计算机基础知识

算法 实习

计算机网络

1. **A类地址**：8位网络号，24位主机号，网络号以0开头； **B类地址**：16位网络号，16位主机号，网络号以10开头；**C类地址**：24位网络号，8位主机号，网络号以110开头；**D类地址**：1110开头，做多播地址；**E类地址**：1111，保留今后使用；

2. HTTP协议状态码表示的意思主要分为五类 ,大体是：
   1××　　保留
   2××　　表示请求成功地接收
   3××　　为完成请求客户需进一步细化请求
   4××　　客户错误
   5××　　服务器错误

   - 1xx（临时响应）：表示临时响应并需要请求者继续执行操作的状态代码。
   - 100（继续） 请求者应当继续提出请求。 服务器返回此代码表示已收到请求的第一部分，正在等待其余部分。
   - 101（切换协议） 请求者已要求服务器切换协议，服务器已确认并准备切换。
   - 200（成功） 服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。 如果针对您的 robots.txt 文件显示此状态，则表示 Googlebot 已成功检索到该文件。
   - 201（已创建） 请求成功并且服务器创建了新的资源。
   - 202（已接受） 服务器已接受请求，但尚未处理。
   - 203（非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。
   - 204（无内容） 服务器成功处理了请求，但没有返回任何内容。
   - 205（重置内容） 服务器成功处理了请求，但没有返回任何内容。 与 204 响应不同，此响应要求请求者重置文档视图（例如，清除表单内容以输入新内容）。
   - 206（部分内容） 服务器成功处理了部分 GET 请求。
   - 300（多种选择） 针对请求，服务器可执行多种操作。 服务器可根据请求者（用户代理）选择一项操作，或提供操作列表供请求者选择。
   - 301（永久移动） 请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。 您应使用此代码告诉 Googlebot 某个网页或网站已永久移动到新位置。
   - 302（暂时移动） 服 务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。 此代码与响应 GET 或 HEAD 请求的 301 代码

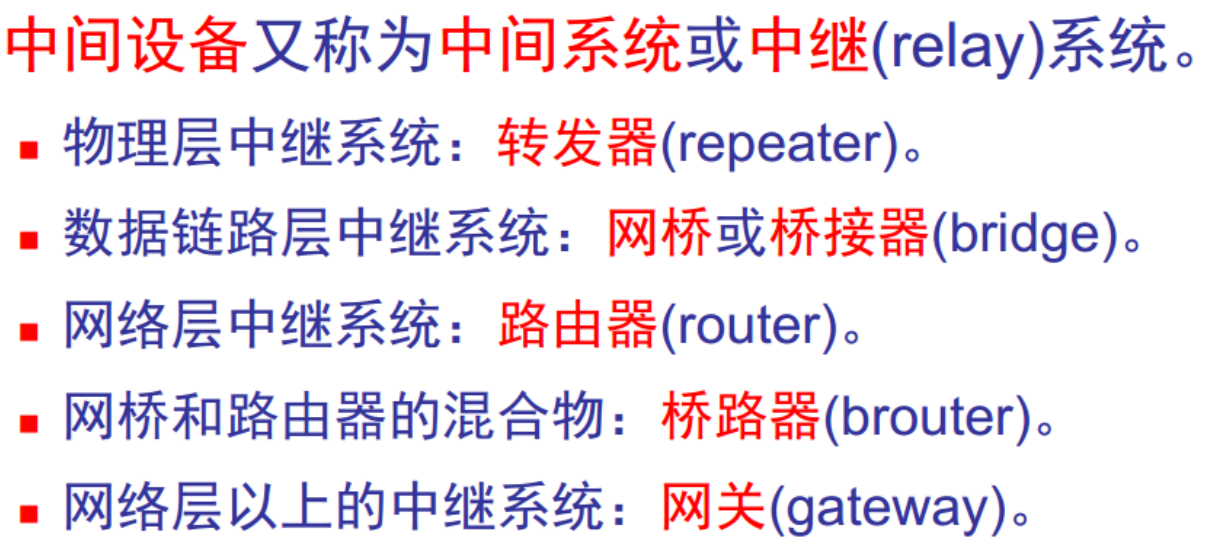
类似，会自动将请求者转到不同的位置，但您不应使用此代码来告诉 Googlebot 某个网页或网站已经移动，因为 Googlebot 会继续抓取原有位置并编入索引。

- 303（查看其他位置） 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。 对于除 HEAD 之外的所有请求，服务器会自动转到其他位置。
- 304（未修改） 自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。如果网页自请求者上次请求后再也没有更改过，您应当将服务器配置为返回此响应（称为 If-Modified-Since HTTP 标头）。 由于服务器可以告诉 Googlebot 自从上次抓取后网页没有更改过，因此可节省带宽和开销 。
- 305（使用代理） 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理。
- 307（暂时重定向） 服 务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。 此代码与响应 GET 和 HEAD 请求的 301 代码类似，会自动将请求者转到不同的位置，但您不应使用此代码来告诉 Googlebot 某个页面或网站已经移动，因为 Googlebot 会继续抓取原有位置并编入索引。400（错误请求） 服务器不理解请求的语法。
- 401（未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。
- 403（禁止） 服务器拒绝请求。 如果您看到 Googlebot 在尝试抓取您网站上的有效网页时收到此状态代码（可以在 Google 网站管理员工具诊 断下的网络抓取页面上看到此信息），可能是您的服务器或主机拒绝 Googlebot 访问。
- 404（未找到） 服务器找不到请求的网页。 例如，如果请求服务器上不存在的网页，服务器通常会返回此代码。如果您的网站上没有 robots.txt 文件，而您在 Google 网站管理员工具”诊断”标签的 robots.txt 页上看到此状态，那么这是正确的状态。 但是，如果您有 robots.txt 文件而又看到此状态，则说明您的 robots.txt 文件可能命名错误或位于错误的位置 （该文件应当位于顶级域名，名为 robots.txt）。 如果您看到有关 Googlebot 尝试抓取的网址的此状态（在”诊断”标签的 HTTP 错误页上），则表示 Googlebot 追踪的可能是另一个页面的无效链接（是旧链接或输入有误的链接）。
- 405（禁用的方法） 禁用请求中指定的方法。
- 406（不可接受） 无法使用请求的内容特性响应请求的网页。
- 407（需要代理授权） 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理。 如果服务器返回此响应，还会指明请求者应当使用的代理。
- 408（请求超时） 服务器等候请求时发生超时。
- 409（冲突） 服务器在完成请求时发生冲突。 服务器必须在响应中包含有关冲突的信息。 服务器在响应与前一个请求相冲突的 PUT 请求时可能会返回此代码，同时会附上两个请求的差异列表。
- 410（已删除） 如果请求的资源已永久删除，服务器就会返回此响应。 该代码与 404（未找到）代码相似，但在资源以前存在而现在不存在的情况下，有时会用来替代 404 代码。 如果资源已永久删除，您应当使用 301 指定资源的新位置。
- 411（需要有效长度） 服务器不接受不含有效内容长度标头字段的请求。

- 412（未满足前提条件） 服务器未满足请求者在请求中设置的其中一个前提条件。
- 413（请求实体过大） 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。
- 414（请求的 URI 过长） 请求的 URI（通常为网址）过长，服务器无法处理。
- 415（不支持的媒体类型） 请求的格式不受请求页面的支持。
- 416（请求范围不符合要求） 如果页面无法提供请求的范围，则服务器会返回此状态代码。
- 417（未满足期望要求） 服务器未满足”期望”请求标头字段的要求。
  500（服务器内部错误） 服务器遇到错误，无法完成请求。
- 501（尚未实施） 服务器不具备完成请求的功能。 例如，服务器无法识别请求方法时可能会返回此代码。
- 502（错误网关） 服务器充当网关或代理，从上游服务器收到无效响应。
- 503（服务不可用） 服务器目前无法使用（由于超载或停机维护）。 通常，这只是暂时状态。
- 504（网关超时） 服务器充当网关或代理，但没有及时从上游服务器收到请求。
- 505（HTTP 版本不受支持） 服务器不支持请求中所用的 HTTP 协议版本。

