一说STL里面set和map怎么实现的**

集合，所有元素都会根据元素的值自动被排序，且不允许重复

底层实现：红黑树

set底层是通过红黑树（RB-tree）来实现，由于红黑树是一种平衡二叉搜索树，自动排序的效果很不错，所以标准的STL的set即以RB-tree为底层机制，又由于set所开放的各种操作接口，RB-tree也都提供了，所以几乎所有的set操作行为，都只有转调用RB-tree的操作行为而已

适用场景：有序不重复集合

map

映射，map的所有元素都是pair,同时拥有实值（value）和键值（key），pair 的第一元素被视为键值，第二元素被视为实值，所都会根据元素的键值自动被排序，不允许键值重复

底层：红黑树

适用场景：有序键值对不重复映射

**C++中类成员的访问权限**

C++通过public、protected、private三个关键字来控制成员变量和成员函数的访问权限，他们分别表示共有的，受保护的，私有的被称为成员访问限定符，在类的内部（定义类的代码内部），无论成员被声明为public、protected还是private,都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问Public属性的成员，不能访问private、protected属性的成员

**C++中struct和class的区别**

在C++中，可以用struct和class定义类，都可以继承，区别在于：strcutural的默认继承权限和默认访问权限是public，而class的默认继承权限和默认访问权限是private

另外，class还可以定义模板类形参，比如template<class T,int i>

**C++类内可以定义引用数据成员**

可以，必须通过成员函数初始化列表初始化

**回答什么是右值引用，跟左值又有什么区别**

右值引用是C++11中引入的新特性，它实现了转移语句和精确传递，它的主要目的有两个方面

1.消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率

2.能够更简洁明确的定义泛型函数

左值和右值的概念

左值：能对表达式取地址或具名对象/变量，一般指表达式结束后依然存在的持久对象

右值：不能对表达式取地址，或匿名对象，一般指表达式结束就不再存在的临时对象

右值和左值引用的区别

1.左值可以寻址，而右值不可以

2.左值可以被赋值，右值不可以被赋值，可以用来给左值赋值

3.左值可变，右值不可变（仅对基础类型适用，用户自定义类型右值引用可以通过成员函数改变）

**C++源文件从文本到可知执行文件经历的过程**

对于C++源文件，从文本到可执行文件一般需要四个过程

预处理阶段：对源代码文件中文件包括关系（头文件）、预处理语句（宏定义）进行分析和替换，生成预编译文件

编译阶段：将经过预处理后的预编译文件转换成特定的汇编代码，生成汇编文件

汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

**include头文件的顺序以及双引号“”和尖括号<>的区别**

Include头文件的顺序，对于include的头文件来说，如果在文件a.h中声明一个在文件b.h中定义的变量，而不引用b.h.那么要在a.c文件中引用b.h文件，并且要先引用b.h，后引用a.h，否则汇报变量类型未声明错误

双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样

对于使用双引号包含的头文件，查找头文件路径的顺序为：

当前头文件目录

编译器设置的头文件路径（编译器可使用-I显示指定搜索路径）

系统变量CPLUS\_INCLUDE\_PATH/C\_INCLUDE\_PATH指定的头文件路径

对于使用尖括号包含是头文件，查找头文件的路径顺序为：

编译器设置的头文件路径（编译器可使用-I显示指定搜索路径）

系统变量CPLUS\_INCLUDE\_PATH/C\_INCLUDE\_PATH指定的头文件路径

**说一下malloc的原理，另外brk系统调用和mmap系统系统调用的作用分别是什么**

Malloc函数用于动态分配内存，为了减少内存碎片和系统调用的开销，malloc其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作为内存管理的基本单位，当用户申请内存时，直接从堆区分配一块合适的空闲块。Malloc采用隐式链表结构将堆区分成连续的，大小不一的块，包含已分配块和未分配块，同时malloc采用显示链表结构来管理所有空闲块，即使用一个双向链表将空闲块链接起来，每一个空闲块记录一个连续的未分配的地址

当进行内存分配时，Malloc会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配，当进行内存合并时，malloc采用边界标记法，根据每个块的前后块是否已经分配来决定是否进行块合并

