- 三次握手：
  - 检验双方收发功能完好
  - 避免server端收到失效SYN并一直等待，从而浪费资源
  - 第一二次握手不能携带数据（防止恶意攻击），第三次可以（因为连接已经半建立）
  - Server收到第一次握手的SYN后，将client的相关信息放入半连接队列，并发送SYN + ACK
  - Server收到第三次握手的ACK后，将client的相关信息从半连接队列中移至全连接队列
  - Accept会从全连接队列中取出一个client处理

- 四次挥手：
  - Socket是双工的，收发端需要各自单独关闭
  - 步骤：
    - Client发送FIN，进入FIN_WAIT阶段，Server接收FIN
    - Server发送ACK，进入CLOSE_WAIT状态
    - Client接收ACK，关闭client发-server收的通道
    - Server发送FIN，进入LAST_ACK阶段
    - Client接收FIN，进入TIME_WAIT阶段
    - Server接收ACK，关闭server发-client收的通道
  - TIME_WAIT：
    - 出现在主动断开方，发送最后一次ACK后
    - 客户端无法保证最后的ACK能传到，为了随时接收对方重发的FIN
    - 可使用SO_REUSEADDR快速复用

- TCP相关：
  - Accept发生在三次握手之后
  - TCP和UDP的区别：
    - TCP是connection-oriented，UDP是connectionless的
    - Socket是TCP/IP协议的封装API
  - SYN：用于建立连接
    - 连接请求：SYN = 1, ACK = 0
    - 连接回复：SYN = 1, ACK = 1
  - 如何保证可靠传输：
    - 建立连接，可接受反馈信息
    - 确认 & 重传
    - 数据校验

        - 数据分片及排序（Seq #）
            - 回绕解决方法：
                - 查看时间戳
                - 两次Seq # 相减并转换为有符号整形，判断正负
        - 流量控制（接收方提示发送方降低发送速率）
        - 拥塞控制（网络拥塞时减少数据发送）
- TCP拥塞控制：
    - 发送方维护拥塞窗口，从较小值开始指数增大，超过一定阈值后降低增长速率
    - 发生超时时，该阈值降低一半，拥塞窗口减小，重新回到慢启动阶段

- HashTable和HashMap：
    - HashTable：线程安全
    - HashMap：
        - 非线程安全：缺乏同步机制，多个线程resize时可能造成无法停止的转移操作，极易造成死循环或数据丢失
        - 允许null key & null value
        - 是HashTable的轻量化实现

- C++内存布局：堆，栈，全局/静态存储区，常量存储区

- malloc和new的区别：
    - new返回值为申请类型指针，malloc为void *
    - new失败时抛出异常，malloc返回NULL
    - new无需指定分配大小，malloc需要
    - new/new[]会自动调用构造函数，malloc只是给一片连续内存
    - new可被重载，malloc不行

- 线程间通信方式：
    - 线程间通信目的主要是线程同步，所以没有像IPC中用于数据交换的通信机制
    - 锁（mutex）：互斥锁，自旋锁，读写锁
    - 信号（signal）：一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是进程间通信机制中唯一的异步通信机制，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。进程之间可以互相通过系统调用kill发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。信号机制除了基本通知功能外，还可以传递附加信息。
    - 信号量（semaphore）：信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访

问该信号量保护的共享资源，初始值为1就变成互斥锁（Mutex），即同时只能有一个任务可以访问信号量保护的共享资源。
一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作将把信号量的值减1，若当前信号量的值为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可用；若当前信号量的值为非负数，表示可以获得信号量，因而可以立刻访问被该信号量保护的共享资源。
　　当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加1实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

- 进程间通信方式（IPC）：
  - 管道（pipe）：
    - 半双工，需要建立两个管道
  - 只可用于父子进程或兄弟进程
  - FIFO：
    - 独立于进程，可被复用
  - 信号（signal）：
    - 用于对其他进程做出指令
  - 可自定义信号处理函数
  - 可选择处理或忽略
  - 消息队列：
    - 一个消息的链表
  - 允许多个进程读写
  - 共享内存：
    - 最快的IPC
  - 需要同步机制
  - 一般结合了其他通信机制，如信号量
  - 套接字（socket）：
    - 可与不同计算机上的进程通信
  - 可被多个进程同时使用，需要区分port

- GET和POST的区别：
  - GET一般用于页面获取，POST一般用于提交数据，但其实非要反过来用也能用，但很傻
  - GET可能被cache，POST不会
  - 用GET提交数据时参数直接暴露在URL上，更不安全
  - GET参数通过URL传递，POST则通过request body传递
  - POST有时会发送两个数据包，第一个发header，服务器返回100 continue，再发body，接收

200 ok

- Linux操作：
  - 查看进程：ps
  - 动态显示进程状况：top
  - 查看IO性能：iostat
  - 查看网络状态：netstat
  - 虚拟内存使用状况：vm_stat
  - 终止进程：kill -9 [pid]
  - 文本搜索：grep [content] [filename]
  - 复制：cp [file1] [file2] ... [dest]
    - grep怎么输出文本中包含某个关键字的行，不包含的呢
      - grep 同时满足多个关键字
        ① grep -E "word1|word2|word3" file.txt
        满足任意条件（word1、word2和word3之一）将匹配。
        ② grep word1 file.txt | grep word2 |grep word3
        必须同时满足三个条件（word1、word2和word3）才匹配。
      - grep同时排除文本中包含某个关键字的行

      不说废话, 例如需要排除 abc.txt 中的 mmm nnn

      ```
      grep -v 'mmm\|nnn' abc.txt
      ```

      - grep进行文件内容匹配工作是用到的参数主要有两个，分别是
        1. 取出两个文件中的相同部分内容“-wf”参数.
        2. 取出两个文件中的不同部分内容“-wvf”参数
  - disk usage,是通过搜索文件来计算每个文件的大小然后累加，du能看到的文件只是一些当前存在的，没有被删除的。他计算的大小就是当前他认为存在的所有文件大小的累加和:du
  - disk free: 通过文件系统来快速获取空间大小的信息，当我们删除一个文件的时候，这个文件不是马上就在文件系统当中消失了，而是暂时消失了，当所有程序都不用时，才会根据OS的规则释放掉已经删除的文件， df记录的是通过文件系统获取到的文件的大小，他比du强的地方就是能够看到已经删除的文件，而且计算大小的时候，把这一部分的空间也加上了，更精确了。当文件系统也确定删除了该文件后，这时候du与df就一致了.
  - 显示当前工作目录：pwd

- 解决竞争冒险：
  - 加锁

- Thread local storage

- HTTP请求类型
  - GET：请求指定资源，用于获取数据，等幂
  - HEAD：请求一个与GET相同的响应，但没有响应体，等幂
  - POST：在服务端创建一个新资源，非等幂
  - PUT：对资源整体覆盖，等幂
  - DELETE：删除指定资源，等幂
  - CONNECT：建立用户与服务端通道，非等幂
  - OPTIONS：获取目的资源支持的通信选项（检测服务器所支持的请求方法），等幂
  - TRACE：沿通向目标资源路径的消息环回测试（最终接收者原样反射接收到的信息），等幂
  - PATCH：对资源进行部分修改，非等幂

- HTTP响应类型
  - 100：信息响应，目前为止所有内容可行，客户端可继续请求
  - 200：成功响应，请求被成功处理
  - 300：重定向，客户端访问资源已被移动
    -

301	**Moved Permanently**	**永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替**
302	Found	临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI
304	Not Modified	未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源
307	Temporary Redirect	临时重定向。与302类似。使用GET请求重定向