3. 以太网交换机通常都有十几个接口。因此，以太网交换机实质上就是一个多接口的网桥，可见交换机工作在数据链路层。

4. 中间设备



5. HTTP 1.0 协议是无状态的(stateless)。HTTP/1.1 协议使用持续连接。

6. **慢开始**：一开始的时候设置发送方窗口大小为1（使用报文段的个数作为窗口大小的单位，实际中是字节数），然后每经过一个传输轮次，拥塞窗口加倍。（传输轮次是指将拥塞窗口内所允许发送的报文段都连续发送出去，并收到对发送的最后一个字节的确认）；
   慢开始门限：ssthresh；
   到达慢开始门限之后，改用拥塞避免算法，让拥塞窗口按线性规律缓慢增长。
   无论在慢开始还是拥塞避免阶段，只要发送方判断网络拥塞，都要把慢开始门限ssthresh设置为出现拥塞时的发送方窗口的一半。（这个也被称为**乘法减小**）
   加法增大是指：执行拥塞避免算法后，是拥塞窗口缓慢增大。
   两种算法合起来称为AIMD（加法增大乘法减小）
   快重传算法：要求接收方每收到一个失序的报文段之后就立即发出重复确认，而不要等待自己发送数据时捎带确认。发送方一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传服务器到期。
   快恢复算法：1）当发送方连续收到三个重复确认的时候，就执行乘法减小算法，把慢开始门限减半；2）由于发送方认为网络很可能没有发生拥塞，因此此时不执行慢开始算法，而是把拥塞窗口值设置为慢开始门限ssthresh减半后的数值，然后执行拥塞避免算法。
   在采用块恢复算法时，慢开始算法只在TCP建立连接和网络超时的时候才使用。
   网络层对TCP拥塞控制的影响最大的就是分组丢弃策略。因为路由器采用FIFO，队列满的时候将采用尾部丢弃策略。
   全局同步现象：
   随机早期检测TED：

7. **UDP：**无连接，尽最大努力交付，面向报文，无拥塞控制，支持一对一、一对多、多对一和多对多的交互通信；首部开销小（只有是个字段：源端口，目的端口、长度、检验和）；
   TCP: 面向连接，每一条TCP连接只能是点对点的，提供可靠交付的服务，提供全双工通信，面向字节流。

8. **traceroute**：使用UDP发送错误端口，从而使接收方返回ICMP端口不可达差错报文；

9. UDP首部四个字段，每个 2 字节，加入 1 2 字节的伪首部，用于计算校验和；
   TCP首部前20字节固定，后面4n字节根据需要添加，最大不能超过40字节；

10. TCP连接释放的过程：
    假设客户端发起释放请求，客户端经历的状态为：FIN_WAIT_1, FIN_WAIT_2, TIME_WAIT ； 服务端经历的状态为： CLOSE_WAIT, LASK_ACK；
    为什么会有TIME_WAIT状态，并且要持续2MSL(最长报文寿命) ？ 因为1）为了保证A发出的最后一个确认信号ACK能够正确到达B； 2）为了防止已失效的连接请求报文段出现在本连接中。

11.

数据结构

1. 二叉树遍历的非递归解法：

    - 前序遍历：1)访问结点P，并将结点P入栈; 2)判断结点P的左孩子是否为空，若为空，则取栈顶结点并进行出栈操作，并将栈顶结点的右孩子置为当前的结点P，循环至1);若不为空，则将P的左孩子置为当前的结点P; 3)直到P为NULL并且栈为空，则遍历结束

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> results;
        stack<TreeNode*> s;
        TreeNode* p = root;
        while (p != NULL || !s.empty()) {
            while (p != NULL) {
                results.push_back(p->val);
                s.push(p);
                p = p->left;
            }

            if (!s.empty()) {
                p = s.top();
                s.pop();
                p = p->right;
            }
        }
        return results;
    }
};
```

    - 中序遍历： 1)若其左孩子不为空，则将P入栈并将P的左孩子置为当前的P，然后对当前结点P再进行相同的处理； 2)若其左孩子为空，则取栈顶元素并进行出栈操作，访问该栈顶结点，然后将当前的P置为栈顶结点的右孩子； 3)直到P为NULL并且栈为空则遍历结束

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> results;
        stack<TreeNode*> s;
        TreeNode* p = root;
        while (p != NULL || !s.empty()) {

            while (p != NULL) {
                s.push(p);
                p = p->left;
            }

            if (!s.empty()) {
                p = s.top();
                s.pop();
                results.push_back(p->val);
                p = p->right;
            }
        }
        return results;
    }
};
```