Malloc在申请内存时，一般会通过brk或者mmap系统调用进行申请，其中当申请内存小于128K时，会使用系统函数brk在堆区中分配，而当申请内存大于128K时，会使用系统函数mmap在映射区分配

C++的内存管理是怎样的

在C++中，虚拟内存分为代码段，数据段，BSS段，堆区，文件映射区以及栈区六部分

代码段：包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码

数据段：存储程序中已初始化的全局变量和静态变量

bss段：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为0的全局变量和静态变量

堆区：调用new/malloc函数时在堆区动态分配内存，同时需要调用delete/free来手动释放申请的内存

映射区：存储动态链接库以及调用mmap函数进行的文件映射

栈：使用栈空间存储函数的返回地址，参数，局部变量，返回值

**如何判断内存泄露**

内存泄露通常是由调用了malloc/new等内存申请的操作，但是缺少了对应的free/delete，为了判断内存是否泄露，我们一方面可以使用linux环境下的内存泄露检查工具Valgrind，另一方面我们在写代码时可以添加内存申请和释放的统计功能，统计当前申请和释放的内存是否一致，以此来判断内存是否泄露

**C++如何处理内存泄露**

使用varglind,mtrace

**什么时候会发生段错误**

段错误通常发生在访问非法内存地址的时候，具体来说分为以下几种情况

使用野指针

试图修改字符串常量的内容

**什么是memory leak,也就是内存泄露**

内存泄漏是指由于疏忽或错误造成的程序未能释放掉不再使用的内存情况，内存泄露并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费

**内存泄露的分类：**

1、堆内存泄露（Heap leak)对内存指的是程序运行中根据需要分配通过malloc,realloc,new等从堆中分配的一块内存，再是完成后必须通过调用对应的free或者delete删掉，如果程序的设计的错误导致这部分内存没有释放，那么此后这块内存将不会被使用，就会产生Heap Leak

2、系统资源泄露（Resource Leak）主要指程序使用系统分配的资源比如Bitmap,handle,SOCKET等没有使用相应的函数释放掉，导致系统资源的浪费，严重可能导致系统效能降低，系统运行不稳定

3、没有基类的析构函数定义为虚函数，当基类指针指向子类对象时，如果基类的析构函数不是virtual,那么子类的析构函数将不会被调用，子类的资源没有正确释放，因此造成内存泄露

**说一下内存相关的api**

linux允许不同进程访问同一个逻辑内存，提供了一组API，头文件再sys/shm.h中

1）新建共享内存shmget

int shmget(key\_t key,size\_t size,int shmflg);

key:共享内存键值，可以理解为共享内存的唯一性标记

size：共享内存大小

shmflag:创建进程和其他进程的读写权限标识。

返回值：相应的共享内存标识符，失败返回-1

2）链接共享内存到当前进程的地址空间shmflg

void \*shmat(int shm\_id,const void \*shm\_addr,int shmflg);

shm\_id:共享内存标识符

shm\_addr:指定共享内存连接到当前进程的地址，通常为0，表示由系统来选择

shmflg :标志位

返回值：指向共享内存第一个字节的指针，失败返回-1

3）当前进程分离共享内存shmdt

int shmdt(const void \*shmaddr);

4)控制共享内存shmctl

和信号量的semctl函数类似，控制共享内存

int shmctl（int shm\_id,int command,struct shmid\_ds \*buf);

shm\_id:共享内存标识符

command:有三个值

IPC-STAT:获取共享内存的状态，把共享内存的shmid\_ds结构复制到buf中

IPC\_SET：设置共享内存状态，把buf复制到共享内存的shmid\_ds结构

IPC\_RMID:删除共享内存

buf：共享内存管理结构体

**如何采用单线程的方式处理高并发**

在单线程模型中，可以采用I/O复用来提高单线程处理多个请求的能力，然后再采用事件驱动模型，基于异步回调来处理事件来

**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，它们各自又是怎么同步的**

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，指令处理器和处理器状态，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），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，一个进程可以含有多个线程，但至少有一个线程。线程依赖于进程而存在

2.进程再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，同意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：由于在创建或撤销进程时，系统都要为之分配或撤销资源，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销线程时的开销，类似的，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到开销

5.通信：由于同一进程中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，同步和通信都无需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，开销少，切换速度快，但编程调试相对复杂

