2022 年中兴精选 50 面试题及答案

1. 优先队列时间复杂度。

优先级队列用堆实现，只是需要构建初始堆，这个时间复杂度是 O(n) 插入和删除只是修改了堆顶和堆底，不需要所有的都排序，只是需要再次调整好堆，因此时间复杂度都是 O(log2n).

2. 堆的维护时间复杂度。

假如有 N 个节点，那么高度为 H=logN，最后一层每个父节点最多只需要下调 1 次，倒数第二层最多只需要下调 2 次，顶点最多需要下调 H 次，而最后一层父节点共有 2^(H-1) 个，倒数第二层公有 2^(H-2)，顶点只有 1(2^0)个，所以总共的时间复杂度为 s = 1 * 2^(H-1) + 2 * 2^(H-2) + ... + (H-1) * 2^1 + H * 2^0 将 H 代入后 s= 2N - 2 - log2(N)，近似的时间复杂度就是 O(N)。

3. CPU 是怎么执行指令的？

计算机每执行一条指令都可分为三个阶段进行。即取指令-----分析指令-----执行指令。

取指令的任务是：根据程序计数器 PC 中的值从程序存储器读出现行指令，送到指令寄存器。

分析指令阶段的任务是：将指令寄存器中的指令操作码取出后进行译码，分析其指令性质。如指令要求操作数，则寻找操作数地址。

计算机执行程序的过程实际上就是逐条指令地重复上述操作过程，直至遇到停机指令可循环等待指令。

一般计算机进行工作时，首先要通过外部设备把程序和数据通过输入接口电路和数据总线送入到存储器，然后逐条取出执行。但单片机中的程序一般事先我们都已通过写入器固化在片内或片外程序存储器中。因而一开机即可执行指令。

下面我们将举个实例来说明指令的执行过程：

开机时，程序计算器 PC 变为 0000H。然后单片机在时序电路作用下自动进入执行程序过程。执行过程实际上就是取出指令（取出存储器中事先存放的指令阶段）和执行指令（分析和执行指令）的循环过程。

例如执行指令：MOV A,#0E0H，其机器码为“74H E0H”，该指令的功能是把操作数 E0H 送入累加器，

0000H 单元中已存放 74H，0001H 单元中已存放 E0H。当单片机开始运行时，首先是进入取指阶段，其次序是：

1 程序计数器的内容（这时是 0000H）送到地址寄存器；

2 程序计数器的内容自动加 1（变为 0001H）；

3 地址寄存器的内容（0000H）通过内部地址总线送到存储器，以存储器中地址译码电

跟，使地址为 0000H 的单元被选中；

4 CPU 使读控制线有效；

5 在读命令控制下被选中存储器单元的内容（此时应为 74H）送到内部数据总线上，因为是取指阶段，所以该内容通过数据总线被送到指令寄存器。至此，取指阶段完成，进入译码分析和执行指令阶段。

由于本次进入指令寄存器中的内容是 74H（操作码），以译码器译码后单片机就会知道该指令是要将一个数送到 A 累加器，而该数是在这个代码的下一个存储单元。所以，执行该指令还必须把数据（E0H）从存储器中取出送到 CPU，即还要在存储器中取第二个字节。其过程与取指阶段很相似，只是此时 PC 已为 0001H。指令译码器结合时序部件，产生 74H 操作码的微操作系列，使数字 E0H 从 0001H 单元取出。因为指令是要求把取得的数送到 A 累加器，所以取出的数字经内部数据总线进入 A 累加器，而不是进入指令寄存器。至此，一条指令的执行完毕。单片机中 PC="0002H"，PC 在 CPU 每次向存储器取指或取数时自动加 1，单片机又进入下一取指阶段。这一过程一直重复下去，直至收到暂停指令或循环等待指令暂停。

CPU 就是这样一条一条地执行指令，完成所有规定。

4. 什么函数不能声明为虚函数?

常见的不能声明为虚函数的有普通函数（非成员函数）、静态成员函数、内联成员函数、构造函数和友元函数。以下将分别对这几种情况进行分析。

1）普通函数（非成员函数）只能 overload（重载），不能被 override（覆盖），不能 被声明为虚函数，因此，编译器会在编译时绑定函数。

2）静态成员函数不能是虚函数，因为静态成员函数对于每个类来说只有一份代码，所有的对象都共享这一份代码，它不归某个对象所有，所以，它也没有动态绑定的必要性。

3）内联成员函数不能是虚函数，因为内联函数本身就是为了在代码中直接展开，减少函数调用花费的代价而设立的，而虚函数是为了在继承后对象能够准确地执行自己的动作，这是不可能统一的。再说，inline 函数在编译时被展开，虚函数在运行时才能动态地绑定函数。

4）构造函数之所以不能是虚函数，因为构造函数本来是为了明确初始化对象成员才产生的，然而虚函数主要是为了在不完全了解细节的情况下也能正确处理对象。另外，虚函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用虚函数来完成你想完成的动作？