- 层次遍历：非递归法——使用两个栈，一个栈保存当前层节点，一个栈保留下一层的节点，使用BFS遍历即可。di
- 后序遍历：要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点P，先将其入栈。如果P不存在左孩子和右孩子，则可以直接访问它；或者P存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将P的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> results;
        stack<TreeNode*> s;
        TreeNode* p;
        TreeNode* pre = NULL;
        if (NULL != root) {
            s.push(root);
        }
        while (!s.empty()) {
            p = s.top();
            if ( (p->left == NULL && p->right == NULL) ||
            (pre != NULL && (pre == p->left || pre == p->right) ) ) {
                results.push_back(p->val);
                pre = p;
                s.pop();
            } else {
                if (NULL != p->right) {
                    s.push(p->right);
                }
                if (NULL != p->left) {
                    s.push(p->left);
                }
            }
        }
        return results;
    }
};
```

2. 关于五大常用算法：

   贪心算法：在对问题求解时，总是做出在当前看来是最好的选择，有可能陷入局部最优。

   分治：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

   动态规划：将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。与分治的区别：经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

   回溯：在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。

   分支限界：采用广度优先的策略，在问题的解空间树T上搜索问题解。与回溯区别：回溯法的求解目标是找出T中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

3. 卡特兰数：

   $$C_n = (1/n + 1)C(2n, n)$$

4. 编程之美 —— 寻找水王：

   ```
   1.
   ```

   扩展：如果有三个水王，每人都超过1/3，找出来，思路跟1个类似。

5. 编程之美 —— **：

   ```
   1.
   ```

   思路：如果两个链表都没有环，那么同原算法；
   如果两个链表一个有环，一个没环，那么必然不相交。如果两个链表都有环，判断一个链表环上的任一点是否在另一个链表上，如果是，则必相交，反之不相交。这时，需要找到另一个链表完整的环都包括了哪些结点，才能进行判断。可以看出，解答这个问题要解决判断是否有环。

6. 编程之美 —— **链表相交**：

   ```
   1.
   ```

思路：
无环的时候： 1）将第二个链表连接到第一个链表的末尾，然后从第二个链表出发判断是否有环；2）hash计数存储链表每个节点的地址；3）判断尾节点是否相同；
有环的时候：
- 如果两个链表都没有环，那么同原算法；
- 如果两个链表一个有环，一个没环，那么必然不相交。
- 如果两个链表都有环，判断一个链表环上的任一点是否在另一个链表上，如果是，则必相交，反之不相交。这时，需要找到另一个链表完整的环都包括了哪些结点，才能进行判断。可以看出，解答这个问题要解决判断是否有环。
如果必须要求出两个链表相交的第一个节点呢？: 用两个指针p1、p2指向表头，每次循环时p1指向它的后继，p2指向它后继的后继。若p2的后继为NULL，表明链表没有环；否则有环且p1==p2时循环可以终止。此时为了寻找环的入口，将p1重新指向表头且仍然每次循环都指向后继，p2每次也指向后继。当p1与p2再次相等时，相等点就是环的入口。

7. 编程之美 —— **二叉树中节点的最大距离**：
   思路：每个节点维护一个左右子树的最大距离，然后递归查找最大距离；

8. 编程之美 —— **：
   思路：

9. **dynamic_cast**: 用于C++继承之间的多态类型转换：分为向上跟向下的转换，向上(子类到父类)肯定转换成功；向下转换将基类类型的指针或引用安全的转换为派生类的指针或引用。dynamic_cast会根据基类指针是否真正指向继承类指针来做相应处理，如果绑定到引用或者指针的对象不是目标类型的对象，转换失败。dynamci_cast将父类cast到子类，父类必须有虚函数。因为dynamic_cast运行时需要检查RTTI信息。

10. 字符串匹配算法：

算法	预处理时间	匹配时间
朴素算法	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
有限自动机算法	$O(m \mid \Sigma \mid)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

图 32-2　本章的字符串匹配算法以及它们的预处理和匹配时间

11. vector 扩容因子为2：空间和时间的权衡。简单来说， 空间分配的多，平摊时间复杂度低，但浪费空间也多。具体参见算法导论中，平摊分析那一章关于动态表扩张的分析。据说很多操作系统会使用伙伴系统（Buddy System）管理内存，于是乎用2^n的数组会不那么容易造成内存碎片。更少的内存碎片 = 更少的内存整理时间。

JAVA ArrayList为1.5， 1.5可以实现空间复用，最佳为1.618；

12. **题目：在一个文件中有 10G 个整数，乱序排列，要求找出中位数。内存限制为 2G。只写出思路即可（内存限制为 2G的意思就是，可以使用2G的空间来运行程序，而不考虑这台机器上的其他软件的占用内存）。**

分析： 既然要找中位数，很简单就是排序的想法。那么基于字节的桶排序是一个可行的方法；

思想：将整形的每1byte作为一个关键字，也就是说一个整形可以拆成4个keys，而且最高位的keys越大，整数越大。如果高位keys相同，则比较次高位的keys。整个比较过程类似于字符串的字典序。

第一步:把10G整数每2G读入一次内存，然后一次遍历这536,870,912个数据。每个数据用位运算”>>”取出最高8位(31-24)。这8bits(0-255)最多表示255个桶，那么可以根据8bit的值来确定丢入第几个桶。最后把每个桶写入一个磁盘文件中，同时在内存中统计每个桶内数据的数量，自然这个数量只需要255个整形空间即可。

代价：(1) 10G数据依次读入内存的IO代价(这个是无法避免的，CPU不能直接在磁盘上运算)。(2)在内存中遍历536,870,912个数据，这是一个O(n)的线性时间复杂度。(3)把255个桶写会到255个磁盘文件空间中，这个代价是额外的，也就是多付出一倍的10G数据转移的时间。

第二步：根据内存中255个桶内的数量，计算中位数在第几个桶中。很显然，2,684,354,560个数中位数是第1,342,177,280个。假设前127个桶的数据量相加，发现少于1,342,177,280，把第128个桶数据量加上，大于1,342,177,280。说明，中位数必在磁盘的第128个桶中。而且在这个桶的第1,342,177,280-N(0-127)个数位上。N(0-127)表示前127个桶的数据量之和。然后把第128个文件中的整数读入内存。(平均而言，每个文件的大小估计在10G/128=80M左右，当然也不一定，但是超过2G的可能性很小)。

代价：(1)循环计算255个桶中的数据量累加，需要O(M)的代价，其中m<255。(2)读入一个大概80M左右文件大小的IO代价。

注意，变态的情况下，这个需要读入的第128号文件仍然大于2G，那么整个读入仍然可以按照第一步分批来进行读取。

第三步：继续以内存中的整数的次高8bit进行桶排序(23-16)。过程和第一步相同，也是255个桶。

第四步：一直下去，直到最低字节(7-0bit)的桶排序结束。我相信这个时候完全可以在内存中使用一次快排就可以了。

整个过程的时间复杂度在O(n)的线性级别上(没有任何循环嵌套)。但主要时间消耗在第一步的第二次内存-磁盘数据交换上，即10G数据分255个文件写回磁盘上。一般而言，如果第二步过后，内存可以容纳下存在中位数的某一个文件的话，直接快排就可以了

13. C++ 指针特点：

（1）汇编和指令级别就有指针概念，所以c，c++里有指针是很自然的事。