  - 400：客户端错误，客户端发送的请求无法被服务器处理
    -

400	**Bad Request**	**客户端请求的语法错误，服务器无法理解**
401	Unauthorized	请求要求用户的身份认证
403	Forbidden	服务器理解请求客户端的请求，但是拒绝执行此请求
404	Not Found	服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面

- 500：服务器错误，请求有效，但服务器运行出错
  -

500	**Internal Server Error**	**服务器内部错误，无法完成请**
502	Bad Gateway	作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应

- MVC：
  - 实现动态程序设计，简化后续修改和扩展，提高程序部分复用性
  - Model：功能实现及数据库管理
  - View：图形界面设计
  - Controller：负责转发请求，对请求进行处理
- MTV：
  - M：模型层，功能同上
  - T：templates:模板层，用于处理用户显示部分的内容，和MVC中的V是一样的，通过html展示
  - V：views 视图层，在MTV中视图层是处理用户交互的部分，从模型层中获取数据，再将数据交给模板层，再显示给用户。和MVC中的控制层用法一样
- ORM (Object-Relational Mapping)
  - 将关系数据库中的记录映射成为对象，以对象的形式展现，可将对数据库的操作转变为对对象的操作
  - 方便开发人员以面向对象思想来实现对数据库的操作

- REST (REpresentational State Transfer)
  - 每一个URI代表一种资源；
  - 客户端和服务器之间，传递这种资源的某种表现层；
  - 客户端通过四个HTTP动词，对服务器端资源进行操作，实现”表现层状态转化”。
    Web应用要满足REST最重要的原则是:客户端和服务器之间的交互在请求之间是无状态的,即从客户端到服务器的每个请求都必须包含理解请求所必需的信息。如果服务器在请求之间的任何时间点重启，客户端不会得到通知。此外此请求可以由任何可用服务器回答，这十分适合云计算之类

的环境。因为是无状态的，所以客户端可以缓存数据以改进性能。
    - 另一个重要的REST原则是**系统分层**，这表示组件无法了解除了与它直接交互的层次以外的组件。通过将系统知识限制在单个层，可以限制整个系统的复杂性，从而促进了底层的独立性。

- 死锁：
    - 产生条件：
        - 互斥条件：资源只能被一个线程/进程使用（无法避免）
        - 占有且等待条件：线程/进程因请求资源而等待时，不会释放已分配资源
        - 不可强行剥夺条件：线程/进程已获得的资源在使用完以前不会被强行剥夺
        - 循环等待条件：若干线程/进程形成头尾相连的循环等待资源关系
    - 解决办法（破坏任意一个死锁成立条件）：
        - 预防：
            - 资源一次性分配
            - 可剥夺资源
            - 资源有序分配
        - 避免：分配资源前先进行风险评估
        - 检测：建立资源分配表和线程/进程等待表
        - 解除：
            - 从其他线程/进程强制剥夺资源给死锁线程/进程
            - 撤销死锁线程/进程

- 事务（Transaction）：
    - 访问数据库的一系列操作集合，由DBMS实现
    - 并发事务问题：
        - Lost Update Problem
        - Temporary Update Problem
        - Incorrect Summary Problem
    - ACID：
        - Atomicity：一次事务中所有操作不可分割（DBMS）
        - Consistency：事务开始和结束后，数据库完整性约束没有被破坏，多个并行事务的执行结果必须与某一顺序的串行执行结果一致
        - Isolation：一个事务的执行对其他事务不造成影响
        - Durability：已提交事务对数据库做出的改变不被丢失，不会回滚
    - 数据库的隔离级别：
        - read uncommitted
    - Read committed
    - Repeatable read

    - Serializable

- 条件变量：
    - wait：阻塞，直到收到signal或bradcast
    - signal：使另一线程停止阻塞
    - broadcast：使所有线程停止阻塞

- 如何避免内存泄漏：
    - 检测内存泄漏：Valgrind
    - 在所有能用STL的地方不要自己设计
    - 使用智能指针申请资源
    - 善用RAII：通过设计析构函数，合理释放已分配资源
    - 抛出异常时先catch，释放资源，再rethrow

- B树：
    - 定义：
        - 每个节点包含多个关键字（key-value pair），从小到大排列
        - 每个节点的子节点数在$(1, M]$之间
        - 每个节点的关键字数在$[ceil(m/2) - 1, M - 1]$之间
        - 所有叶子节点在同一层
    - 特点：
        - 每个节点关键字增多，层级比二叉树更少，减少了数据查询的次数和复杂度
        - 在应用到数据库时，由于每次磁盘读取为一个磁盘块，将节点大小设置为磁盘块大小，可以一次读取出来

- B+树：
    - 定义：
        - 非叶子结点只保存key不保存value
        - 叶子节点保存了所有value
        - 叶子节点关键字从小到大排列，结尾保存右侧叶子节点开始数据的指针
    - 特点：
        - 比普通B树层级更少，查询更快
        - 所有key-value pair都在叶子节点，每次查找次数相同，查询速度更稳定
        - 所有叶子节点构成一有序链表，查找区间更加方便
        - 全表扫描更快，只需扫描所有叶子节点
        - 经常访问的数据离根节点很近时，检索效率不如B树

- JOIN：
  - 基于表之间的共通字段，将两表结合起来

id	name	id	address
1	Google	1	美国
2	淘宝	3	中国
3	微博	5	中国
4	Facebook	6	美国

（以上为两个表）

  - INNER JOIN：
    - 左表和右表满足ON的行

id	name	address
1	Google	美国
3	微博	中国

  - LEFT JOIN：
    - 左表全部行 + 右表满足ON条件的行
    - 左表行在右表无匹配时，填NULL

id	name	address
1	Google	美国
2	淘宝	NULL
3	微博	中国
4	Facebook	NULL

  - RIGHT JOIN：
    - 左表满足ON条件的行 + 右表全部行
    - 右表行在左表无匹配时，填NULL

id	name	address
1	Google	美国
3	微博	中国
5	NULL	中国
6	NULL	美国

- FULL JOIN：
  - 左表所有行 + 右表所有行
  - 若一个表行在另一个表中无匹配，填NULL

id	name	address
1	Google	美国
2	淘宝	NULL
3	微博	中国
4	Facebook	NULL
5	NULL	中国
6	NULL	美国

- Linux编译过程：
  - 预处理（.i）
  - 编译 (.s)
    - 生成汇编代码（Assembly）
  - 汇编 (.o)
    - 将汇编代码转为二进制码
  - 链接
    - 生成可执行文件

.o 就相当于windows里的obj文件 ， 一个.c或.cpp文件对应一个.o文件
.a 是好多个.o合在一起,用于静态连接 ， 即STATIC mode，多个.a可以链接生成一个exe的可执行文件
.so 是shared object,用于动态连接的,和windows的dll差不多，使用时才载入。

- Linux内存管理：
  - 每个进程拥有一段虚拟内存（Virtual Memory）

    - 每段虚拟内存通过Page Table映射到实际内存
    - Page Table可能有多级

- kill -9 杀不死进程的原因：
    - 该进程为僵尸进程，已释放所有资源，但无父进程确认
    - 该进程处于核心态且在等待不可获得的资源

- 互斥锁和自旋锁的区别：
    - 互斥锁：
        - Sleep-waiting型锁，在请求锁时会block，CPU可以跳过这个线程做其他事
        - 起始开销较大，但不会因为线程持锁时间增大
        - 一般用于持锁时间较长的操作
    - 自旋锁：
        - Busy-waiting型锁，在请求锁时会一直占用CPU
        - 死循环检测，全程消耗CPU
        - 起始消耗较小，但随持锁时间线性增长
        - 一般用于短时间持锁且CPU资源充裕的情况下（如多核服务器）