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.进程适用于多核，多机分布；线程适用于多核

有了进程，为什么还需要线程

**进程间通信方式：**

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）他是半双工的（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的系统，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，是消息的连接表，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）具有写权限的进程可以按照一定的规则

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于接受进程，进程终止时，消息队列及其内容并不会被删除。

3）消息队列可以实现信息的随机查询，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一减一，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

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的更新，这种方式需要依靠某种同步操作，如互斥锁和信号等

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，与其它通信机制不同的是，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

**Linux虚拟地址空间**

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，不同进程只需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.虚拟内存的管理需要建立很多数据结构，这些数据结构要占用二额外的内存

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

**操作系统中程序的内存结构**

一个程序本质上是由BSS段，data段，text段三个组成的，可以看到一个可执行的程序在存储时（没有调入内存）分为代码段，数据区和未初始化数据区三个部分

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段并不占有可执行文件的大小，它是由连接器来获取内存的

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(已经初始化是数据）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它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，栈区是从高地址像低地址位增长的，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就会提示溢出，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。

说一说操作系统中的缺页中断

malloc()和mmap()等内存分配函数，在分配时只是建立了进程虚地址空间，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时，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.缺页中断返回时，执行产生中断的一条指令，而一般的中断返回时，执行下一条指令

**回答fork和vfork的区别**

fork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（）会创建一个新进程，它几乎与调用fork（）的进程一样，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，逐页的复制方式是什么耗时的，现代的unix系统采用了更多的优化，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，BSS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即执行一次对exec的系统调用，或者调用\_exit()退出，对vfork（）的成功调用并所产生的结果和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，vfok（）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。所以这就是名字来源：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

写时复制在内核中实现的方式很简单，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

f**ork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

**如何修改文件最大句柄数**

Linux默认最大文件句柄数是1024个，在Linux服务器文件并发量比较大的情况下，系统就会报“too many open files”的错误。故在Linux服务器高并发调优时，往往需要预先调优Linux参数，修改Linux最大文件句柄数

有两种方法：

1.ulimit-n<可以同时打开的文件数>，将当前进程的最大句柄数修改为指定的参数（注：该方法只针对当前进程有效，重新打开一个shell或者重新开启一个进程，参数还是之前的值）

首先用ulimit-a查询Linux相关操作，如下

|  |
| --- |
| core file size (blocks,-c) 0  data seg size (kbytes,-d) unlimited  scheduling priority (-e) 0  file size (blocks ,-f) unlimited  pending signals (-i) 94739  max locked memory (kbytes,-1) 64  max memory size (kbytes,-m) unlimited  open files (-n) 1024  pipe size (512 bytes,-p) 8  POSIX message queues (bytes,-q) 8192  real-time priority (r) 0  stack size (kbytes,-s) 8192  cpu time (seconds,-t) unlimited  max user processes (-u) 94739  virtual memory (kbytes,-v) unlimited  file locks (-x) unlimited |

其中，open file就是最大文件句柄数，默认时1024个

修改Linux最大文件句柄数：ulimit-n 2048,将最大句柄数修改为2048

2.对所有进程都有效的方法，修改Linux系统参数

vi/etc/security/limits.conf添加

。。。 sort nofile 65536

。。。 hard nofile 65536

将最大句柄数改为65536

修改以后保存，注销当前用户，重新登录，修改后的参数就生效了

**说一说并发（concurrency)和并行(parallelism)**

并发（concurrency）：指宏观上看起来两个程序子啊同时运行，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，我的指令也穿插着你的，在单个周期内值运行了一个指令。这种并发并不能提高计算机的性能，只能提高效率

并行（parallelism):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都往多核发展

**请问MYSQL的端口号多少，如何修改这个端口号**

查看端口号：

使用命令show global variables like ‘port’;查看端口号，mysql的默认端口是3306.（补充：sqlserver默认端口号为：1433；oracle默认端口号为：1521；DB2默认端口号为：5000；postgreSQL默认端口号为：5432）