（2）配合内存管理，必须需要指针概念。

（3）例如，文件的区块性加载或存储，一整块数据结构。其他语言因为OO了，你可能通常要一个字段一个字段来。

（4）指向代码的指针：函数指针，是插件架构，不同人编写不同模块合作的基础。

（5）c、c++中特有的半开口的可扩展内存。

（6）指针和数组模型的天然无缝。

（7）只有把内存管理权交给程序员，C++程序才能做到内存的使用如此灵活和自由，才能达到高效。C++程序的内存分配策略可以是灵活而动态的，达到真正的按需分配，根据配置文件决定，这不是在编译期的傻傻的静态决定，而是运行期决定。

（8）C的指针是操纵硬件的唯一方式，有了指针你才能和真实的世界打交道。

（9）C 语言只有值的传递，无法直接传递引用，要想传递引用必须通过指针间接实现。

C++的缺点：

（1） C++ 的函数重载决议规则是所有语言中最复杂的，因为他允许用户以两种方式自定义隐式类型转换；

（2）因为兼容c，带来风格上的不一致。

14. java TCP连接的步骤：

TCP连接的建立步骤：

客户端向服务器端发送连接请求后，就被动地等待服务器的响应。典型的TCP客户端要经过下面三步操作：

1、创建一个Socket实例：构造函数向指定的远程主机和端口建立一个TCP连接；

2.通过套接字的I/O流与服务端通信；

3、使用Socket类的close方法关闭连接。

服务端的工作是建立一个通信终端，并被动地等待客户端的连接。典型的TCP服务端执行如下两步操作：

1、创建一个ServerSocket实例并指定本地端口，用来监听客户端在该端口发送的TCP连接请求；

2、重复执行：

1）调用ServerSocket的accept（ ）方法以获取客户端连接，并通过其返回值创建一个Socket实例；

2）为返回的Socket实例开启新的线程，并使用返回的Socket实例的I/O流与客户端通信；

3）通信完成后，使用Socket类的close（ ）方法关闭该客户端的套接字连接。

15. Java通过DatagramPacket类和DatagramSocket类来使用UDP套接字，客户端和服务器端都通过DatagramSocket的send（ ）方法和receive（ ）方法来发送和接收数据，用DatagramPacket来包装需要发送或者接收到的数据。发送信息时，Java创建一个包含待发送信息的DatagramPacket实例，并将其作为参数传递给DatagramSocket实例的send（ ）方法；接收信息时，Java程序首先创建一个DatagramPacket实例，该实例预先分配了一些空间，并将接收到的信息存放在该空间中，然后把该实例作为参数传递给DatagramSocket实例的receive（ ）方法。在创建DatagramPacket实例时，要注意：如果该实例用来包装待接收的数据，则不指定数据来源的远程主机和端口，只需指定一个缓存数据的byte数组即可（在调用receive（ ）方法接收到数据后，源地址和端口等信息会自动包含在DatagramPacket实例中），而如果该实例用来包装待发送的数据，则要指定要发送到的目的主机和端口。
    UDP的通信建立的步骤
    UDP客户端首先向被动等待联系的服务器发送一个数据报文。一个典型的UDP客户端要经过下面三步操作：
    1、创建一个DatagramSocket实例，可以有选择地对本地地址和端口号进行设置，如果设置了端口号，则客户端会在该端口号上监听从服务器端发送来的数据；
    2、使用DatagramSocket实例的send（ ）和receive（ ）方法来发送和接收DatagramPacket实例，进行通信；
    3、通信完成后，调用DatagramSocket实例的close（ ）方法来关闭该套接字。

    由于UDP是无连接的，因此UDP服务端不需要等待客户端的请求以建立连接。另外，UDP服务器为所有通信使用同一套接字，这点与TCP服务器不同，TCP服务器则为每个成功返回的accept()方法创建一个新的套接字。一个典型的UDP服务端要经过下面三步操作：
    1、创建一个DatagramSocket实例，指定本地端口号，并可以有选择地指定本地地址，此时，服务器已经准备好从任何客户端接收数据报文；
    2、使用DatagramSocket实例的receive（ ）方法接收一个DatagramPacket实例，当receive（ ）方法返回时，数据报文就包含了客户端的地址，这样就知道了回复信息应该发送到什么地方；
    3、使用DatagramSocket实例的send（ ）方法向服务器端返回DatagramPacket实例。

16. C++ 类的成员访问权限底层实现：

17. 运算符优先级

Precedence	Operator	Description	Associativity
1	++ -- () [] . –> (*type*){*list*}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + – ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ –	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	= += –= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

一些容易出错的优先级问题

上表中，优先级同为1 的几种运算符如果同时出现，那怎么确定表达式的优先级呢？这是很多初学者迷糊的地方。下表就整理了这些容易出错的情况：

优先级问题	表达式	经常误认为的结果	实际结果
.的优先级高于* ->操作符用于消除这个问题	*p.f	p所指对象的字段f (*p).f	对p取f偏移，作为指针，然后进行解除引用操作。*(p.f)
[]高于*	int *ap[]	ap是个指向int数组的指针 int (*ap)[]	ap是个元素为int指针的数组 int *(ap[])
函数()高于*	int *fp()	fp是个函数指针，所指函数返回int。 int (*fp)()	fp是个函数，返回int * int *(fp())
== 和!=高于位操作	(val & mask != 0)	(val & mask)!= 0	val & (mask != 0)
== 和!=高于赋值符	c = getchar() != EOF	(c = getchar()) != EOF	c = (getchar() != EOF)
算术运算符高于位移运算符	msb << 4 + lsb	(msb << 4) + lsb	msb << (4 + lsb)
逗号运算符在所有运算符中优先级最低	i = 1,2	i = (1,2)	(i = 1),2

18. select、poll 和epoll：

指的是linux/unix/freebsd等系统下，调用select/poll/epoll/kqueue等函数来告知操作系统自己关心的描述符，让操作系统在这些描述符可读可写或异常时通知你。IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：

（1）当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。

（2）当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。

（3）如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。

（4）如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。

（5）如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。

与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

elect，poll，epoll都是IO多路复用的机制。所谓I/O多路复用机制，就是说通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。关于阻塞，非阻塞，同步，异步将在下一篇文章详细说明。

一、select实现

1、使用copy_from_user从用户空间拷贝fd_set到内核空间

2、注册回调函数__pollwait

3、遍历所有fd，调用其对应的poll方法（对于socket，这个poll方法是sock_poll，sock_poll根据情况会调用到tcp_poll,udp_poll或者datagram_poll）

4、以tcp_poll为例，其核心实现就是__pollwait，也就是上面注册的回调函数。

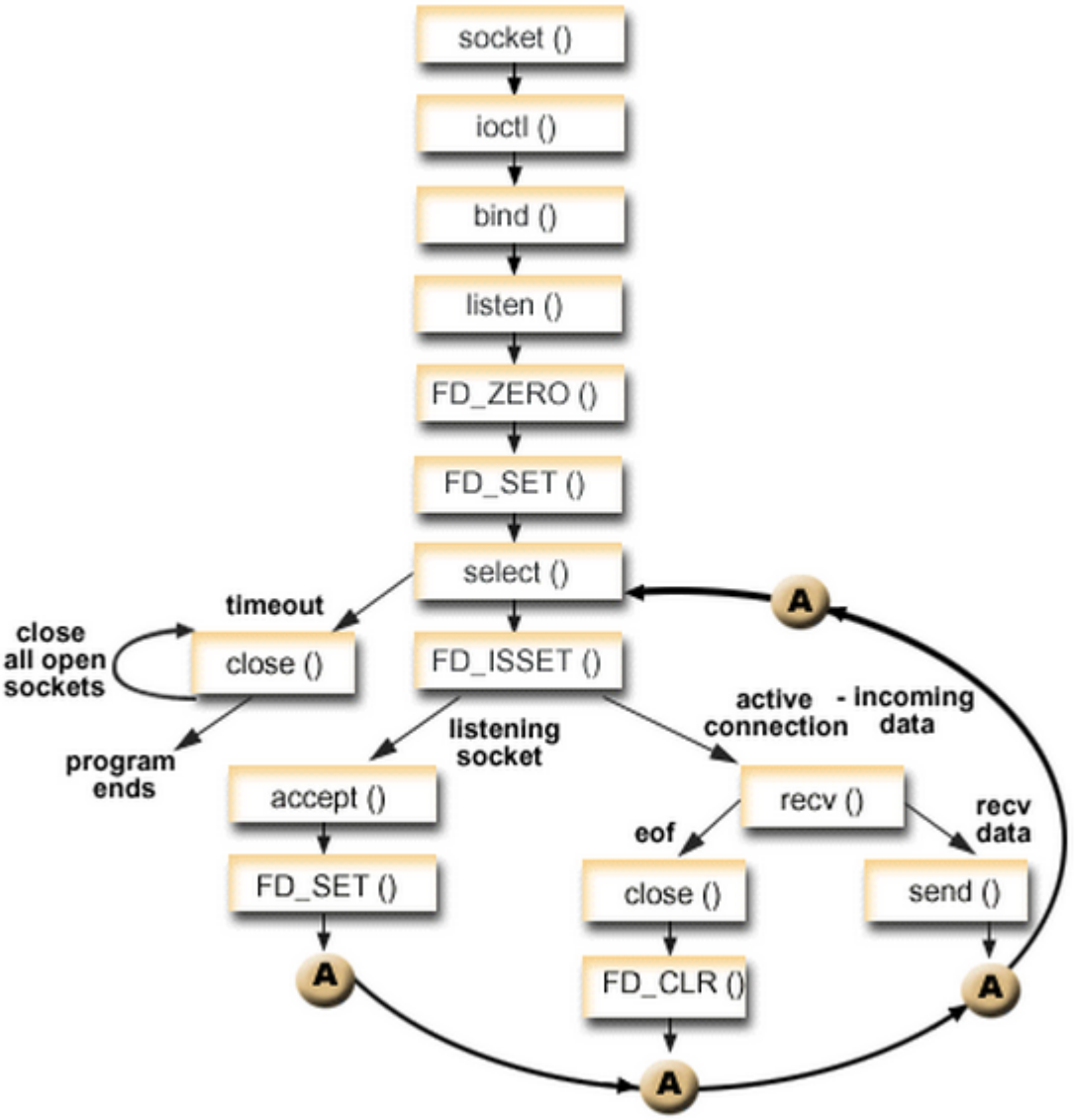
5、__pollwait的主要工作就是把current（当前进程）挂到设备的等待队列中，不同的设备有不同的等待队列，对于tcp_poll来说，其等待队列是sk->sk_sleep（注意把进程挂到等待队列中并不代表进程已经睡眠了）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上睡眠的进程，这时current便被唤醒了。

6、poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd_set赋值。