- C++ allocator原理：
    - 第一层配置器：
        - 申请内存大于128bytes
        - 直接使用malloc、free及realloc
        - 内存不足时调用用户自定义的内存不足处理函数，继续申请内存，如果未定义该函数则抛出异常
    - 第二层配置器：
        - 申请内存小于128bytes
        - 维护一个大小为16的free-list（编号$n \in [0, 15]$），分别指向大小为$8(n + 1)bytes$的内存块
        - 没有所需内存块时，调用refill函数申请N个新的内存块（一般为20个）

- Linux IO模型：
    - 同步：调用者主动等待；异步：被调用者在完成后通知调用者
    - 同步阻塞IO：
        - 整个进程进入阻塞（blocking）状态，CPU转去处理其他进程
        - 内核开发者省事，用户性能下降。。。
    - 同步非阻塞IO：

        - 进程进入轮询（polling）状态，IO不会马上完成，可能会返回一个error
        - 后台执行，进程可以做其他事情
        - 任务可能在两次轮询间完成，整体吞吐量降低
    - IO多路复用（属于同步阻塞）：
        - 包含select、poll和epoll
        - kernel会负责轮询所有进程，无需进程自己轮询导致占用大量CPU资源
    - 信号驱动IO：
        - 收到SIGIO时进行处理
        - 数据准备阶段异步
    - 异步非阻塞IO：
        - 发起操作后立刻返回，进程不会阻塞
        - kernel等待数据完成后，拷贝到用户内存，然后给该进程一个signal
        - 数据准备和处理阶段均异步

- 分治法和动态规划的区别：
    - 分治法（Divide & Conquer）：
        - 分解：将原问题分解为一系列与原问题相似的子问题
    - 解决：递归解决各子问题
    - 合并：将子问题的结果合并成原问题的解
    - 动态规划（Dynamic Programming）：
        - 适用于子问题包含公共子子问题的情况，对每个子子问题只求解一次，避免重复运算
    - 步骤：
        - 描述最优解的结构
            - 递归定义最优解的值
            - 自底向上计算最优解的值

- 数据结构底层实现：
    - vector：数组
        - 快速随机访问
    - list：双向链表
        - 快速增删
    - map/set：红黑树
        - 占用内存较小，有序，访问稍慢
    - unordered_map：哈希表
        - 占用内存较大，但访问较快
    - priority_queue：Max-Heap（以vector实现）
    - deque：索引数组+分段数组

        - 支持双端增删和随机访问，但没有vector快
    - stack/queue：list或deque + 封闭头部
        - 不用vector因为扩容耗时（拷贝）

- 智能指针：
    - shared_ptr：
        - 多个指针指向相同对象
        - 使用引用计数，每次拷贝+1，析构-1，减为0时自动delete
        - 不能用同一指针初始化多个shared_ptr，否则会重复释放
    - unique_ptr：
        - 指针同时只能被一个智能指针拥有，不可赋值或拷贝
        - 可通过move移交所有权，原unique_ptr指向nullptr
        - 析构函数会自动delete该指针指向的内存
    - weak_ptr：
        - 用于观测资源的使用情况，不具有指针的功能
        - 可获取一个与观测对象等价的shared_ptr，从而操作资源

- SQL索引：
    - 定义：排好序的快速查找数据结构
    - 目的：提高查找效率，高效获取数据
    - 缺点：
        - 占据较大内存
        - 更新表时还要更新索引

- static和extern：
    - 静态局部变量：
        - 存在于静态存储区
        - 在函数内定义，生存周期为整个源程序，但作用域仍在该函数内
        - 在该函数外，该变量虽然存在但不能使用，在下一次该函数调用中可使用
        - 允许赋初值，若未赋初值则自动赋0
    - 静态全局变量：
        - 只在定义该变量的源文件内有效
        - 与非静态全局变量相同，都在静态存储区
        - 非静态全局变量可被其他源文件使用
        - 允许赋初值，若未赋初值则自动赋0
    - static函数：

        - 静态函数（内部函数）：
            - 函数类型前加一个static
            - 只能被本文件内的函数调用
    - extern变量：
        - extern "C" + 函数定义：
            - 取消mangling，按C语言规则翻译函数名
        - 修饰变量或函数时，声明作用范围为全部源文件，但不是定义，只是用其他文件内的定义
        - 文件A中定义extern后，文件B只需包含A的头文件即可，虽然编译时找不到但不会报错，链接时会从A生成的目标代码中找到该变量/函数

- 服务器多核软件开发：

- 编译和链接时找不到XX文件的原因：

- 信号与中断的区别：
    - 相似点：
        - 异步通信
        - 暂停正在执行的程序，执行相应的处理程序
        - 处理完毕后返回原断点
        - 可被屏蔽
    - 区别：
        - 中断有优先级，信号平等
        - 信号处理程序为用户态，中断处理程序为核心态
        - 中断响应及时，信号响应延迟较大

- 编译性语言和解释性语言：
    - 编译性语言：
        - C/C++
        - 程序执行前需要编译为机器语言的文件
        - 之后再运行时无需翻译，直接使用编译结果
        - 程序执行效率高
        - 不同操作系统间不兼容
        - 一般用于开发操作系统、大型应用程序、数据库系统等
    - 解释性语言：
        - Java，Python
        - 程序执行时才翻译，每执行一次翻译一次，效率较低
        - 兼容性好，但需要解释器，占用空间

        - 一般用于网页开发等，对效率要求不如可移植性高

- Linux进程内存空间布局：

- RTTI：(Runtime Type Identification的缩写，意思是运行时类型识别)
    - C++通过以下的两个操作提供RTTI：
        - typeid：
        - 返回表达式或类型名的实际类型
            - dynamic_cast：
                - 将基类指针/引用转换为子类指针/引用

- HTTPS更安全的原因：
    - HTTPS使用了RSA加密
    - Bob生成密钥对，并将公钥交给可信任的机构
    - 机构认证后用自己的密钥将Bob的公钥加密并发送给Alice
    - Alice用机构的公钥解密后获取Bob的公钥
    - Alice用公钥加密信息并发送给Bob
    - Bob接收到消息后用密钥解密

非对称加密算法用于在握手过程中加密生成的密码
对称加密算法用于对真正传输的数据进行加密
而HASH算法用于验证数据的完整性。

客户端有公钥，服务器有私钥，客户端用公钥对`对称密钥`进行加密，将加密后的对称密钥发送给服务器，服务器用私钥对其进行解密，所以客户端和服务器可用对称密钥来进行通信。公钥和私钥是用来加密密钥，而对称密钥是用来加密数据，分别利用了两者的优点。

1.浏览器将自己支持的一套加密规则发送给网站，如RSA加密算法，DES对称加密算法，SHA1摘要算法
2.网站从中选出一组加密算法与HASH算法，并将自己的身份信息以证书的形式发回给浏览器。证书里面包含了网站地址，加密公钥，以及证书的颁发机构等信息（证书中的私钥只能用于服务器端进行解密，在握手的整个过程中，都用到了证书中的公钥和浏览器发送给服务器的随机密码以及对称加密算法）

3.获得网站证书之后浏览器要做以下工作：
a) 验证证书的合法性（颁发证书的机构是否合法，证书中包含的网站地址是否与正在访问的地址一致等），如果证书受信任，则浏览器栏里面会显示一个小锁头，否则会给出证书不受信的提示。
b) 如果证书受信任，或者是用户接受了不受信的证书，浏览器会生成一串随机数的密码（这其实就是用于之后数据通信的对称加密算法的秘钥），并用证书中提供的公钥加密（对秘钥的加密采用非对称的方法）。
c) 使用约定好的HASH算法计算握手消息，并使用生成的随机数（对称算法的秘钥）对消息进行加密，最后将之前生成的被公钥加密的随机数密码，HASH摘要值（已经被对称算法加密）一起发送给服务器