修改端口号：

修改端口号：编辑、etc/my.cnf文件，早期版本有可能是my.conf文件名，增加端口参数，并且设定端口，注意该端口未被使用，保存退出

**操作系统中的页表寻址**

页式内存管理，内存分成固定长度的一个个页片。操作系统为每一个进程维护了一个从虚拟地址到物理地址的映射关系的数据结构，叫页表，页表的内容就是该进程的虚拟地址到物理地址的一个映射，页表中的每一项都记录了这个页的基地址，通过页表，由逻辑地址的高位部分先找到逻辑地址对应的页基地址，再由页基地址偏移一定长度就得到了最后的物理地址，偏移的长度由逻辑地址的低位部分决定。一般情况下，这个过程都可以由硬件完成，所以效率还是比较高的，页式内存管理的优点就是比较灵活，内存管理以较小的页为单位，方便内存换入换出和扩充地址空间

Linux最初的两级页表机制

两级分页机制将32位的虚拟空间分成三段，低十二位表示页面偏移，高20分成两段分别表示两级页面的偏移

\*PGD(page global directory):最高10位，全局页目录表索引

\*PTE(page table entry)中间10位，页表入口索引

当在进行地址转换时，结合在CR3寄存器中存放的页目录（page directory,PGD),d 这一页的物理地址，再加上从虚拟地址中抽出高10位叫做页目录表项（内核也称为pgd）的部分作为偏移，即定位到可以描述该地址的Pgd，从该pgd中可以获取可以描述该地址的页表的物理地址，再加上从虚拟地址中抽取中间的10位作为偏移，即定位到可以描述该地址的pte；在这个pte中即可获取该地址对应的页的物理地址，加上从虚拟地址中抽取的最后12位，即形成该页的页内偏移，即可最终完成从虚拟地址到物理地址的转换。从上述过程中，可以看出，对虚拟地址的分级解析过程，实际上就是不断深入页表层次，逐渐定位到最终地址的过程，所以这一过程被叫做page table walk

linux的三级页表机制

当X86引入物理地址扩展（pisycal addrress extension,PAE)后。可以支持大于4G的物理内存（36G），但虚拟地址依然是32位，原先的页表项不适用，它实际多4bytes被扩充到8bytes,这意味着，每一页现在能存放的Pte数目从1024变成了512了（4k/8）相应的，页表层级发生变化，Linus新增加了一个层级，叫做页中间目录（page middle directory,PMD),变成

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 位数 |
| cr3 | 指向一个PDPT | crs寄存器存储 |
| PGD | 指向PDPT中4个项中的一个 | 位31~30 |
| PMD | 指向页目录中512项中的一个 | 位29~21 |
| PTE | 指向页表512项中的一个 | 位20~12 |
| page offset | 4KB页中的偏移 | 位11~0 |

现在就停同时在2级页表和3级页表，在代码管理上肯定不方便。巧妙的是，Linux采取了一种抽象方法，所有架构全部使用三级页表：即PGD->PMD->PTE。那只使用2级页表（如非PAE的X86）怎么办

办法是针对使用2级页表的架构，把PMD抽象掉，即虚设一个PMD表项，这样在page table walk过程中，PGD本是直接指向PTE的，现在不了，指向一个虚拟的PMD,然后再由PMD指向PTE，这种抽象保持了代码结构的统一

Linux的四级页表机制：

硬件在发展，3级页表很快就捉襟见肘了，原因是64位CPU出现了，比如X86\_64，它的硬件实实在在支持4级页表的。它支持48位的虚拟地址空间，如下

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 位数 |
| PML4 | 指向一个PDPT | 位47~39 |
| PGD | 指向PDPT中4个项中的一个 | 位38~30 |
| PMD | 指向页目录中512项中的一个 | 位29~21 |
| PTE | 指向页表中512项中的一个 | 位20~12 |
| page offset | 4KB页中的偏移 | 位11~0 |

Linux内核针为使用原来的3级列表（PGD->PMD->PTE)，做了折中，即采用了一个唯一的，共享的顶级层次，叫PML4没有编码在地址中，这样就能套用原来的3级列表方案了，不过代价就是，由于只有唯一的PML4，寻址空间被局限在（239=）512G，而本来PML4段有9位，可以支持512个PML4表项的，现在为了使用3级列表方案，只能限制使用一个，512G的空间很快就不够用了，解决方案呼之欲出