5）友元函数。C++语言不支持友元函数的继承，对于没有继承特性的函数没有虚函数的说法。友元函数不属于类的成员函数，不能被继承。所以，友元函数不能是虚函数。

5. 在函数内定义一个字符数组，用 gets 函数输入字符串的时候，如果输入越界，为什么程序会崩溃？

因为 gets 无法截断数组越界部分，会将所有输入都写入内存，这样越界部分就可能覆盖其他内容，造成程序崩溃。

6. 线上 CPU 爆高，请问你如何找到问题所在。

1、top 命令：Linux 命令。可以查看实时的 CPU 使用情况。也可以查看最近一段时间的 CPU 使用情况。
2、PS 命令：Linux 命令。强大的进程状态监控命令。可以查看进程以及进程中线程的当前 CPU 使用情况。属于当前状态的采样数据。
3、jstack：Java 提供的命令。可以查看某个进程的当前线程栈运行情况。根据这个命令的输出可以定位某个进程的所有线程的当前运行状态、运行代码，以及是否死锁等等。
4、pstack：Linux 命令。可以查看某个进程的当前线程栈运行情况。

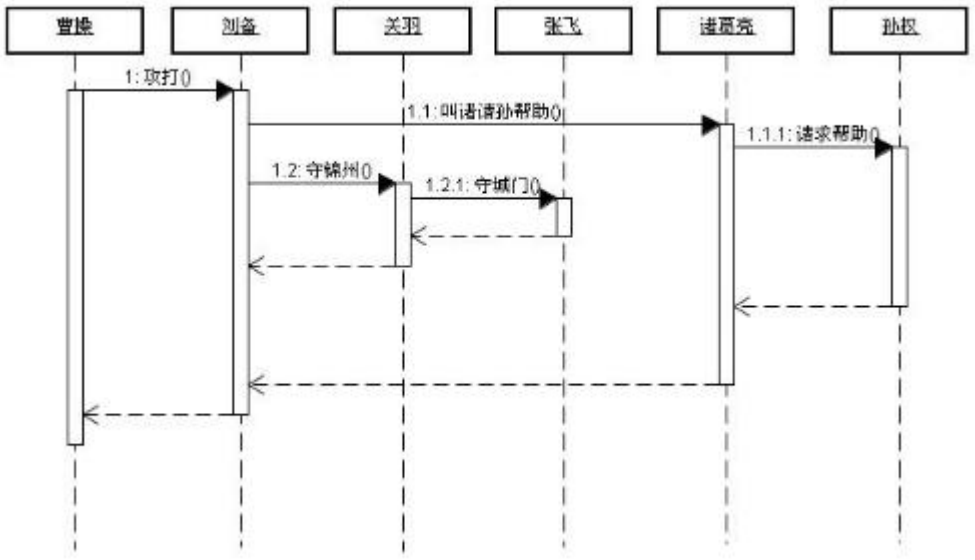
7. 从 innodb 的索引结构分析，为什么索引的 key 长度不能太长？

key 太长会导致一个页当中能够存放的 key 的数目变少，间接导致索引树的页数目变多，索引层次增加，从而影响整体查询变更的效率。

8. MySQL 的数据如何恢复到任意时间点？

恢复到任意时间点以定时的做全量备份，以及备份增量的 binlog 日志为前提。恢复到任意时间点首先将全量备份恢复之后，再此基础上回放增加的 binlog 直至指定的时间点。

9. 曹操南下攻打刘备,刘备派关羽守锦州,关羽派张飞去守城门。刘备又派诸葛亮去向孙权求援。孙权派兵攻打曹操.请画出 UML 图.



10. 信号的生命周期？

信号产生-》信号在进程中注册-》信号在进程中的注销-》执行信号处理函数

11. 信号的产生方式？

（1）当用户按某些终端键时产生信号
（2）硬件异常产生信号【内存非法访问】
（3）软件异常产生信号【某一个条件达到时】
（4）调用 kill 函数产生信号【接受和发送的所有者必须相同，或者发送的进程所有者必须为超级用户】（5）运行 kill 命令产生信号