4.网站接收浏览器发来的数据之后要做以下的操作：
a) 使用自己的私钥将信息解密并取出浏览器发送给服务器的随机密码（得到了对称加密算法的秘钥），使用密码解密浏览器发来的握手消息（用对称算法的秘钥解HASH摘要值），并验证HASH是否与浏览器发来的一致。
b) 使用随机密码加密一段握手消息，发送给浏览器。
5.浏览器解密并计算握手消息的HASH，如果与服务端发来的HASH一致，此时握手过程结束，之后所有的通信数据将由之前浏览器生成的随机密码并利用对称加密算法进行加密。

从上面的4个大的步骤可以看到，握手的整个过程使用到了数字证书、对称加密、HASH摘要，非对称加密算法。

- 传入参数很大或很多：
  - 参数很大：
    - 值传递成本太大
    - 选择传递指针或引用
  - 参数很多：
    - 选择封装到对象中传递
    - 寄存器会存一个指向内存空间的指针，这篇内存空间存着所有的传参

- const：
  - 修饰普通变量：
    - 该变量初始化后不可修改
  - 修饰指针变量：
    - const * p：p所指的变量不可修改
    - * const p：p不可指向其他变量

- n个数中获取最大的k个数：
  - 分治法：
    - 将n个数分为多组
    - 从每组中获取前k个最大的数
    - 合并任意数量的组获得新的多个组
    - 持续此过程直到只剩一组
  - 哈希法：
    - 先对每个值进行哈希映射，从而去重复
    - 再找前k大的数
    - 重复率较高时效果较好

- Heap操作：
  - 要点：保证操作前后均为完全二叉树
  - 插入：
    - 在最后一行最右方插入
    - 插入结点上浮至正确位置
  - 删除：
    - 根结点与最后一行最右方结点互换
    - 删除互换后的叶子结点（原根结点）
    - 根结点下沉至正确位置

- 用户态切换至内核态：
  - 系统调用
  - 异常
  - 中断

- 库函数和系统调用：
  - 库函数：
    - 通常用于应用程序对一般文件的访问
    - 与系统无关，可移植性好
    - 在用户态下运行
  - 系统调用：
    - 通常用于对底层文件的访问
    - 操作系统相关，可移植性差
    - 有着从用户空间切换到内核空间的开销

- GDB调试多进程：
  - follow-fork-mode
    - set follow-fork-mode child
    - fork后继续调试子进程
  - attach
    - attach <pid>
    - 附着于某进程
    - 在程序中加入一段代码，某条件成立时睡眠，attach后改变该条件，使其开始执行
  - GDB wrapper
    - 适用于fork后紧接exec的情况
    - 以GDB调用待执行代码作为一个整体被exec执行

- 服务器如何提高并发量：
  - 多线程多核编程，取消CPU瓶颈
  - 多路复用IO，取消IO阻塞瓶颈
  - 采用异步信号机制，减少等待时间
  - 将通信量较大的两个server放在同一主机，使用共享内存取代socket

- TCP短连接和长连接：
  - 短连接：
    - 每次收发信息需要重新连接
  - 长连接：
    - client先与server建立连接且不断开，之后进行信息的收发
    - 心跳保活：
      - 某端每隔一段时间发送自定义指令，确认双方存活
      - HTTP：client可发送Connection:Keep-Alive请求将连接保持在打开状态

- 高并发数据库：
  - 分库：

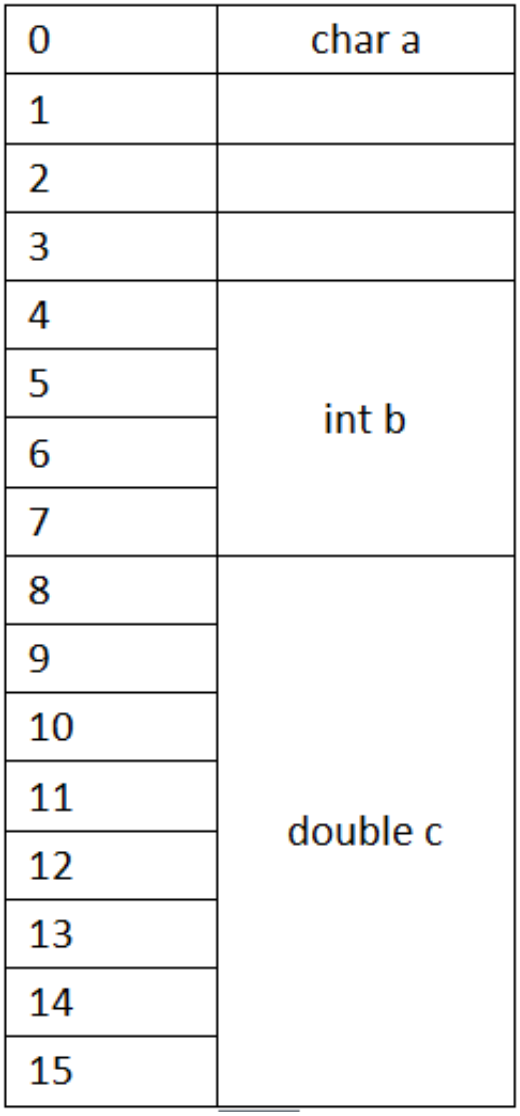
    垂直拆分或者水平拆分
- 分表：
  - e.g. 通过uid将一个DB分为10个库，每个库中含有8个子DB，当访问uid = 9527的资源时，通过如下计算定位DB：
    - 库编号 = (uid / 10) % 8 + 1
    - 表编号 = uid % 10
  - 订单ID：
    - 结构：版本号 + 库编号 + 表编号 + 时间戳 + 机器号 + 自增序号
    - 版本号：目前没有用到，属于冗余数据
    - 时间戳：毫秒级
    - 机器号：每个订单服务器的唯一编号

- 三种多路复用：
  - select：
    - 遍历所有fd，如有ready则返回
    - 缺点：

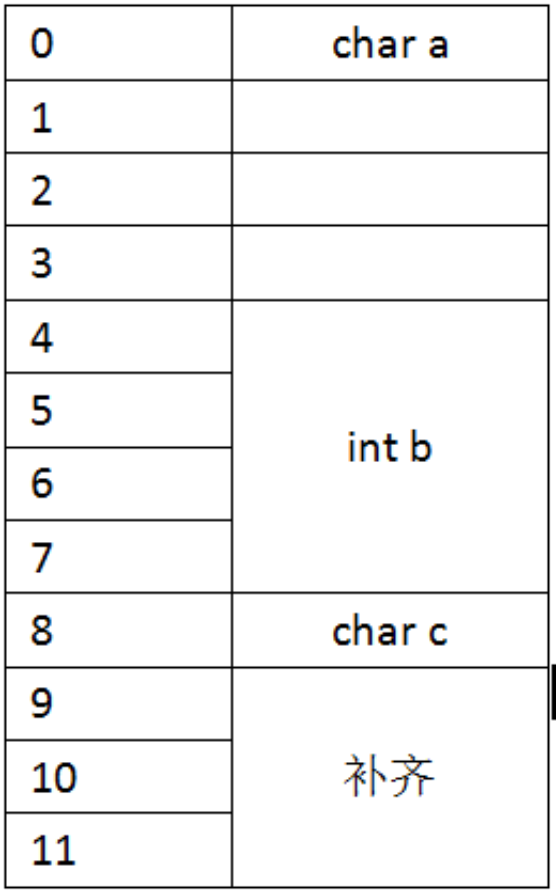
        - 拷贝和遍历fd开销大
        - 最多支持1024个fd
    - poll：
        - 与select类似，但通过pollfd来存储fd
        - 无最大数量限制，但仍需遍历，所以效率会随数量增长而下降
    - epoll：
        - 参考：https://zhuanlan.zhihu.com/p/64746509
        - 步骤：
            - epoll_create：创建一个eventpoll对象【句柄】
            - epoll_ctl：添加或删除需要监听的socket(红黑树，mmap)