7、如果遍历完所有的fd，还没有返回一个可读写的mask掩码，则会调用schedule_timeout是调用select的进程（也就是current）进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间（schedule_timeout指定），还是没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。

8、把fd_set从内核空间拷贝到用户空间。

1 基本原理



总结：

select的几大缺点：

（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大

（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大

（3）select支持的文件描述符数量太小了，默认是1024

poll() 的实现和 select() 非常相似，只是描述 fd 集合的方式不同，poll() 使用 pollfd 结构而不是 select() 的 fd_set 结构，其他的都差不多。

epoll 的优点主要是一下几个方面：
1）监视的描述符数量不受限制，它所支持的 FD 上限是最大可以打开文件的数目，这个数字一般远大于 2048,举个例子,在 1GB 内存的机器上大约是 10 万左右，具体数目可以 cat /proc/sys/fs/file-max 察看,一般来说这个数目和系统内存关系很大。select() 的最大缺点就是进程打开的 fd 是有数量限制的。这对于连接数量比较大的服务器来说根本不能满足。虽然也可以选择多进程的解决方案(Apache 就是这样实现的)，不过虽然 Linux 上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。

2）I/O 的效率不会随着监视 fd 的数量的增长而下降。select()，poll() 实现需要自己不断轮询所有 fd 集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而 epoll 其实也需要调用 epoll_wait() 不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪 fd 放入就绪链表中，并唤醒在 epoll_wait() 中进入睡眠的进程。虽然都要睡眠和交替，但是 select() 和 poll() 在“醒着”的时候要遍历整个 fd 集合，而 epoll 在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的 CPU 时间。这就是回调机制带来的性能提升。

3）select()，poll() 每次调用都要把 fd 集合从用户态往内核态拷贝一次，而 epoll 只要一次拷贝，这也能节省不少的开销。

在I/O编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者I/O多路复用技术进行处理。I/O多路复用技术通过把多个I/O的阻塞复用到同一个select的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型比，I/O多路复用的最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程的运行，降底了系统的维护工作量，节省了系统资源，I/O多路复用的主要应用场景如下：

服务器需要同时处理多个处于监听状态或者多个连接状态的套接字。

服务器需要同时处理多种网络协议的套接字。

目前支持I/O多路复用的系统调用有 select，pselect，poll，epoll，在Linux网络编程过程中，很长一段时间都使用select做轮询和网络事件通知，然而select的一些固有缺陷导致了它的应用受到了很大的限制，最终Linux不得不在新的内核版本中寻找select的替代方案，最终选择了epoll。epoll与select的原理比较类似，为了克服select的缺点，epoll作了很多重大改进，现总结如下：

1. 支持一个进程打开的socket描述符（FD）不受限制（仅受限于操作系统的最大文件句柄数）。

select最大的缺陷就是单个进程所打开的FD是有一定限制的，它由FD_SETSIZE设置，默认值是1024。对于那些需要支持上万个TCP连接的大型服务器来说显然太少了。可以选择修改这个宏，然后重新编译内核，不过这会带来网络效率的下降。我们也可以通过选择多进程的方案（传统的Apache方案）解决这个问题，不过虽然在Linux上创建进程的代价比较小，但仍旧是不可忽视的，另外，进程间的数据交换非常麻烦，对于Java由于没有共享内存，需要通过Socket通信或者其他方式进行数据同步，这带来了额外的性能损耗，增加了程序复杂度，所以也不是一种完美的解决方案。值得庆幸的是，epoll并没有这个限制，它所支持的FD上限是操作系统的最大文件句柄数，这个数字远远大于1024。例如，在1GB内存的机器上大约是10万个句柄左右，具体的值可以通过cat/proc/sys/fs/filemax察看，通常情况下这个值跟系统的内存关系比较大。

2. I/O效率不会随着FD数目的增加而线性下降。

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，由于网络延时或者链路空闲，任一时刻只有少部分的socket是“活跃”的，但是select/poll每次调用都会线性扫描全部集合，导致效率呈现线性下降。epoll不存在这个问题，它只会对“活跃”的socket进行操作-这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的，那么，只有“活跃”的socket才会主动的去调用callback函数，其他idle状态socket则不会。在这点上，epoll实现了一个伪AIO。针对epoll和select性能对比的benchmark测试表明：如果所有的socket都处于活跃态。例如一个高速LAN环境，epoll并不比select/poll效率高太多；相反，如果过多使用epoll_ctl，效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境，epoll的效率就远在select/poll之上了。

3. 使用mmap加速内核与用户空间的消息传递

无论是select，poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存复制就显得非常重要，epoll是通过内核和用户空间mmap使用同一块内存实现。

4. epoll的API更加简单

用来克服select/poll缺点的方法不只有epoll，epoll只是一种Linux的实现方案。在freeBSD下有kqueue，而dev/poll是最古老的Solaris的方案，使用难度依次递增。但epoll更加简单。

19. LVS十种调度算法：
    1、静态调度：
    ①rr（Round Robin）:轮询调度，轮叫调度
    轮询调度算法的原理是每一次把来自用户的请求轮流分配给内部中的服务器，从1开始，直到N(内部服务器个数)，然后重新开始循环。算法的优点是其简洁性，它无需记录当前所有连接的状态，所以它是一种无状态调度。【提示：这里是不考虑每台服务器的处理能力】

②wrr：weight,加权（以权重之间的比例实现在各主机之间进行调度）
由于每台服务器的配置、安装的业务应用等不同，其处理能力会不一样。所以，我们根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。
③sh:source hashing,源地址散列。主要实现会话绑定，能够将此前建立的session信息保留了
源地址散列调度算法正好与目标地址散列调度算法相反，它根据请求的源IP地址，作为散列键（Hash Key）从静态分配的散列表找出对应的服务器，若该服务器是可用的并且没有超负荷，将请求发送到该服务器，否则返回空。它采用的散列函数与目标地址散列调度算法的相同。它的算法流程与目标地址散列调度算法的基本相似，除了将请求的目标IP地址换成请求的源IP地址，所以这里不一个一个叙述。
④Dh:Destination hashing:目标地址散列。把同一个IP地址的请求，发送给同一个server。
目标地址散列调度算法也是针对目标IP地址的负载均衡，它是一种静态映射算法，通过一个散列（Hash）函数将一个目标IP地址映射到一台服务器。目标地址散列调度算法先根据请求的目标IP地址，作为散列键（Hash Key）从静态分配的散列表找出对应的服务器，若该服务器是可用的且未超载，将请求发送到该服务器，否则返回空。
2、动态调度
①lc（Least-Connection）：最少连接
最少连接调度算法是把新的连接请求分配到当前连接数最小的服务器，最小连接调度是一种动态调度短算法，它通过服务器当前所活跃的连接数来估计服务器的负载均衡，调度器需要记录各个服务器已建立连接的数目，当一个请求被调度到某台服务器，其连接数加1，当连接中止或超时，其连接数减一，在系统实现时，我们也引入当服务器的权值为0时，表示该服务器不可用而不被调度。
简单算法：active*256+inactive(谁的小，挑谁)
②wlc(Weighted Least-Connection Scheduling)：加权最少连接。