12. 信号处理方式？

（1）执行默认处理方式
（2）忽略处理
（3）执行用户自定义的函数

13. 红黑树如何插入和删除的？

插入：

（1）如果父节点为黑色，直接插入不处理

（2）如果父节点为红色，叔叔节点为红色，则父节点和叔叔节点变为黑色，祖先节点变为红色，将节点操作转换为祖先节点

（3）如果当前节点为父亲节点的右节点，则以父亲结点为中心左旋操作

（4）如果当前节点为父亲节点的左节点，则父亲节点变为黑色，祖先节点变为红色，以祖先节点为中心右旋操作

删除：

（1）先按照排序二叉树的方法，删除当前节点，如果需要转移即转移到下一个节点

（2）当前节点，必定为这样的情况：没有左子树。

（3）删除为红色节点，不需要处理，直接按照删除二叉树节点一样

（4）如果兄弟节点为黑色，兄弟节点的两个子节点为黑色，则将兄弟节点变为红色，将着色转移到父亲节点

（5）如果兄弟节点为红色，将兄弟节点设为黑色，父亲结点设为红色节点，对父亲结点进行左旋操作

（6）如果兄弟节点为黑色，左孩子为红色，右孩子为黑色，对兄弟节点进行右旋操作

（7）如果兄弟节点为黑色，右孩子为红色，则将父亲节点的颜色赋值给兄弟节点，将父亲节点设置为黑色，将兄弟节点的右孩子设为黑色，对父亲节点进行左旋

14. 输入某班级学生的姓名、分数，并对分数进行降幂排列并输出；

```
#include <iostream>
using namespace std;

struct Node
{
	char name[100];
	int score;
	Node *next;
};

void show(Node *head)
{
	while(head)
	{
		cout<<head->name<<" "<<head->score<<endl;
		head = head->next;
	}
```

```
}

int linkLen(Node *head)
{
	int len = 0;
	while(head)
	{
		len++;
		head = head->next;
	}
	return len;
}
void sort(Node *head)
{
	int len = linkLen(head);
	for(int j=len-1; j>0; j--)
	{
		Node *tempHead = head;
		for(int i=0; i<j; i++)
		{
			if(tempHead->score> tempHead->next->score)
			{
				char tempName[100];
				int tempScore;
				strcpy(tempName,tempHead->next->name);
				strcpy(tempHead->next->name,
					tempHead->name);
				strcpy(tempHead->name, tempName);
				tempScore = tempHead->next->score;
				tempHead->next->score = tempHead->score;
				tempHead->score = tempScore;
			}
		}
	}
}
int main()
{
	char name[100];
	int score;
	Node *head = NULL;
	Node *tail = NULL;
	scanf("%s %d", name, &score);
	while(score != -1)
	{
```

```
                Node *temp = new Node;
                strcpy(temp->name, name);
                temp->score = score;
                temp->next = NULL;
                if(head == NULL)
                {
                        head = temp;
                        tail = temp;
                }
                else
                {
                        tail->next = temp;
                        tail = temp;
                }
                scanf("%s %d", name, &score);
        }
        show(head);
        sort(head);
        show(head);
        return 0;
}
```

15. 随即产生一字符串，每次产生的字符串内容、长度都不同；

```
#include <iostream>
using namespace std;

int main()
{
        //可见字符的范围为 32～126
        srand(time(0));
        int len = rand()%1000;
        char *a = new char[len+1];
        for(int i=0; i<len; i++)
        {
                a[i] = char(rand()%95 + 32);
        }
        a[len] = '\0';
        cout<<a<<endl;
        return 0;
}
```

16. 设 X[1..n]和 Y[1..n]为两个数组每个都包含 n 个已排好序的数。请使用伪代码给出一个求数组 X 和 Y 中所有 2n 个元素的中位数的 O(lgn)时间的算法。

数组 X 有 n 个元素，数组 Y 有 n 个元素，且都是从小到大排好的。那么找中位数是找第(2n+1)/2 个元素是哪个。

一共有 2n 的元素，取各自数组中间的一个第（n－1）/2 个元素，设为 Xmid 和 Ymid（下标为 0 到 n-1）

1）n 为奇数的时候，对于数组 X 和数组 Y 有(n-1)/2 个元素在第 Xmid 和 Ymid 之前，现在假设 a.Xmid>Ymid 这个时候如果两个数组有序的排好后，会有至少(n-1)/2*2+1(这个 1 为 Ymid) ＝ n 个元素在 Xmid 之前也就是说，这个待寻找的中位数也在 Xmid 之前，也就是说这个中位数不可能在 X 数组 Xmid 元素之后，所以可以排除一半 X 数组中的元素。而且最多会有(n-1)/2*2 = n-1 个元素在 Ymid 之前,所以这个中位数肯定不会比 Ymid 小，所以不会出现在 Y 数组 Ymid 之前，排除一半 Y 中的元素(同理 Xmid<Ymid 的时候也是这样分析)b.Xmid = Ymid 这个就比较好了，中位数就是 Xmid 了。

2）n 为偶数的时候，也取 Xmid =X[(n-1)/2]

a.Xmid>Ymid，这个时候各自有(n-2)/2 个元素在 Xmid 和 Ymid 元素之前，那么如果这两个数组排序，会有至少(n-2)/2*2+1=n-1 个元素在 Xmid 之前，如果为偶数那么中位数为第(2n+1)/2=n 个元素，这个时候 Xmid 至少为第 n 个元素，所以这个中位数也不可能出现在 Xmid 这个元素之后。至多会有 n-2 个元素在 Ymid 之前，所以中位数肯定在 Ymid 之后。同样排除了一半的 X 和 Y 中的元素。

17. static_cast<>,dynamic_cast<>,const_cast<>,reinterpret_cast<>的各自作用和使用环境？

static_cast：能完成大部分转换功能，但是并不确保安全

const_cast：无法从根本上转变类型，如果是 const，它就依旧是 const,只是如果原对象不是 const,可以通过此转换来处理，针对指针和引用而言。

dynamic_cast：针对基类和派生类指针和引用转换，基类和派生类之间必须要继承关系，是安全的

reinterpret_cast：允许将任何指针类型转为其他指针类型，是安全的

18. 出现异常时，try 和 catch 做了什么？

Catch(Ep a)发生异常-》建立一个异常对象-》拷贝一个异常对象-》catch 处理

Catch(Ep &a)发生异常-》建立一个异常对象-》引用异常对象-》catch 处理

异常对象通常建立在全局或者堆中【需要在函数外进行捕捉】

Catch 捕捉异常的转换：异常处理时，如果用基类的处理派生类的对象会导致派生类完全当做基类来使用，即便有虚函数也没用，所以派生类必须放在基类前处理。

19. 为什么要字节对齐？

（1）有些特殊的 CPU 只能处理 4 倍开始的内存地址
（2）如果不是整倍数读取会导致读取多次
（3）数据总线为读取数据提供了基础

20. 什么是虚拟设备?为什么在操作系统中引入虚拟设备?