              （当把事件添加进来的时候时候会完成关键的一步，那就是该事件都会与相应的设备（网卡）驱动程序建立回调关系，当相应的事件发生后，就会调用这个回调函数，该回调函数在内核中被称为：ep_poll_callback,**这个回调函数其实就所把这个事件添加到rdllist这个双向链表中**。一旦有事件发生，epoll就会将该事件添加到双向链表中。那么当我们调用epoll_wait时，epoll_wait只需要检查rdlist双向链表中是否有存在注册的事件，效率非常可观。这里也需要将发生了的事件复制到用户态内存中即可。）
            - 某一socket收到数据后，中断程序操作eventpoll对象，在其就绪列表(双向链表)中添加相应的socket引用
            - epoll_wait：若rdlist中已有socket引用则直接返回，否则进程阻塞
        - int epoll_create(int size) 中，size只是建议值，不是数量限制
        - 优点：
            - 无最大数量限制
            - 通过回调机制直接激活fd，无需返回后遍历，O(1)
            - 采用mmap, 每个fd仅拷贝一次，节省开销

- 结构体内存对齐&补齐：
    - 内存对齐：
        - 变量按顺序放入内存，每个元素放入时，认为内存以自己的大小划分
        - 若放入元素大小为4（如int），则会放入离结构体首部4n byte的位置，跳过被占用的4(n - 1) bytes



    - 空闲的内存在对齐规则下可以随意增加变量，内存总大小不变
- 内存补齐：
    - 内存总大小为所有元素中占内存最大元素的整数倍
    - 若不满足则会补齐



- Iterator删除元素：
  - 关联容器：
    - map，set，multimap，multiset
    - 删除当前iterator后，仅会使当前iterator失效，可以删除后**直接it++**
    - 原因：采用红黑树实现，插入和删除不会对其他结点造成影响
  - 序列容器：
    - vector，deque，list
    - 删除当前iterator后，后面的所有iterator都失效
    - erase可以返回下一个有效的iterator，需要it = vec.erase(it)
    - 原因：使用了连续分配的内存，删除一个元素会导致后面的元素向前移动

- 内存中堆和栈的区别：
  - 申请方式：
    - 栈：系统自动分配
    - 堆：用户手动申请
  - 申请空间：
    - 栈：剩余空间大于申请空间即可提供，但总空间较小
    - 堆：从空闲链表中提出某个合适的内存块，总空间大小受限于有效的虚拟内存

- 申请效率：
  - 栈：速度较快，但用户无法控制
  - 堆：速度较慢，但灵活且方便

- 虚函数：
  - C++中的虚函数的作用主要是实现了多态的机制。基类定义虚函数，子类可以重写该函数；在派生类中对基类定义的虚函数进行重写时，需要在派生类中声明该方法为虚方法。
  - 构造函数不能是虚函数：
    - vtable指针需要构造函数完成后才会生成，之后才能找到虚函数，构造函数如果是虚函数，就会陷入先有鸡还是先有蛋的问题
  - 继承时析构函数必须是虚函数：
    - 如果不是虚函数，子类对象成员将无法被正确释放，从而造成资源泄漏
- 指针和引用：
  1.指针和引用的定义和性质区别：
  - 指针：指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。如：

    int a=1;int *p=&a;

    int a=1;int &b=a;

    上面定义了一个整形变量和一个指针变量p，该指针变量指向a的存储单元，即p的值是a存储单元的地址。

    而下面2句定义了一个整形变量a和这个整形a的引用b，事实上a和b是同一个东西，在内存占有同一个存储单元。
  - 指针可以有多级，但是引用只能是一级（int **p；合法 而 int &&a是不合法的）
  - 指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；
  - 指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。
  - "sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小；
  - 指针和引用的自增(++)运算意义不一样；
- 头文件中的ifndef/define/endif有什么作用?
  - 这是C++预编译头文件保护符，保证即使文件被多次包含，头文件也只定义一次。
- ＃include<file.h> 与 ＃include "file.h"的区别?
  - 前者是从标准库路径寻找和引用file.h，而后者是从当前工作路径搜寻并引用file.h。
- 在C++ 程序中调用被C 编译器编译后的函数，为什么要加extern “C”?
  - extern 是一个关键字，可以放置于变量或函数前，以表示变量或函数的定义在别的文件中。 提示编译器遇到此变量或函数时，在其他模块中寻找其定义
    extern "C"是连接申明(linkage

declaration),被extern
"C"修饰的变量和函数是按照C语言方式编译和连接的。作为一种面向对象的语言，C++支持函数重载，而过程式语言C则不支持。函数被C++编译后在符号库中的名字与C语言的不同。例如，假设某个函数的原型为：void
foo(int x, int y);该函数被C编译器编译后在符号库中的名字为*foo，而C++编译器则会产生像*foo_int_int之类的名字。这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。
所以，可以用一句话概括extern “C”这个声明的真实目的:解决名字匹配问题，实现C++与C的混合编程。