加权最小连接调度算法是最小连接调度的超集，各个服务器用相应的权值表示其处理性能。服务器的缺省权值为1，系统管理员可以动态地设置服务器的权限，加权最小连接调度在调度新连接时尽可能使服务器的已建立连接数和其权值成比例。
简单算法：（active*256+inactive）/weight【（活动的连接数+1）/除以权重】（谁的小，挑谁）
③sed(Shortest Expected Delay)：最短期望延迟
基于wlc算法
简单算法：（active+1)*256/weight 【（活动的连接数+1）*256/除以权重】
④nq（never queue）:永不排队（改进的sed）
无需队列，如果有台realserver的连接数＝0就直接分配过去，不需要在进行sed运算。
⑤LBLC（Locality-Based Least Connection）：基于局部性的最少连接
基于局部性的最少连接算法是针对请求报文的目标IP地址的负载均衡调度，不签主要用于Cache集群系统，因为Cache集群中客户请求报文的布标IP地址是变化的，这里假设任何后端服务器都可以处理任何请求，算法的设计目标在服务器的负载基本平衡的情况下，将相同的目标IP地址的请求调度到同一个台服务器，来提高个太服务器的访问局部性和主存Cache命中率，从而调整整个集群系统的处理能力。
基于局部性的最少连接调度算法根据请求的目标IP地址找出该目标IP地址最近使用的

RealServer，若该Real Server是可用的且没有超载，将请求发送到该服务器；若服务器不存在，或者该服务器超载且有服务器处于一半的工作负载，则用“最少链接”的原则选出一个可用的服务器，将请求发送到该服务器。
⑥LBLCR（Locality-Based Least Connections withReplication）：带复制的基于局部性最少链接
带复制的基于局部性最少链接调度算法也是针对目标IP地址的负载均衡，该算法根据请求的目标IP地址找出该目标IP地址对应的服务器组，按“最小连接”原则从服务器组中选出一台服务器，若服务器没有超载，将请求发送到该服务器；若服务器超载，则按“最小连接”原则从这个集群中选出一台服务器，将该服务器加入到服务器组中，将请求发送到该服务器。同时，当该服务器组有一段时间没有被修改，将最忙的服务器从服务器组中删除，以降低复制的程度。

20. A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types (discussed in Enum Types), the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer (discussed in Numbers and Strings).
在Java语言中Swith可以使用参数类型有：Only convertible int values, strings or enum variables are permitted
可以自动转换为整型的（byte,short,int）,String类型，枚举类型。
Java中不能做为Switch参数的有boolean，float,double,long。
原生switch语法的condition支持整数，枚举或者根据上下文能够隐式转换为整数或者枚举的类，再或者是非数组类型的=或{}初始化语句，举例来说就是如下四类：

21.

操作系统

1. 进程的三种基本状态 ： 就绪状态，执行状态，阻塞状态。
处于就绪状态的进程，在调度程序为之分配了处理机之后，该进程便可执行，相应地，它就由就绪状态转变为执行状态。正在执行的进程也称为当前进程，如果因分配给它的时间片已完而被暂停执行时， 该进程便由执行状态又回复到就绪状态； 如果因发生某事件而使进程的执行受阻(例如，进程请求访问某临界资源，而该资源正被其它进程访问时)，使之无法继续执行，该进程将由执行状态转变为阻塞状态。

2. 当系统创建一个新进程时，就为它建立了一个 PCB；进程结束时又回收其 PCB，进程于是也随之消亡。PCB 可以被操作系统中的多个模块读或修改，如被调度程序、资源分配程序、中断处理程序以及监督和分析程序等读或修改。因为 PCB 经常被系统访问，尤其是被运行频率很高的进程及分派程序访问，故 PCB 应常驻内存。系统将所有的 PCB 组织成若干个链表(或队列)，存放在操作系统中专门开辟的 PCB 区内。例如在 Linux 系统中用task_struct 数据结构来描述每个进程的进程控制块，在 Windows 操作系统中则使用一个执行体进程块(EPROCESS)来表示进程对象的基本属性。

3. **进程控制块中的信息**：进程标识符，处理机状态，进程调度信息，进程控制信息

4. **进程的创建**：
   (1) 申请空白 PCB。为新进程申请获得惟一的数字标识符，并从 PCB 集合中索取一个空白 PCB。
   (2) 为新进程分配资源。为新进程的程序和数据以及用户栈分配必要的内存空间。显然，此时操作系统必须知道新进程所需内存的大小。对于批处理作业，其大小可在用户提出创建进程要求时提供。若是为应用进程创建子进程，也应是在该进程提出创建进程的请求中给出所需内存的大小。对于交互型作业，用户可以不给出内存要求而由系统分配一定的空间。如果新进程要共享某个已在内存的地址空间(即已装入内存的共享段)，则必须建立相应的链接。
   (3) 初始化进程控制块。PCB 的初始化包括：① 初始化标识信息，将系统分配的标识符和父进程标识符填入新 PCB 中；② 初始化处理机状态信息，使程序计数器指向程序的入口地址，使栈指针指向栈顶；③ 初始化处理机控制信息，将进程的状态设置为就绪状态或静止就绪状态，对于优先级，通常是将它设置为最低优先级，除非用户以显式方式提出高优先级要求。
   (4) 将新进程插入就绪队列，如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。

5. **进程的终止**：
   (1) 根据被终止进程的标识符，从 PCB 集合中检索出该进程的 PCB，从中读出该进程的状态。
   (2) 若被终止进程正处于执行状态，应立即终止该进程的执行，并置调度标志为真，用于指示该进程被终止后应重新进行调度。
   (3) 若该进程还有子孙进程，还应将其所有子孙进程予以终止，以防它们成为不可控的进程。
   (4) 将被终止进程所拥有的全部资源，或者归还给其父进程，或者归还给系统。
   (5) 将被终止进程(PCB)从所在队列(或链表)中移出，等待其他程序来搜集信息。

6. **同步机制应遵循的规则**： 空闲让进，忙则等待，有限等待，让权等待；

7. 进程同步机制：信号量，管道；

8. 进程通信的类型：
   共享存储器系统：基于共享数据结构的通信方式，基于共享存储区的通信方式。
   消息传递系统：
   管道通信；
   在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区，除了需要为进程设置消息缓冲队列外，还应在进程的 PCB 中增加消息队列队首指针，用于对消息队列进行操作。

9. 在引入线程的 OS 中，通常都是把**进程作为分配资源的基本单位**，而把**线程作为独立运行和独立调度的基本单位**。

10. **线程的状态**：**(1) 状态参数。**在 OS 中的每一个线程都可以利用线程标识符和一组状态参数进行描述。状态参数通常有这样几项：① 寄存器状态，它包括程序计数器 PC 和堆栈指针中的内容；② 堆栈，在堆栈中通常保存有局部变量和返回地址；③ 线程运行状态，用于描述线程正处于何种运行状态；④ 优先级，描述线程执行的优先程度；⑤ 线程专有存储器，用于保存线程自己的局部变量拷贝；⑥ 信号屏蔽，即对某些信号加以屏蔽。**(2) 线程运行状态。**如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。相应地，线程在运行时也具有下述三种基本状态: ① 执行状态，表示线程正获得处理机而运行；② 就绪状态，指线程已具备了各种执行条件，一旦获得 CPU 便可执行的状态；③ 阻塞状态，指线程在执行中因某事件而受阻，处于暂停执行时的状态。

11. **线程间的同步和通信**：互斥锁(mutex)；条件变量；信号量机制；

12. **线程的实现方式**：线程已在许多系统中实现，但各系统的实现方式并不完全相同。在有的系统中，特别是一些数据库管理系统如 Infomix，所实现的是用户级线程(UserLevel Threads)；而另一些系统(如 Macintosh 和 OS/2 操作系统)所实现的是内核支持线程(KernelSupported Threads)； 还有一些系统如 Solaris 操作系统，则同时实现了这两种类型的线程。
内核支持线程：对于通常的进程，无论是系统进程还是用户进程，进程的创建、 撤消，以及要求由系统设备完成的 I/O 操作，都是利用系统调用而进入内核，再由内核中的相应处理程序予以完成的。进程的切换同样是在内核的支持下实现的。因此我们说，不论什么进程，它们都是在操作系统内核的支持下运行的，是与内核紧密相关的。这里所谓的内核支持线程 KST(Kernel Supported Threads)，也都同样是在内核的支持下运行的，即无论是用户进程中的线程，还是系统进程中的线程，他们的创建、撤消和切换等也是依靠内核，在内核空间实现的。此外，在内核空间还为每一个内核支持线程设置了一个线程控制块，内核是根据该控制块而感知某线程的存在，并对其加以控制。 **相应优点：**(1) 在多处理器系统中，内核能够同时调度同一进程中多个线程并行执行；(2) 如果进程中的一个线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程；(3) 内核支持线程具有很小的数据结构和堆栈，线程的切换比较快，切换开销小；(4) 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率。