虚拟设备是通过虚拟技术将一台独占设备变换为若干台逻辑设备,供若干个用户进程同时使用,通常把这种经过虚拟技术处理后的设备称为虚拟设备。
在操作系统设备管理中,引入虚拟设备是为了克服独占设备速度较慢、降低设备资源利用率的缺点,从而提高设备的利用率。

21. 程序员规范中要求不要写出类似(++)+(i++)或 f(++i;i++这样的代码,请说明原因。

计算子表达式的顺序由编译器决定的,虽然参数的压栈顺序在给定的调用方式下式固定的但参数表达式的计算顺序也由编译器决定的。不同的编译器或不同的表达式计算的顺序可能不一致

22. 操作系统中进程调度策略有哪几种?

FCFS(先来先服务)，优先级，时间片轮转，多级反馈-调度算法。

先来先服务调度算法：是一种最简单的调度算法，每次调度是从进程队列中选择一个最先进入该队列的进程，为之分配资源投入运行。该进程一直运行完成或发生某事件而阻塞后才继续处理后面的进程。

优先级调度算法：有短进程优先级、高优先权优先级、高响应比优先级等，按照优先级来执行就绪进程队列中的调度。 （高响应比：（等待时间+服务运行时间）/服务运行时间）

时间片轮转调度算法：系统还是按照先来先服务调度就绪进程，但每次调度时，CPU 都会为队首进程分配并执行一个时间片（几 ms~百 ms）。执行时间片用完后计时器即产生时钟中断，停止该进程并将它送到队尾，其他依次执行。这样保证系统能在给定的时间内执行所有用户进程的请求。

多级反馈调度算法：前面都有局限性，综合-> 多级反馈调度算法则不必事先知道各进程所需的执行时间，而且还可以满足各类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。

（1）设置多个就绪队列，每个队列优先级依次减小。为各个队列分配的时间片大小不同，优先级队列越高，里面进程规定的执行时间片就越小。

（2）队列中还是按照 FCFS 原则排队等待，如果第一队列队首进程在规定的时间片内未执行完，则直接调送至第二队尾，依次向后放一个队列的队尾。因此一个长作业进程

会分配到 n 个队列的时间片执行。

（3）按照队列先后依次执行，如果新进的待处理进程优先级较高，则新进程将抢占正在运行的进程，被抢占的进程放置正在运行的队尾。

23. Linux 操作系统重要组成部分?

Linux 系统一般有 4 个主要部分：内核、shell、文件系统和应用程序。内核、shell 和文件系统一起形成了基本的操作系统结构，它们使得用户可以运行程序、管理文件并使用系统。

1. Linux 内核是操作系统的核心，具有很多最基本功能，如虚拟内存、多任务、 共享库、需求加载、可执行程序和 TCP/IP 网络功能。Linux 内核的模块分为以下几个部分：存储管理、CPU 和进程管理、文件系统、设备管理和驱动、网络通信、系统的初始化和系统调用等。

2. Linux shell 是系统的用户界面，提供了用户与内核进行交互操作的一种接 口。它接收用户输入的命令并把它送入内核去执行，是一个命令解释器。另外，shell 编程语言具有普通编程语言的很多特点，用这种编程语言编写的 shell 程序与其他应用程序具有同样的效果。

3. Linux 文件系统文件系统是文件存放在磁盘等存储设备上的组织方法。Linux 系统能支持多种目前流行的文件系统，如 EXT2、EXT3、FAT、FAT32、VFAT 和 ISO9660。

4. Linux 应用程序标准的 Linux 系统一般都有一套都有称为应用程序的程序集，它包括文本编辑器、编程语言、XWindow、办公套件、Internet 工具和数据库等。

24. 简述中断装置的主要职能.

中断装置的职能主要有三点:

1)检查是否有中断事件发生

2)若有中断发生,保护好被中断进程的断点及现场信息,以便进程在适当时候能恢复驼行

3)启动操作系统的中断处理程序

25. 什么是 RAII 资源管理？

即资源获取就是初始化，利用对象生命周期来控制程序资源，简单来说就是通过局部对象来处理一些资源问题

26. 如果在构造函数中调用 memset(this, 0, sizeof(*this))来初始化内存空间，有什么问题吗？

对于有虚函数和虚表存在的类，在进行 memset 后不能调用虚函数和虚基表继承而来的数据和函数

27. struct{char a[0];}的作用？有什么好处？

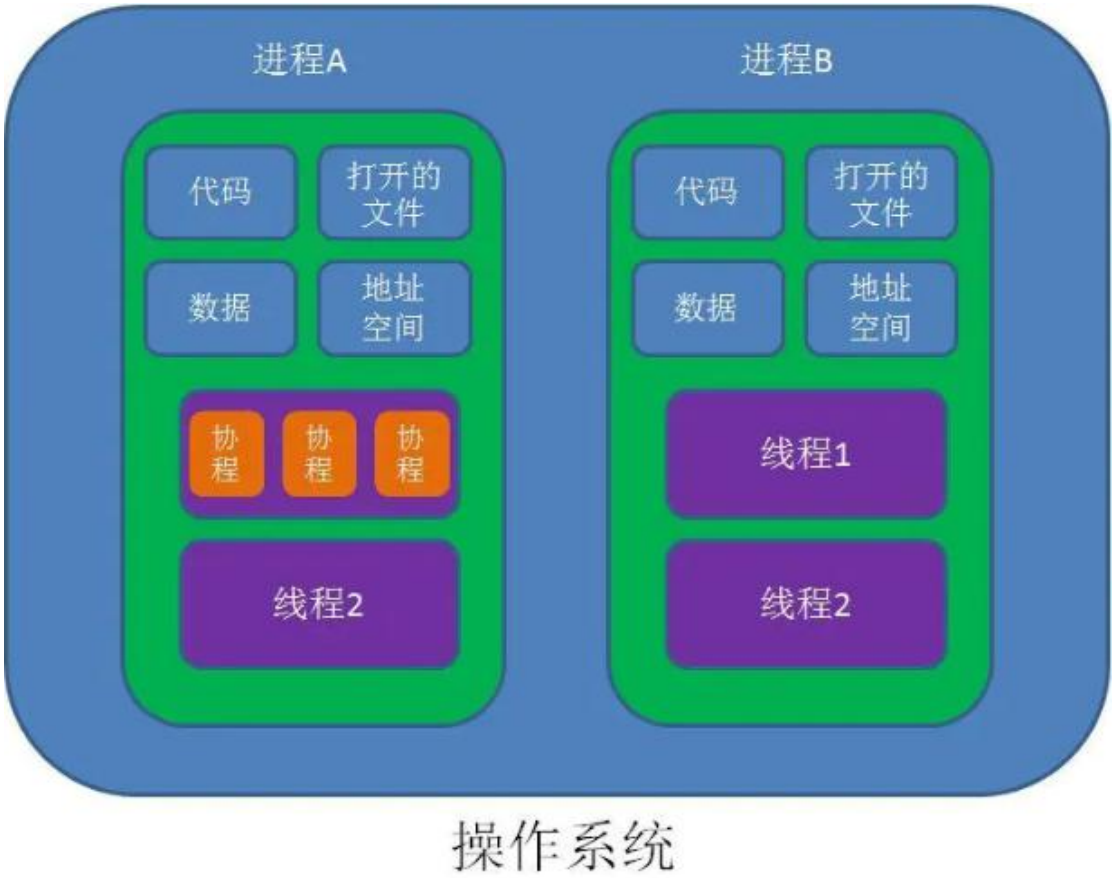
充当可变缓冲区的作用，同时 char a[0]不占用内存空间。

28. TCP 的 nagle 算法和延迟 ack，还有 CORK 呢?他们有什么好处？一起用会有什么效果？你觉得可以有什么改进？

nagle 算法：防止网络中存在太多小包而造成网络拥塞
延迟 ack：减少 ACK 包的频繁发送
CORK：将多个包变成一个包发送，提高网络利用率，使载荷率更大
不可一起使用

29. 什么是协程?

协程是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。



最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。

这样的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。
其中 yield 是 python 当中的语法。当协程执行到 yield 关键字时，会暂停在那一行，等到主线程调用 send 方法发送了数据，协程才会接到数据继续执行。
但是，yield 让协程暂停，和线程的阻塞是有本质区别的。协程的暂停完全由程序控制，线程的阻塞状态是由操作系统内核来进行切换。
因此，协程的开销远远小于线程的开销。

30. 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：

```
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_String;                  //私有成员，保存字符串
};
```

```
#include <iostream>
using namespace std;
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_String;    //私有成员，保存字符串
};

String::~String(void)
{
cout << "Destructing"<< endl;
if (m_String != NULL)           //如果 m_String 不为 NULL，释放堆内存
    {
    delete [] m_String;
    m_String = NULL;            //释放后置为 NULL
    }
    }
```

```
String::String(const char *str)
{
    cout << "Construcing" << endl;
    if(str == NULL)               //如果 str 为 NULL，存空字符串""
    {
        m_String = new char[1];   //分配一个字节
        *m_String = '\0';         //将之赋值为字符串结束符
    }
    else
    {
        //分配空间容纳 str 内容
        m_String = new char[strlen(str) + 1];
        //拷贝 str 到私有成员
        strcpy(m_String, str);
    }
}

String::String(const String &other)
{
    cout << "Constructing Copy" << endl;
    //分配空间容纳 str 内容
    m_String = new char[strlen(other.m_String) + 1];
    //拷贝 str 到私有成员
    strcpy(m_String, other.m_String);
}

String & String:perator = (const String &other)
{
    cout << "Operate = Function" << endl;
    if(this == &other)                //如果对象与 other 是同一个对象
    {                                 //直接返回本身
        return *this;
    }
    delete [] m_String;               //释放堆内存
    m_String = new char[strlen(other.m_String)+1];
    strcpy(m_String, other.m_String);
    return *this;
}

int main()
{
    String a("hello");            //调用普通构造函数
    String b("world");            //调用普通构造函数
```

```
    String c(a);                    //调用拷贝构造函数
    c = b;                          //调用赋值函数

    return 0;
}
```

31. 确保线程安全的几种方法?