- 共享内存的使用和实现原理

  两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间

  1、共享内存允许两个或更多进程共享一个给定的存储区，因为数据不需要再客户进程和服务进程之间复制。所以这是最快的一种ipc。

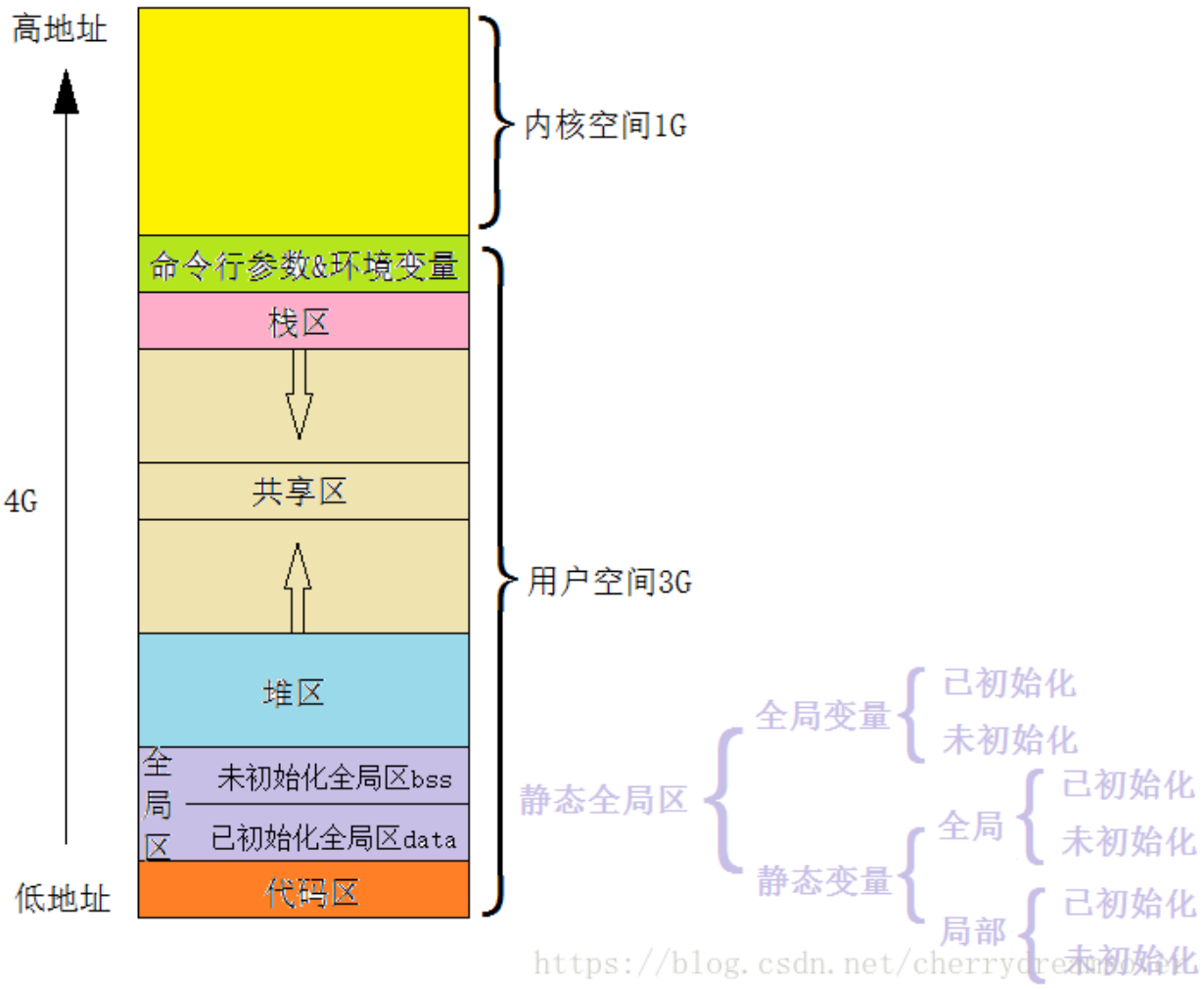
  2、使用共享内存时需要注意：多个进程对共享内存的同步访问。

  3、通常用信号量实现对共享内存的同步访问。

  共享内存段被映射进进程空间之后，存在于进程空间的什么位置？共享内存段最大限制是多少?

  共享内存段紧靠在栈之下，最大限制为32M

  分配一个新的共享内存块会创建新的内存页面。因为所有进程都希望共享对同一块内存的访问，只应由一个进程创建一块新的共享内存。再次分配一块已经存在的内存块不会创建新的页面，而只是会返回一个标识该内存块的标识符。一个进程如需使用这个共享内存块，则首先需要将它绑定到自己的地址空间中。这样会创建一个从进程本身虚拟地址到共享页面的映射关系。当对共享内存的使用结束之后，这个映射关系将被删除。当再也没有进程需要使用这个共享内存块的时候，必须有一个（且只能是一个）进程负责释放这个被共享的内存页面。
  所有共享内存块的大小都必须是系统页面大小的整数倍。系统页面大小指的是系统中单个内存页面包含的字节数。**在 Linux 系统中，内存页面大小是4KB，不过您仍然应该通过调用 getpagesize 获取这个值。**
  （1）进程通过调用**shmget**（Shared Memory GET，获取共享内存）来分配一个共享内存块。
  （2）要让一个进程获取对一块共享内存的访问，这个进程必须先调用 **shmat**（（（SHared Memory Attach，绑定到共享内存）。将 shmget 返回的共享内存标识符 SHMID 传递给这个函数作为第一个参数。该函数的第二个参数是一个指针，指向您希望用于映射该共享内存块的进程内存地址；如果您指定NULL则Linux会自动选择一个合适的地址用于映射。
  （3）调用 shmctl（"Shared Memory Control"，控制共享内存）函数会返回一个共享内存块的相关信息。要删除一个共享内存块，则应将 IPC_RMID 作为第二个参数，而将 NULL 作为第三个参数。当最后一个绑定该共享内存块的进程与其脱离时，该共享内存块将被删除。



- linux netstat tcpdump ipcs ipcrm
  - netstat 显示网络相关的信息，如网络连接，路由表(-r)，接口状态
  - tcpdump：对网络上的数据包进行截获的包分析工具
  - ipcs提供进程间通信方式的信息，包括共享内存，信号量，消息队列
  - ipcrm移除一个消息对象、或者共享内存、或者信号量

- **动态链接与静态链接**
  - 先来阐述一下DLL(Dynamic Linkable Library)的概念，你可以简单的把DLL看成一种仓库，它提供给你一些可以直接拿来用的变量、函数或类。
  - 静态链接库与动态链接库都是**共享代码的方式**，如果采用静态链接库，则无论你愿不愿意，lib中的指令都被直接包含在最终生成的EXE文件中了。但是若使用DLL，该DLL不必被包含在最终的EXE文件中，EXE文件执行时可以**“动态”地引用和卸载**这个与EXE独立的DLL文件。
  - 采用动态链接库的优点：（1）更加节省内存；（2）DLL文件与EXE文件独立，只要输出接口不变，更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性。
- 判断当前系统是大端还是小端

- int a = 1;
  假设有一个数字为 a = 1，分别画出 a 在大端和小端两种情况下的内存分布。

a.大端
00 00 00 01

b.小端
01 00 00 00

实际上，我们只需要判断低地址处是否为1。当前系统为大端存储模式时，其低地址处存储00，而当前系统为小端存储模式时，其低地址处存储01。那么这个问题可简化为只判断一个字节的内容是否为1，而当前数字为整型，这里对该数字强制类型转换为char型即可。

```
int JudgeSystem(void)
{
  int a = 1;
  char * p = (char *)&a;

  //如果是小端则返回1，如果是大端则返回0
  return *p;
}
```

- 列出常见的信号，信号怎么处理
  - 信号(signal)是很短的信息，可以被发送到一个进程或一组进程，发送给进程的唯一信息通常是一个数，以此来表示信号。
    信号的两个主要目的：让进程知道已经发生了一个特定的事件；强迫进程执行它自己代码中的信号处理程序。

2	SIGINT	terminate	来自键盘的中断(通常是Ctrl+c)
9	SIGKILL	terminate	强迫进程终止(kill -9)
5	SIGTERM	terminate	进程终止(不带参数时kill默认发送的信号)
19	SIGSTOP	stop	停止进程执行（ctrl+z）
18	SIGCONT	continue	如果进程已停止则恢复执行
11	SIGSEGV	Terminate and core	1. buffer overflow --- usually caused by a pointer reference out of range. 野指针 stack 2.overflow --- please keep in mind that the default stack size is 8192K. 栈溢出 illegal file 3.access --- file operations are forbidden on our judge system. 非法文件访问

|11|SIGSEGV|dump|无效的内存引用
|14|SIGALRM|terminate|实时定时器时钟
|20|SIGTSTP|stop|从tty发出停止进程(Ctrl+z)

当程序运行的过程中异常终止或崩溃，操作系统会将程序当时的内存状态记录下来，保存在一个文件中，这种行为就叫做**Core Dump**

- 进程对一个信号的应答
    1. 显示地忽略信号
    2. 执行与信号相关的缺省操作。内核预定义的缺省操作取决于信号的类型，有以下几种：
        - terminate：进程被终止（杀死）
        - dump：进程被终止（杀死）并且如果可能，创建包含进程执行上下文的核心转储文件，core dump文件。
        - ignore：信号被忽略
        - stop：进程被停止，把进程置为TASK_STOPPED状态
        - continue：如果进程被停止(TASK_STOPPED)，就把它置为TASK_RUNNING状态
    3. 通过调用相应的信号处理函数捕获信号
- 创建守护进程的步骤
    - 1、fork()创建子进程，父进程exit()退出；