用户级线程：用户级线程 ULT(User Level Threads)仅存在于用户空间中。对于这种线程的创建、撤消、线程之间的同步与通信等功能，都无须利用系统调用来实现。对于用户级线程的切换，通常发生在一个应用进程的诸多线程之间，这时，也同样无须内核的支持。由于切换的规则远比进程调度和切换的规则简单，因而使线程的切换速度特别快。可见，这种线程是与内核无关的。我们可以为一个应用程序建立多个用户级线程。在一个系统中的用户级线程的数目可以达到数百个至数千个。由于这些线程的任务控制块都是设置在用户空间，而线程所执行的操作也无须内核的帮助，因而**内核完全不知道用户级线程的存在。**
值得说明的是，对于设置了用户级线程的系统，其调度仍是以进程为单位进行的。假如系统中设置的是内核支持线程，则调度便是以线程为单位进行的。在采用轮转法调度时，是各个线程轮流执行一个时间片。
用户级线程的优点：(1) 线程切换不需要转换到内核空间，对一个进程而言，其所有线程的管理数据结构均
在该进程的用户空间中，管理线程切换的线程库也在用户地址空间运行。因此，进程不必切换到内核方式来做线程管理，从而节省了模式切换的开销，也节省了内核的宝贵资源。(2) 调度算法可以是进程专用的。在不干扰操作系统调度的情况下，不同的进程可以根据自身需要，选择不同的调度算法对自己的线程进行管理和调度，而与操作系统的低级调度算法是无关的。
(3) 用户级线程的实现与操作系统平台无关，因为对于线程管理的代码是在用户程序内的，属于用户程序的一部分，所有的应用程序都可以对之进行共享。因此，用户级线程甚至可以在不支持线程机制的操作系统平台上实现。 **缺点：**(1) 系统调用的阻塞问题。在基于进程机制的操作系统中， 大多数系统调用将阻塞进程，因此，当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程都会被阻塞。而在内核支持线程方式中，则进程中的其它线程仍然可以运行。(2) 在单纯的用户级线程实现方式中，多线程应用不能利用多处理机进行多重处理的优点。内核每次分配给一个进程的仅有一个 CPU，因此进程中仅有一个线程能执行，在该线程放弃 CPU 之前，

其它线程只能等待。
组合调用模式：有些操作系统把用户级线程和内核支持线程两种方式进行组合，提供了组合方式
ULT/KST 线程。在组合方式线程系统中，内核支持多 KST 线程的建立、调度和管理，同时，
也允许用户应用程序建立、调度和管理用户级线程。一些内核支持线程对应多个用户级线
程，程序员可按应用需要和机器配置对内核支持线程数目进行调整，以达到较好的效果。
组合方式线程中，同一个进程内的多个线程可以同时在多处理器上并行执行，而且在阻塞
一个线程时，并不需要将整个进程阻塞。所以，组合方式多线程机制能够结合 KST 和 ULT
两者的优点，并克服了其各自的不足。

13. 线程的实现：不论是进程还是线程，都必须直接或间接地取得内核的支持。由于内核支持线程可以直接利用系统调用为它服务，故线程的控制相当简单；而用户级线程必须借助于某种形式的中间系统的帮助方能取得内核的服务，故在对线程的控制上要稍复杂些。
    参考书：P78-81

14. **进程调度中的三个基本机制**：排队器，分派器， 上下文切换机制。

15. 进程调度方式：非抢占方式，抢占方式

16. 所谓周转时间，是指从作业被提交给系统开始，到作业完成为止的这段时间间隔(称为作业周转时间)。作业的周转时间 T 与系统为它提供服务的时间 Ts 之比，即 W = T/Ts，称为带权周转时间

17. 先来先服务(FCFS)调度算法比较有利于长作业(进程)，而不利于短作业(进程)。FCFS 调度算法有利于 CPU 繁忙型的作业，而不利于 I/O 繁忙型的作业(进程)。SJF 调度算法能有效地降低作业的平均等待时间，提高系统吞吐量。短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。

18. 在时间片轮转算法中，时间片的大小对系统性能有很大的影响，如选择很小的时间片将有利于短作业，因为它能较快地完成，但会频繁地发生中断、进程上下文的切换，从而增加系统的开销；反之，如选择太长的时间片，使得每个进程都能在一个时间片内完成，时间片轮转算法便退化为 FCFS 算法， 无法满足交互式用户的需求。 一个较为可取的大小是，时间片略大于一次典型的交互所需要的时间。这样可使大多数进程在一个时间片内完成。

19. 产生死锁的原因可归结为如下两点：(1) 竞争资源。当系统中供多个进程共享的资源如打印机、公用队列等，其数目不足以满足诸进程的需要时，会引起诸进程对资源的竞争而产生死锁。(2) 进程间推进顺序非法。进程在运行过程中，请求和释放资源的顺序不当，也同样会导致产生进程死锁。

20. 产生死锁的必要条件：互斥条件，请求和保持条件，不剥夺条件，环路等待条件；

21. 处理死锁的基本方法：预防，避免，检测，解除。预防：摒弃“请求和保持”条件，摒弃“不剥夺”条件，摒弃“环路等待”条件。

22. 银行家算法：

    数据结构：(1) **可利用资源向量 Available**。这是一个含有 m 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 Available[j]=K，则表示系统中现有 R j类资源K 个。(2) **最大需求矩阵 Max**。这是一个 n×m 的矩阵，它定义了系统中 n 个进程中的每一个进程对 m 类资源的最大需求。如果 Max[i,j]=K，则表示进程 i 需要 Rj 类资源的最大数目为 K。(3) **分配矩阵 Allocation**。这也是一个 n×m 的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 Allocation[i,j]=K，则表示进程 i 当前已分得 R j类资源的数目为 K。(4) **需求矩阵 Need**。 这也是一个 n×m 的矩阵，用以表示每一个进程尚需的各类资源数。如果 Need[i,j]=K，则表示进程 i 还需要 R j类资源 K 个，方能完成其任务。上述三个矩阵间存在下述关系：Need[i, j]=Max[i, j]-Allocation[i, j]；

    算法流程：设 Request_i是进程 Pi的请求向量，如果 Request_i[j]=K，表示进程 P i 需要 K 个 R j类型的资源。当 P i发出资源请求后，系统按下述步骤进行检查：(1) 如果 Request_i[j]≤Need[i,j]，便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。(2) 如果 Request_i[j]≤Available[j]，便转向步骤(3)；否则，表示尚无足够资源，Pi 须等待。(3) 系统试探着把资源分配给进程 P i，并修改下面数据结构中的数值：

    Available[j] := Available[j] - Request_i[j]；

    Allocation[i,j] := Allocation[i,j] + Request_i[j]；

    Need[i,j] := Need[i,j] - Request_i[j]；

    (4) 系统执行安全性算法，检查此次资源分配后系统是否处于安全状态。若安全，才正式将资源分配给进程 Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程 Pi等待。

    安全性算法: 系统所执行的安全性算法可描述如下：(1) 设置两个向量：① 工作向量 Work，它表示系统可提供给进程继续运行所需的各类资源数目，它含有 m 个元素，在执行安全算法开始时，Work:=Available。

    ② Finish，它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做 Finish[i]:=false；当有足够资源分配给进程时，再令 Finish[i]:=true。(2) 从进程集合中找到一个能满足下述条件的进程：① Finish[i]=false；② Need[i,j]≤Work[j]；若找到，执行步骤(3)，否则，执行步骤(4)。(3) 当进程 Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：Work[j]:= Work[j]+Allocation[i,j]；Finish[i]:=true；go to step 2；(4) 如果所有进程的 Finish[i]=true 都满足，则表示系统处于安全状态；否则，系统处于不安全状态。

23. 如何**将一个用户源程序变为一个可在内存中执行的程序**，通常都要经过以下几个步骤：首先是要编译，由编译程序(Compiler)将用户源代码编译成若干个目标模块(Object Module)；其次是链接，由链接程序(Linker)将编译后形成的一组目标模块，以及它们所需要的库函数链接在一起，形成一个完整的装入模块(Load Module)；最后是装入，由装入程序(Loader)将装入模块装入内存。