在2004年10月，当时的X86\_64架构代码的维护者andi kleen提交了一个叫做4level page table for linux的PATCH系列，为Linux内核带来了4级页表的支持。在它的解决方案中，不出意料地，按照X86\_64规范，新增了一个PML4的层级，在这种解决方案中，X86\_64用一个有512条目的PML4,512条目的PGD,512条目的PMD,512条目的PTE。对于仍使用3级目录的架构来说，它们依然拥有一个虚拟的PML4，相关的代码会在编译时被优化掉，这样，就把Linux内核的3级列表扩充为4级列表。这系列PATCH工作的不错，不久被纳入andrew mortond 的-mm树接受测试，不出意外的话，他将在v2.6.11版本中释出，但是，另一个知名开发者nick piggin提出了一些看法，他认为andi的patch很不错，不过他认为最好的还是把PGD作为第一级目录，把新增加的层次放在中间，并且给出了他自己的patch:alternate 4-level page tables patches.andi更想保持自己的PATCH，他认为你Nick不过是玩了改名的游戏，而且它的PATCH经过测试很稳定，快被合并到主线了，不宜再折腾。不过Linus却表达了对nick piggin的支持，理由是Nick的做法conceptually least intrusive.毕竟作为Linux的扛把子，稳定对于Linux来说意义重大，最终，不出意外的，最后Nick piggin的PATCH在v2.6.11版本中被合并入主线，在这种方案中，4级页表分别是PGD->PUD->PMD-.PTE

**在单核机器上写多线程程序，是否需要考虑加锁，为什么**

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这些寄存器是干什么用的**

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

**说一说线程间的同步方式，最好说出具体的系统调用**

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，将他减一，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一，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

互斥量又称为互斥锁，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互斥锁加锁，如果目标互斥锁以及被上锁，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，然后接受信号后会再加锁，保证该线程对共享资源正确访问

**说一下多线程和多进程的区别**

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，切换和销毁对开销比较大，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**

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

**说一说OS缺页置换算法**

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，当前操作系统最常用的缺页置换算法如下

先进先出（FIFO）算法：置换最近一段时间以来最长时间未访问过的页面，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

**说一说死锁发生的条件以及如何解决死锁**

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互相等待的现象，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，此时请求堵塞，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

**请问虚拟内存和物理内存怎么对应**

1.概念

物理地址（physical address)

用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应

虽然可以直接把物理地址理解成插在机器上那根内存本身，把内存看成一个从0字节一直到最大空量逐字节的编号的大数组，然后把这个数组叫做物理地址，但是事实上，这只是一个硬件提供给软件的抽象，内存的寻址方式并不是这样，所以，说他是“与地址总线相对应”是更贴切一些，不过抛开对物理内存寻址方式的考虑，直接把物理的内存一一对应，也是可以接受的，也许错误的理解更利于形而上的抽象

虚拟地址（virtual memory）

这是对整个内存（不要与机器上插那条对上号）的抽象描述，它是相对于物理内存来讲的，可以直接理解成“不真实的”,“假的”内存，例如，一个0x08000000内存地址，它并不对就物理地址上那个大数组中0x08000000-1那个地址元素：

之所以是这样，是因为现代操作系统都提供了一种内存管理的抽象，即虚拟内存（virtual memory)。进程使用虚拟内存中的地址，由操作系统协助相关硬件，把他“转换”成真正的物理地址，这个“转换”是所有问题讨论的关键

有了这样的抽象，一个程序，就可以使用比真实物理地址大的多的地址空间，甚至多个进程可以使用相同的地址，不奇怪，因为转换后的物理地址并非相同

可以把连接后的程序反编译看一下，发现连接器已经为程序分配了一个地址，例如，要调用某个函数A，代码不是call A，而是call 0x0811111111,也就是说，函数A的地址已经被定下来了，没有这样的“转换”，没有虚拟地址的概念，这样做根本是行不通的

2.地址转换

第一步：CPU段式管理中——逻辑地址转线性地址

CPU要利用其段式内存管理单元，先将每个逻辑地址转换成一个线性地址

一个逻辑地址由两部分组成，【段标识符：段内偏移量】

段标识符是由一个16位长的字段组成，称为段选择符。其中前13位是一个索引号，后面2位包含一些硬件细节，如图：

通过段标识符中的索引号从GDT或者LDT找到该段的段描述符，段描述符中的base字段是段的起始地址

段描述符：base字段，它面熟了一个段的开始位置的线性地址

一些全局的段描述符，就放在“全局段描述符表（GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符（LDT）”中