      这是创建守护进程的第一步。由于守护进程是脱离控制终端的，完成这一步后就会在Shell终端里造成程序已经运行完毕的假象。之后的所有工作都在子进程中完成，而用户在Shell终端里则可以执行其他命令，从而在形式上做到了与控制终端的脱离，在后台工作。
      由于父进程先于子进程退出，子进程就变为孤儿进程，并由 init 进程作为其父进程收养。
    - 2、在子进程调用setsid()创建新会话；

      在调用了 fork() 函数后，子进程全盘拷贝了父进程的会话期、进程组、控制终端等，虽然父进程退出了，但会话期、进程组、控制终端等并没有改变。这还不是真正意义上的独立开来，而 setsid() 函数能够使进程完全独立出来。
      setsid()创建一个新会话，调用进程担任新会话的首进程，其作用有：
      使当前进程脱离原会话的控制
      使当前进程脱离原进程组的控制
      使当前进程脱离原控制终端的控制
    - 3、再次 fork() 一个子进程，父进程exit退出；

      现在，进程已经成为无终端的会话组长，但它可以重新申请打开一个控制终端，可以通过 fork() 一个子进程，该子进程不是会话首进程，该进程将不能重新打开控制终端。退出父进程。

      也就是说通过再次创建子进程结束当前进程，使进程不再是会话首进程来禁止进程重新打开控制终端。
    - 4、在子进程中调用chdir()让根目录“/”成为子进程的工作目录；

这一步也是必要的步骤。使用fork创建的子进程继承了父进程的当前工作目录。由于在进程运行中，当前目录所在的文件系统（如“/mnt/usb”）是不能卸载的，这对以后的使用会造成诸多的麻烦（比如系统由于某种原因要进入单用户模式）。因此，通常的做法是让"/"作为守护进程的当前工作目录，这样就可以避免上述的问题，当然，如有特殊需要，也可以把当前工作目录换成其他的路径，如/tmp。改变工作目录的常见函数是chdir。(避免原父进程当前目录带来的一些麻烦)

- 5、在子进程中调用umask()重设文件权限掩码为0；

  文件权限掩码是指屏蔽掉文件权限中的对应位。比如，有个文件权限掩码是050，它就屏蔽了文件组拥有者的可读与可执行权限（就是说可读可执行权限均变为7）。由于使用fork函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多的麻烦。因此把文件权限掩码重设为0即清除掩码（权限为777），这样可以大大增强该守护进程的灵活性。通常的使用方法为umask(0)。(相当于把权限开发)

- 6、在子进程中close()不需要的文件描述符；

  同文件权限码一样，用fork函数新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法卸下。其实在上面的第二步之后，守护进程已经与所属的控制终端失去了联系。因此从终端输入的字符不可能达到守护进程，守护进程中用常规方法（如printf）输出的字符也不可能在终端上显示出来。所以，文件描述符为0、1和2 的3个文件（常说的输入、输出和报错）已经失去了存在的价值，也应被关闭。（关闭失去价值的输入、输出、报错等对应的文件描述符）

- 7、守护进程退出处理

  当用户需要外部停止守护进程运行时，往往会使用 kill 命令停止该守护进程。所以，守护进程中需要编码来实现 kill 发出的signal信号处理，达到进程的正常退出。

- Linux任务调度机制

  - 在Linux中，每一个CPU都会有一个队列来存储处于TASK_RUNNING状态的任务，任务调度就是从这些队列中取出优先级最高的任务作为下一个放入CPU执行的任务。

    任务的调度需要进过两个过程：选择算法和上下文切换

    上下文切换
    从一个进程的上下文切换到另一个进程的上下文，因为其发生频率很高，所以通常都是调度器效率高低的关键。schedule()函数中调用了switch_to宏，这个宏实现了进程之间的真正切换，其代码存放于include/i386/system.h。switch_to宏是用嵌入式汇编写成的，较难理解。由switch_to()实现，而它的代码段在schedule()过程中调用，以一个宏实现。switch_to()函数正常返回，栈上的返回地址是新进程的task_struct::thread::eip，即新进程上一次被挂起时设置的继续运行的位置（上一次执行switch_to()时的标号”1:”位置）。至此转入新进程的上下文中运行。这其中涉及到wakeup，sleepon等函数来对进程进行睡眠与唤醒操作。

    选择算法
    Linux schedule()函数将遍历就绪队列中的所有进程，调用goodness()函数计算每一个进程的权值weight，从中选择权值最大的进程投入运行。Linux的调度器主要实现在schedule()函数中。

  - 调度步骤：

    Schedule函数工作流程如下：

    （1）选择下一个要运行的进程（pick_ next_ task）

（2） 进程上下文切换

- 上下文切换(有时也称做进程切换或任务切换)是指CPU从一个进程或线程切换到另一个进程或线程

  稍微详细描述一下，上下文切换可以认为是内核（操作系统的核心）在 CPU 上对于进程（包括线程）进行以下的活动：

  1. 挂起一个进程，将这个进程在 CPU 中的状态（上下文）存储于内存中的某处，
  2. 在内存中检索下一个进程的上下文并将其在 CPU 的寄存器中恢复
  3. 跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程

- tcp头部格式：

  1. 源端口号,16位，发送方的端口号。
  2. 目标端口号，16位，发送方的目标端口号。
  3. 32为序列号，sequence number，保证网络传输数据的顺序性。
  4. 32位确认号，acknowledgment number，用来确认确实有收到相关封包，内容表示期望收到下一个报文的序列号，用来解决丢包的问题。
  5. 头部大小，4位，偏移量：最大值为0x0F，即15,

  单位为32位(bit)，单位也就是4个字节，给出头部占32bit的数目。没有任何选项字段的TCP头部长度为20字节；最多可以有60(15*4)字节的TCP头部。

  6. Reserved 4位 ，预留字段，都为0
  7. TCP标志位

     （1）CWR：Congestion window reduced，拥塞窗口减少。拥塞窗口减少标志被发送主机设置，用来表明它接收到了设置ECE标志的TCP包。拥塞窗口是被TCP维护的一个内部变量，用来管理发送窗口大小。

     （2）ECN-Echo：显式拥塞提醒回应。当一个IP包的ECN域被路由器设置为11时，接收端而非发送端被通知路径上发生了拥塞。ECN使用TCP头部来告知发送端网络正在经历拥塞，并且告知接收端发送段已经受到了接收端发来的拥塞通告，已经降低了发送速率。