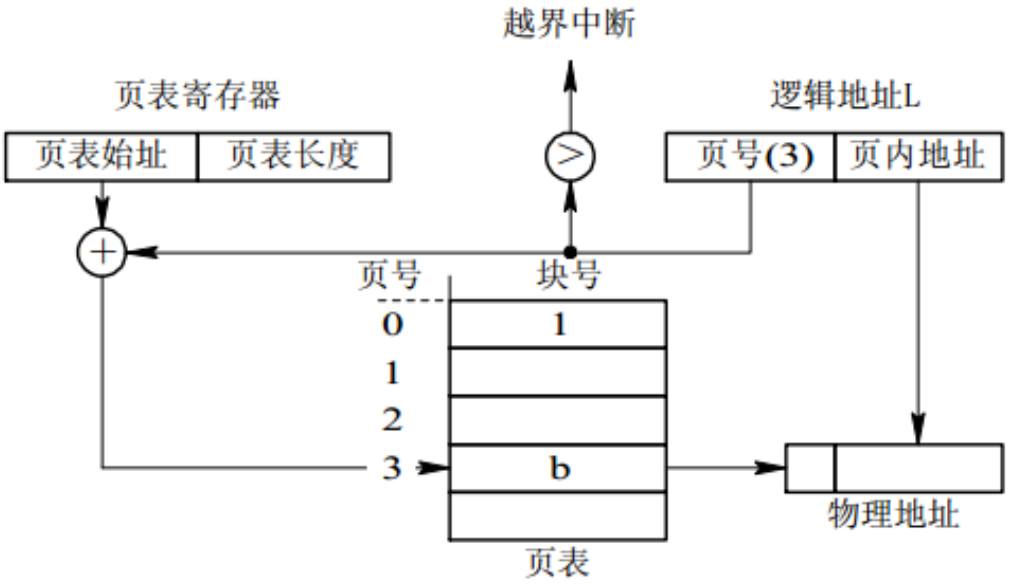
24. 在将一个装入模块装入内存时，可以有绝**对装入方式、可重定位装入方式和动态运行时装入**方式.

- 绝对装入方式只能将目标模块装入到内存中事先指定的位置。在多道程序环境下，编译程序不可能预知所编译的目标模块应放在内存的何处，因此，绝对装入方式只适用于单道程序环境。
- 通常是把在装入时对目标程序中指令和数据的修改过程称为重定位。又因为地址变换通常是在装入时一次完成的，以后不再改变，故称为静态重定位。
- 可重定位装入方式可将装入模块装入到内存中任何允许的位置，故可用于多道程序环境；但这种方式并不允许程序运行时在内存中移动位置。因为，程序在内存中的移动，意味着它的物理位置发生了变化，这时必须对程序和数据的地址(是绝对地址)进行修改后方能运行。然而，实际情况是，在运行过程中它在内存中的位置可能经常要改变，此时就应采用动态运行时装入的方式.
- 动态运行时的装入程序在把装入模块装入内存后，并不立即把装入模块中的相对地址转换为绝对地址，而是把这种地址转换推迟到程序真正要执行时才进行。因此，装入内存后的所有地址都仍是相对地址。

25. 程序的链接: 静态链接，装入时动态链接, 运行时动态链接.

26. **连续分配方式**，是指为一个用户程序分配一个连续的内存空间。可把连续分配方式进一步分为**单一连续分配、固定分区分配、动态分区分配以及动态重定位分区分配**四种方式。
    固定分区式分配是将内存用户空间划分为若干个固定大小的区域，在每个分区中只装入一道作业，这样，把用户空间划分为几个分区，便允许有几道作业并发运行。
    动态分区分配是根据进程的实际需要，动态地为之分配内存空间。为了实现对空闲分区的分配和链接，在每个分区的起始部分，设置一些用于控制分区分配的信息，以及用于链接各分区所用的前向指针；在分区尾部则设置一后向指针，通过前、后向链接指针，可将所有的空闲分区链接成一个双向链.
    哈希算法就是利用哈希快速查找的优点，以及空闲分区在可利用空间表中的分布规律，建立哈希函数，构造一张以空闲分区大小为关键字的哈希表，该表的每一个表项记录了一个对应的空闲分区链表表头指针。
    在连续分配方式中，必须把一个系统或用户程序装入一连续的内存空间。如果在系统中只有若干个小的分区，即使它们容量的总和大于要装入的程序，但由于这些分区不相邻接，也无法把该程序装入内存。在动态运行时装入的方式中，作业装入内存后的所有地址都仍然是相对地址，将相对地址转换为物理地址的工作，被推迟到程序指令要真正执行时进行。为使地址的转换不会影响到指令的执行速度，必须有硬件地址变换机构的支持，即须在系统中增设一个重定位寄存器，用它来存放程序(数据)在内存中的起始地址。程序在执行时，真正访问的内存地址是相对地址与重定位寄存器中的地址相加而形成的。地址变换过程是在程序执行期间，随着对每条指令或数据的访问自动进行的，故称为动态重定位。当系统对内存进行了“紧凑”而使若干程序从内存的某处移至另一处时，不需对程序做任何修改，只要用该程序在内存的新起始地址，去置换原来的起始地址即可。

27. **基本分页存储管理方式**：分页存储管理是将一个进程的逻辑地址空间分成若干个大小相等的片，称为页面或页，并为各页加以编号，从 0 开始，如第 0 页、第 1 页等。相应地，也把内存空间分成与页面相同大小的若干个存储块，称为(物理)块或页框(frame)，也同样为它们加以编号，如 0#块、1#块等等。在为进程分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于进程的最后一页经常装不满一块而形成了不可利用的碎片，称之为“页内碎片” 。
系统为每个进程建立了一张页面映像表，简称页表。



由于页表是存放在内存中的，这使 CPU 在每存取一个数据时，都要两次访问内存。第一次是访问内存中的页表，从中找到指定页的物理块号，再将块号与页内偏移量 W 拼接，以形成物理地址。第二次访问内存时，才是从第一次所得地址中获得所需数据(或向此地址中写入数据)。为了提高地址变换速度，可在地址变换机构中增设一个具有并行查寻能力的特殊高速缓冲寄存器，又称为“联想寄存器”(Associative Memory)，或称为“快表” ，在 IBM 系统中又取名为 TLB(Translation Lookaside buffer)，用以存放当前访问的那些页表项。

28. 分段存储管理方式的引入： 方便编程，信息共享，信息保护，动态增长，动态链接。

29. 所谓虚拟存储器，是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。其逻辑容量由内存容量和外存容量之和所决定，其运行速度接近于内存速度，而每位的成本却又接近于外存。

30. 对磁盘的访问时间分成以下三部分： 寻道时间 Ts ， 旋转延迟时间 Tr， 传输时间 Tt

31. **外存分配方式** ： **连续分配、链接分配和索引分配。**在采用连续分配方式时的文件物理结构，将是顺序式的文件结构；链接分配方式将形成链接式文件结构；而索引分配方式则将形成索引式文件结构。

    - **连续分配的主要优点**如下：(1) 顺序访问容易。(2) 顺序访问速度快。**连续分配的主要缺点**如下：(1) 要求有连续的存储空间。(2) 必须事先知道文件的长度。
    - **链接方式**又可分为**隐式链接**和**显式链接**两种形式。在采用隐式链接分配方式时，在文件目录的每个目录项中，都须含有指向链接文件第一个盘块和最后一

个盘块的指针。在每个盘块中都含有一个指向下一个盘块的指针。 **隐式链接分配方式的主要问题**在于：它只适合于顺序访问，它对随机访问是极其低效的。
    - **显式链接**：指把用于链接文件各物理块的指针，显式地存放在内存的一张链接表中（文件分配表 FAT）。该表在整个磁盘仅设置一张。 在每个表项中存放链接指针，即下一个盘块号。在该表中，凡是属于某一文件的第一个盘块号，或者说是每一条链的链首指针所对应的盘块号，均作为文件地址被填入相应文件的 FCB 的“物理地址”字段中。
    - 在微软公司的早期 MS-DOS 中，所使用的是 12 位的 FAT12 文件系统，后来为 16 位的FAT16文件系统；在 Windows 95和 Windows 98操作系统中则升级为 32位的 FAT32； WindowsNT、Windows 2000 和 Windows XP 操作系统又进一步发展为新技术文件系统 NTFS(NewTechnology File System)。
    - **链接分配方式的问题**：(1) 不能支持高效的直接存取。(2) FAT 需占用较大的内存空间。
    - **索引分配**：为每个文件分配一个索引块(表)，再把分配给该文件的所有盘块号都记录在该索引块中，因而该索引块就是一个含有许多盘块号的数组。在建立一个文件时，只需在为之建立的目录项中填上指向该索引块的指针。
    - 为了能对一个文件进行正确的存取，必须为文件设置用于描述和控制文件的数据结构，称之为**“文件控制块(FCB)”**

32. 文件存储空间的管理： 空闲表法和空闲链表法， 位示图法，成组链接法

33. 当应用程序中需要操作系统提供服务时，如请求 I/O 资源或执行 I/O 操作，应用程序必须使用系统调用命令。由操作系统捕获到该命令后，便将 CPU 的状态从用户态转换到系统态，然后执行操作系统中相应的子程序(例程)，完成所需的功能。执行完成后，系统又将CPU 状态从系统态转换到用户态，再继续执行应用程序。

34.
35.
36.

数据库

1. 关系模型的完整性规则: 实体完整性规则， 参照完整性规则， 用户定义的完整性规则；
   实体完整性规则：关系中元组的主键值不能为空；
   参照完整性规则：外键必须是另一个表的主键的有效值或者空值，
   用户定义的完整性规则：

2. 与Mysql服务器相互作用的通讯协议包括**TCP/IP，Socket，共享内存，命名管道；**
   TCP/IP协议，通常我们通过来连接MySQL，各种主要编程语言都是根据这个协议实现了连接模块
   Unix Socket协议，这个通常我们登入MySQL服务器中使用这个协议，因为要使用这个协议连接MySQL需要一个物理文件，文件的存放位置在配置文件中有定义，值得一提的是，这是所有协议中最高效的一个。
   Share Memory协议，这个协议一般人不知道，肯定也没用过，因为这个只有windows可以使用，使用这个协议需要在配置文件中在启动的时候使用–shared-memory参数，注意的是，使用此协议，一个host上只能有一个server，所以这个东西一般没啥用的，除非你怀疑其他协议不能正常工作，实际上微软的SQL Sever也支持这个协议
   Named Pipes协议，这个协议也是只有windows才可以用，同shared memory一样，使用此协议，一个host上依然只能有一个server，即使是使用不同的端口也不行，Named Pipes 是为局域网而开发的协议。内存的一部分被某个进程用来向另一个进程传递信息，因此一个进程的输出就是另一个进程的输入。第二个进程可以是本地的（与第一个进程位于同一台计算机上），也可以是远程的（位于联网的计算机上）。正因为如此，假如你的环境中没有或者禁用TCP/IP环境，而且是windows服务器，那么好歹你的数据库还能工作。使用这个协议需要在启动的时候添加–enable-named-pipe选项

3.
4.
5.
6.
7.