GDT在内存中的地址和大小存放在CPU的gdtr控制寄存器中，而LDT则在idtr寄存器中

段起始地址+段内偏移量=线性地址

**请你说说操作系统中结构体对齐，字节对齐**

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）应该尽可能在自然边界上对齐，原因在于，为了访问未对齐内存，处理器需要两次内存访问，而对齐的内存访问只需要一次访问

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）本身也要仅从对齐，对齐按照#pragma pack指定的数值或结构（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：如果一个结构体里有某些结构体成员，则结构体成员要从其内部最大元素的整数倍地址来时存储

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; //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() |

进程间怎么通信

进程间通信主要包括

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

一、进程的五种状态

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

二、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，处理及就会空闲，I/O速度比处理机慢很多，可能出现所有进程堵塞等待I/O

针对以上问题，提出两种解决办法：

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，或进程所需的出局和程序换入到内存

从而出现了进程挂起状态：进程被交换到外存，进程状态就成为了挂起状态

三、活动阻塞，禁止阻塞，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yunx

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）

请你回答一下软连接和硬链接区别

为了解决文件共享问题，Linux引入了软连接和硬链接，除了为Linux解决文件共享使用，还带来了隐藏文件路径，增加权限安全及节约存储等好处

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IN创建，若文件用户数据块中存放的内容时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

什么是大端小端以及如何判断大端小端

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

|  |
| --- |
| //判断系统是大端还是小端：通过联合体，因为联合体的所有成员都从低地址开始存放  int fun()  {  union test  {  int i;  char c;  };  text t;  t.i=1;  //如果是大端，则他t.c为0x00，则t.c!=1,返回0 ，是小端，则t.c为0x01,则t.c==1,返回1  return (t.c==1);  } |

回答一下静态变量什么时候初始化

静态变量存储在虚拟地址空间的数据段和BSS段，C语言中其在执行代码前初始化，属于编译期初始化二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

说一下用户态和内核态区别

用户态和内核态是操作系统的两种运行级别，两者最大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

如何设计server，使得能够接收多个客户端的请求

多线程，线程池，io复用

死循环+来连接时新建线程的方法效率有点低，怎么改进

提前创建好一个线程池，用生产者消费者模型，创建一个任务队列，队列作为临界资源，有了新连接，就挂在到任务队列上，队列为空所有线程睡眠，改进死循环：使用select epoll这样的技术

怎么唤醒被阻塞的socket线程

给阻塞时候缺少的资源

怎样确定当前线程是繁忙还是阻塞

使用ps命令查看

请问就绪状态的进程在等待什么

被调度使用cpu的运行权

说一说多线程的同步，锁的机制

同步的时候用一个互斥量，在访问共享资源前对互斥量进程加锁，在访问完成后释放互斥量上的锁，对互斥量进行加锁之后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。如果释放互斥锁时有多个线程阻塞，所有在该互斥锁上的阻塞线程都会变成可运行状态，第一个变为运行状态的线程可以对互斥量加锁，其他线程将会看到互斥锁依然被锁住，只能回去再次等待它重新变为可用。在这种方式下，每次只有一个线程可以向前执行

两个进程访问临界区资源，会不会出现都获得自旋锁的情况

单核CPU，并且开了抢占可以造成这种情况

windows消息机制知道吗，请说一说

当用户有操作（鼠标，键盘等）时，系统会将这些时间转化为消息，每个打开的进程系统都为其维护了一个消息队列，系统会将这些信息放到进程的消息队列中，二应用程序

**计算机网络一**

**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**

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的数据序列号，如果发送方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。这段时间一般是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，确认应答都会不停的方式序号为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，会疯狂向他提醒

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，一般开始都是设为65536，拥塞避免是指党拥塞窗口达到一个阈值，用死啊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），以此来不免拥塞

将报文段的超时重传看作拥塞，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

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,并将该数据包发送给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。

四次握手

由于TCP连接是双全工的，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受服务器发来的数据

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

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3MSL(MSL:报文段最大生存空间），然后关闭连接