确保线程安全的方法有这几个：竞争与原子操作、同步与锁、可重入、过度优化。

1.竞争与原子操作

多个线程同时访问和修改一个数据，可能造成很严重的后果。出现严重后果的原因是很多操作被操作系统编译为汇编代码之后不止一条指令，因此在执行的时候可能执行了一半就被调度系统打断了而去执行别的代码了。一般将单指令的操作称为原子的(Atomic)，因为不管怎样，单条指令的执行是不会被打断的。

因此，为了避免出现多线程操作数据的出现异常，Linux 系统提供了一些常用操作的原子指令，确保了线程的安全。但是，它们只适用于比较简单的场合，在复杂的情况下就要选用其他的方法了。

2.同步与锁

为了避免多个线程同时读写一个数据而产生不可预料的后果，开发人员要将各个线程对同一个数据的访问同步，也就是说，在一个线程访问数据未结束的时候，其他线程不得对同一个数据进行访问。

同步的最常用的方法是使用锁(Lock)，它是一种非强制机制，每个线程在访问数据或资源之前首先试图获取锁，并在访问结束之后释放锁；在锁已经被占用的时候试图获取锁时，线程会等待，直到锁重新可用。

二元信号量是最简单的一种锁，它只有两种状态：占用与非占用，它适合只能被唯一一个线程独占访问的资源。对于允许多个线程并发访问的资源，要使用多元信号量(简称信号量)。

3.可重入

一个函数被重入，表示这个函数没有执行完成，但由于外部因素或内部因素，又一次进入该函数执行。一个函数称为可重入的，表明该函数被重入之后不会产生任何不良后果。可重入是并发安全的强力保障，一个可重入的函数可以在多线程环境下放心使用。

4.过度优化

在很多情况下，即使我们合理地使用了锁，也不一定能够保证线程安全，因此，我们可能对代码进行过度的优化以确保线程安全。

我们可以使用 volatile 关键字试图阻止过度优化，它可以做两件事：第一，阻止编译器为了提高速度将一个变量缓存到寄存器而不写回；第二，阻止编译器调整操作 volatile 变量的指令顺序。

在另一种情况下，CPU 的乱序执行让多线程安全保障的努力变得很困难，通常的解决办法是调用 CPU 提供的一条常被称作 barrier 的指令，它会阻止 CPU 将该指令之前的指令交换到 barrier 之后，反之亦然。

32. 请求页面置换策略有哪些方式？他们的区别是什么？各自有什么算法解决？

全局和局部
全局：在整个内存空间置换
局部：在本进程中进行置换
全局：　（1）工作集算法
　　（2）缺页率置换算法
局部：　（1）最优算法
　　（2）FIFO 先进先出算法
　　（3）LRU 最近最久未使用
　　（4）时钟算法

33. 系统调用与函数调用的区别？

（1）一个在用户地址空间执行；一个在内核空间执行
（2）一个是过程调用，开销小；一个需要切换用户空间和内核上下文，开销大
（3）一般相同；不同系统不同

34. 当接受方的返回的接受窗口为 0 时，发送方会进行什么操作？

开启计时器，发送零窗口探测报文

35. 虚函数表是在什么时候确定的？那虚表指针呢？

编译时确定虚函数表，虚表指针则是运行时

36. 创建进程的步骤？

（1）申请空的 PCB
（2）为新进程分配资源
（3）初始化 PCB
（4）将新进程插入就绪队列中

37. 进程切换发生的原因？处理进程切换的步骤？

原因：中断发生；更高优先级进程唤醒；进程消耗完了时间片；资源阻塞；

步骤：

（1）保存处理器的上下文
（2）用新状态和其它相关信息更新正在运行进程 的 PCB
（3）将原来的进程移到合适的队列中【就绪，阻塞】
（4）选择另外一个执行的 进程，更新被选中进程的 PCB，将它加载进 CPU

38. DNS 协议如何实现将域名解析为 IP 地址的？

（1）客户机的应用程序调用解析程序将域名已 UDP 数据报的形式发给本地 DNS 服务器
（2）本地 DNS 服务器找到对应 IP 以 UDP 形式放松回来
（3）弱本地 DNS 服务器找不到，则需要将域名发送到根域名服务器，根域名服务器返回下一个要访问的域名服务器，则访问下一个域名服务器。

39. 停止等待协议的缺点？为什么？

信道利用率太低，每次都需要等上一次 ACK 包接收到了才能再次发送

40. 拥塞控制的方式？具体怎么做的？快重传的时机是什么？

（1）慢开始
（2）拥塞避免
（3）快重传【收到 3 个失序分组确认】
（4）快恢复

41. 浮点数为什么会有误差？

因为二进制无法精准的表示十进制小数，0.3 和 0.2 都无法完整的用二进制表示。

42. 将”引用”作为函数返回值类型的格式、好处和需要遵守的规则

格式：类型标识符 &函数名(形参列表及类型说明){//函数体}
格式：在内存中不产生被返回值的副本：(注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效。)
注意事项：
不能返回局部变量的引用。主要原因是局部变量会在函数返回时被销毁，因此被返回的引用就成为了”无所指的”引用，程序会进入未知状态。

不能返回函数内部 new 分配的内存的引用。虽然不存在局部变量的被动销毁问题，可对于这种情况(返回函数内部 new 分配内存的引用)，又面临其它尴尬的局面。如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间(由 new 分配)就无法释放。

可以返回类成员的引用，但最好是 const。主要原因是当对象的属性是与某种业务规则相关联时，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常量引用(或指针)，那么对该属性的单纯赋值就会破坏业务规则的完整性。

43. 给定三角形 ABC 和一点 P(x,y,z)，判断点 P 是否在 ABC 内，给出思路并手写代码.

```
#include <iostream>
#include <math.h>
using namespace std;
#define ABS_FLOAT_0 0.0001
struct point_float
{
    float x;
    float y;
};

/**
* @brief 计算三角形面积
*/
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;
}

/**
* @brief 判断给定一点是否在三角形内或边上
*/
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 SumSuqar = SADB + SBDC + SADC;
    if ((-ABS_FLOAT_0 < (SABC - SumSuqar)) &&
    ((SABC - SumSuqar) < ABS_FLOAT_0))
    {
        return true;
    }else{
        return false;
    }
}
```

44．linux 信号有哪些?

常用信号

信号	值	动作解释
SIGHUP	1	终端线路挂断
SIGINT	2	Term 键盘输入的中断命令，从终端输入 Ctrl-C 时发生
SIGQUIT	3	Core 键盘输入的退出命令
SIGILL	4	Core 错误指令
SIGABRT	6	Coreabort(3)发出的中止信号
SIGFPE	8	Core 浮点数异常
SIGKILL	9	TermKILL 信号
SIGSEGV	11	Core 非法内存访问
SIGPIPE	13	Term 管道断开
SIGALRM	14	Termalarm(2)发出的中止信号
SIGTERM	15	Term 强制中止信号
SIGUSR	130,10,16	Term 用户自定义信号 1
SIGUSR	231,12,17	Term 用户自定义信号 2
SIGCHLD	20,17,18	Ign 子进程中止信号
SIGCONT	19,18,25	Cont 继续执行一个停止的进程
SIGSTOP	17,19,23	Stop 非终端来的停止信号
SIGTSTP	18,20,24	Stop 终端来的停止信号
SIGTTIN	21,21,26	Stop 后台进程读终端
SIGTTOU	22,22,27	Stop 后台进程写终端

45. 缓冲溢出报警及解决?

缓冲区溢出的目的在于扰乱具有某些特权运行程序的功能，这样就可以让攻击者取得程序的控制权，从而进行缓冲区溢出攻击行为。假如该程序具有足够的权限，那么整个主机甚至服务器就被控制了。

1.在被攻击程序地址空间里安排攻击代码的方法
1).植人法
攻击者向被攻击的程序输人一个字符串，程序会把这个字符串放到缓冲区里。这个字符串所包含的数据是可以在这个被攻击的硬件平台运行的指令流。在这里攻击者用被攻击
程序的缓冲区来存放攻击代码，具体方式有以下两方面差别:
.攻击者不必为达到此目的而溢出任何缓冲区，;叮以找到足够的空间来放置攻击代码;
.缓冲区可设在任何地方:堆栈(存放白动变量)、堆(动态分配区)和静态数据区初始化或未初始化的资料.
2).利用已经存在的代码
很多时候攻击者所要的代码已经存在于被攻击的程序中了，攻击者所要做的只是对代码传递一些参数，然后使程序跳转到想要执行的代码那里。比方说，攻击代码要求执行“ex-ec("/bin/sh") "，而在 libc 库中的代码执行“exec(arg) "，其中 arg 是一个指向字符串的指针参数，那么攻击者只要把传人的参数指针改为指向“/bin/sh"，然后跳转到 libc 库中相应的指令序列即可。

2.控制程序转移到攻击代码的方法
上面讲到的方法都是在试图改变程序的执行流程，使之跳转到攻击代码。其基本特点就是给没有边界检查或有其他弱点的程序送出一个超长的缓冲区，以达到扰乱程序正常执行顺序的目的。通过溢出一个缓冲区，攻击者可以用近乎暴力的方法(穷尽法)改写相邻的程序空问而直接跳过系统的检查。
这里的分类基准是攻击者所寻求的缓冲区溢出的程序空问类型。原则上可以是任意的空问。比如起初的 Moms Worm(穆尔斯蠕虫)就是使用了 fingerd 程序的缓冲区溢出，扰乱 fingerd 要执行的文件的名字。其实许多缓冲的区溢出是用暴力的方法来寻求改变程序指针的。这类程序不同的地方就是程序空间的突破和内存空间的定位不同。通常情况下，控制程序转移到攻击代码的方法有下面几种:
1).函数返回地址
在一个函数调用发生时，调用者会在堆栈中留下函数返回地址，它包含了函数结束时返回的地址。攻击者通过溢出这些自动变量，使这个返回地址指向攻击代码，这样当函数调用结束时，程序跳转到攻击者设定的地址，而不是原先的地址。这种缓冲区溢出被称为“stacksmashing attack" ,是目前常用的缓冲区溢出攻击方式。
2).函数指针
"void(* foo) ()”中声明了一个返回值为 void 函数指针的变量 foo。函数指针定位任何地址空间，所以攻击者只要在任何空间内的函数指针附近找到一个能够溢出的缓冲区，然后溢出来改变 A 数指针，当程序通过函数指针调用函数时，程序的流程就会发生改变而实现攻击者的目的。
3).长跳转缓冲区

在 C 语言中包含了一个简单的检验/恢复系统，称为“setjmp/longjmp",意思是在检验点设定“setjmp (buffer) "，用“longjmp(buffer)”来恢复检验点。可是，假如攻击时能够进人缓冲区的空间，那么“longjmp(buffer)”实际_I:是跳转到攻击者的代码。像函数指针一样，longjmp 缓冲区能够指向任何地方，所以攻击者所要做的就是找到一个可供溢出的缓冲区。一个典型的例子就是 Perl 5.003，攻击者首先进人用来恢复缓冲区溢出的 longjmp 缓冲区，然后诱导进人恢复模式，这样就使 Perl 的解释器跳转到攻击代码 L 了。

3.综合代码植人和流程控制技术
最常见和最简单的缓冲区溢出攻击类型就是在一个字符串里综合了代码植人和启动记录。攻击者定位一个可供溢出的自动变 Q，接着向程序传递一个很大的字符串，在引发缓冲区溢出改变启动记录的同时植人了代码(C 语言程序员习惯上只为用户和参数开辟很小的缓冲区)。
代码植人和缓冲区溢出不一定要在一次动作内完成，攻击者可以在一个缓冲区内放置代码(这时并不能溢出缓冲区)，接着攻击者通过溢出另一个缓冲区来转移程序的指针。这样的方法通常用来解决可供溢出的缓冲区不够大(不能放下全部的代码)。如果攻击者试图使用已经常驻的代码而不是从外部植入代码，他们通常必须把代码作为参数。举例说明，在 libc(几乎所有的 C 程序都用它来连接)中的一部分代码段会执行“exec(something) "，其中的 something 就是参数，攻击者使用缓冲区溢出改变程序的参数，然后利用另一个缓冲区溢出攻击，使程序指针指向 lihc 中的特定的代码段。

46. 如何判断两个结构体是否相等?

判断两个结构体是否相等：重载操作符"=="
不能用函数 memcpy 来判断两个结构体是否相等：memcmp 函数是逐个字节进行比较的，而 struct 存在字节对齐，字节对齐时补的字节内容是随机的，会产生垃圾值，所以无法比较。

```
#include<iostream>
using namespace std;

struct s
{
    int a;
    int b;
    bool operator == (const s &rhs);
};

bool s::operator == (const s &rhs)
{
    return ((a == rhs.a) && (b == rhs.b));
}
```

```
int main()
{
    struct s s1, s2;
    s1.a = 1;
    s1.b = 2;
    s2.a = 1;
    s2.b = 2;
    if (s1 == s2)
        cout << "两个结构体相等" << endl;
    else
        cout << "两个结构体不相等" << endl;
    return 0;
}
```

47. 云盘中秒传功能是什么原理，说说其中一个算法.

秒传原理:
上传大文件时,会对文件进行比对操作,这里的对比操作其实就是将我们下载的插件对要上传的文件进行"哈希值"的计算,跟百度的"哈希值"数据库中的文件进行匹配操作.如果发现两者的"哈希值"相同,那么,将已存在于百度数据库里面的文件对应的文件链接到我们对应的帐号里,做一个关联就可以,其实并没有对本地文件进行上传,所以我们也就看到了秒传的效果.
1、因为在上传文件之前,如果你是第一次使用百度网盘,那么会提示你安装一个 极速控件.
2、该极速控件的下载是为了更方便的来检测我们将要上传文件的 哈希值.也就是文件的唯一识别码.

哈希算法

48. 栈上分配内存和堆上分配内存有什么区别？

栈上：分配简单，只需要移动栈顶指针，不需要其他的处理。
堆上：分配复杂，需要进行一定程度清理工作，同时是调用函数处理的。

49. 如何将一棵树转化成二叉树?

将节点的孩子 放在左子树;
将节点的兄弟 放在右子树。

50. 请解释分段与分页机制.

1.分段机制

1)什么是分段机制

分段机制就是把虚拟地址空间中的虚拟内存组织成一些长度可变的称为段的内存块单元。

2)什么是段

每个段由三个参数定义：段基地址、段限长和段属性。

段的基地址、段限长以及段的保护属性存储在一个称为段描述符的结构项中。

3)段的作用

段可以用来存放程序的代码、数据和堆栈，或者用来存放系统数据结构。

4)段的存储地址

系统中所有使用的段都包含在处理器线性地址空间中。

5)段选择符

逻辑地址包括一个段选择符或一个偏移量，段选择符是一个段的唯一标识，提供了段描述符表，段描述符表指明短的大小、访问权限和段的特权级、段 类型以 及段的第一个字节在线性地址空间中的位置（称为段的基地址）。逻辑地址的 偏移量部分到段的基地址上就可以定位段中某个字节的位置。因此基地址加上 偏移量就形成了处理器线性地址空间中的地址。

6)逻辑地址到线性地址的变换过程

如果没有开启分页，那么处理器直接把线性地址映射到物理地址，即线性地址被送到处理器地址总线上；如果对线性地址空间进行了分页处理，那么就会使用二级地址转换把线性地址转换成物理地址。

7)虚拟地址到物理地址的变换过程

2.分页机制

1)什么是分页机制

分页机制在段机制之后进行的，它进一步将线性地址转换为物理地址。

2)分页机制的存储

分页机制支持虚拟存储技术，在使用虚拟存储的环境中，大容量的线性地址空间需要使用小块的物理内存（RAM 或 ROM）以及某些外部存储空间来 模拟。当使用分页时，每个段被划分成页面（通常每页为 4K 大小），页面会被存储于物理内存中或硬盘中。操作系统通过维护一个页目录和一些页表来留意这些页 面。当程序（或任务）试图访问线性地址空间中的一个地址位置时，处理器就会使用页目录和页表把线性地址转换成一个物理地址，然后在该内存位置上执行所要求 的操作。

3)线性地址和物理地址之间的变换过程

3．分段机制和分页机制的区别

1)分页机制会使用大小固定的内存块，而分段管理则使用了大小可变的块来管理内存。

2)分页使用固定大小的块更为适合管理物理内存，分段机制使用大小可变的块更适合处理复杂系统的逻辑分区。

3)段表存储在线性地址空间，而页表则保存在物理地址空间。