     （3）URG：为1时，紧急指针（urgent pointer）有效，配合紧急指针使用

     （4）ACK：为1时，确认号有效

     （5）PSH: 为1时，接收方应该尽快将这个报文段交给应用层

     (6)RST：为1时，释放连接，重连。

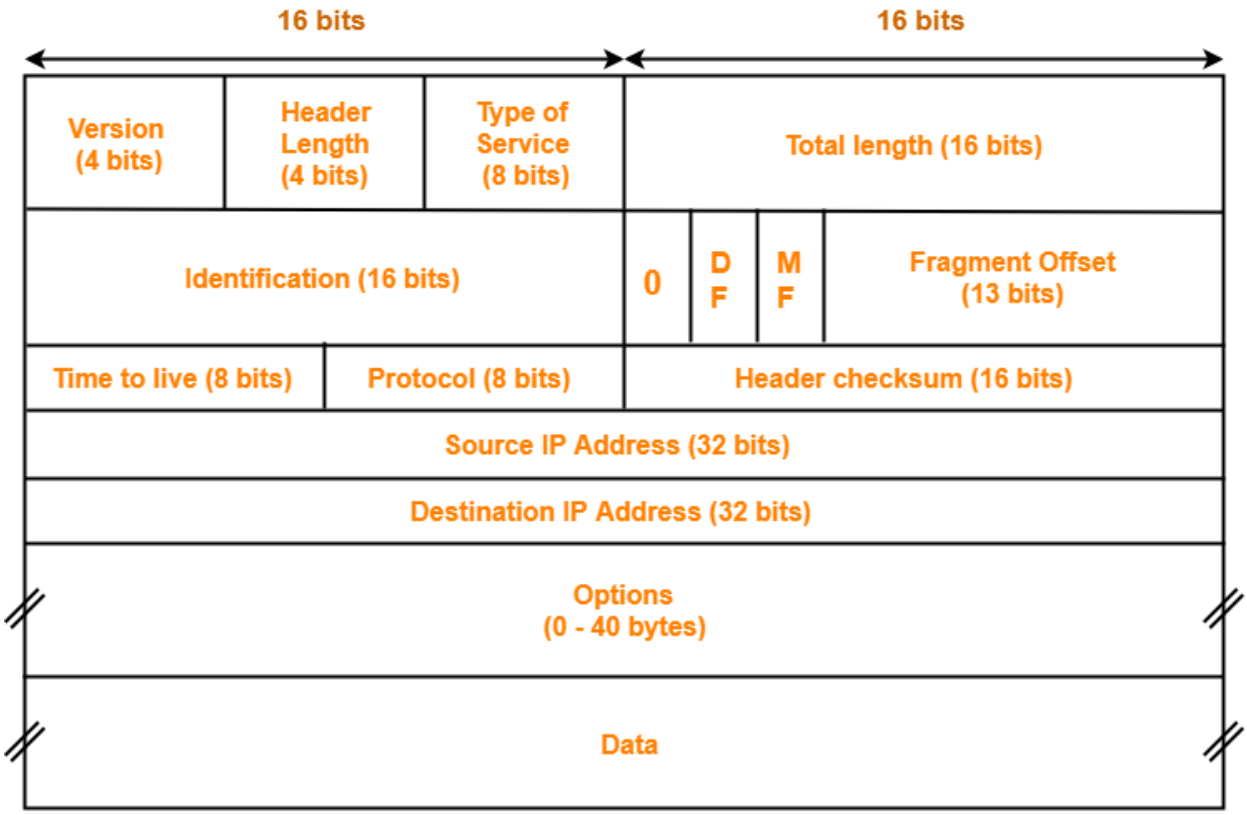
     (7)SYN：为1时，发起一个连接。

     (8)FIN：为1时，关闭一个连接。

  8. 16位窗口大小：占16bit。此字段用来进行流量控制，主要用于解决流控拥塞的问题。单位为字节数，这个值是本机期望一次接收的字节数。

9. 16位校验值: 占16bit。对整个TCP报文段，即TCP头部和TCP数据进行校验和计算，并由目标端进行验证。
10. 16位紧急指针:占16bit。它是一个偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。
11. 32位Tcp选项:一般包含在三次握手中。

- IP 头



IPv4 Header

- **面向对象编程的三大特性**
  - 封装

    封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。
  - 继承

    继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。
    要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。
  - 多态

    多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：**允许将子类类型的指针赋值给父类类型的指针。实现多态，有两种方式，覆盖和重载。覆盖和重载的区别在于，覆盖在运行时决定，重载是在编译时决定。**
- **面向对象的SOLID原则（SOLID）**
  - 单一职责原则

就一个类而言，应该仅有一个引起它变化的原因
如果一个类承担的职责过多，这些职责耦合在一起，一个职责的变化可能削弱或者一直这个类完成其他职责的能力，设计脆弱，变化会导致意向不到的破坏。
软件设计要发现职责，并把那些职责相互分离，判断是否需要分离的依据：如果你能想到多余一个的动机去改变一个类，那么这个类就有多于一个的职责

- 开放封闭原则

类，模板，函数等，应该可以扩展，但是不可修改
两个特征：对于扩展是开放的，对于更改是封闭的
面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码
开放封闭原则是面型对象设计的核心所在，遵循这个原则可以带来面向对象技术所声称的巨大好处，可维护、可扩展、可复用、灵活性好，开发人员应该仅对程序中呈现出频繁变化的那些部分作出抽象，但不要对程序中每个部分都刻意的进行抽象

- 里氏替换原则

子类必须能够替换掉它们的父类型
只有当子类可以替换掉父类，软件单位的功能不受到影响时，父类才能真正的被复用，而子类也能够在父类的基础上增加行为。

- 接口分离原则

在类的结构设计上，每一个类都应该尽量降低成员的访问权限
该法则在适配器模式、解释模式等中有强烈的体现
强调类之间的松耦合，类之间的耦合越弱，越有利于复用

- 依赖倒置原则

高层模块不应该依赖底层模块，两个都应该依赖抽象。
抽象不应该依赖细节，细节应该依赖抽象

- 线程安全
- 线程安全是程式设计中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。
- 多进程、多线程和I/O多路复用三种Web服务器模型

1. 多进程模型的优缺点
   1）优点：

> 1）每个进程互相独立，不影响主程序的稳定性，子进程崩溃没关系；
> 2）通过增加CPU，就可以容易扩充性能；
> 3）可以尽量减少线程加锁/解锁的影响，极大提高性能，就算是线程运行的模块算法效率低也没关系；
> 4）每个子进程都有2GB地址空间和相关资源，总体能够达到的性能上限非常大。

   2）缺点：

1）逻辑控制复杂，需要和主程序交互；
2）需要跨进程边界，如果有大数据量传送，就不太好，适合小数据量传送、密集运算 ；
3）多进程调度开销比较大。

2. 多线程模型的优缺点
   1）优点：

1）无需跨进程边界；
2）程序逻辑和控制方式简单；
3）所有线程可以直接共享内存和变量等；
4）线程方式消耗的总资源比进程方式好；

（2）缺点：

1）每个线程与主程序共用地址空间，受限于2GB地址空间；
2）线程之间的同步和加锁控制比较麻烦；
3）一个线程的崩溃可能影响到整个程序的稳定性；
4）到达一定的线程数程度后，即使再增加CPU也无法提高性能；
5）线程能够提高的总性能有限，而且线程多了之后，线程本身的调度也是一个麻烦事儿，需要消耗较多的CPU 。

（3）如何减少上下文切换

线程池的关键点是：

1）尽量减少线程切换和管理的开支；
2）最大化利用cpu。
对于第一点，要求线程数尽量少，这样可以减少线程切换和管理的开支；
对于第二点，要求尽量多的线程，以保证CPU资源最大化的利用。 所以：

对于任务耗时短的情况，要求线程尽量少，如果线程太多，
有可能出现线程切换和管理的时间，大于任务执行的时间，那效率就低了；
对于耗时长的任务，要分是cpu任务，还是io等类型的任务。如果是cpu类型的任务，
线程数不宜太多；但是如果是io类型的任务，线程多一些更好，可以更充分利用cpu。

所以高并发，低耗时的情况：建议少线程，只要满足并发即可。例如并发100，线程池可能设置为10就可以。低并发，高耗时的情况：建议多线程，保证有空闲线程，接受新的任务。例如并发10，线程池可能就要设置为20。高并发高耗时：1要分析任务类型，2增加排队，3、加大线程数。

3. I/O多路复用的优缺点
   （1）优点：

1）相比于多线程和多进程，I/O多路复用是在单一进程的上下文中的，
当有多个并发连接请求时，多线程或者多进程模型需要为每个连接创建一个线程或者进程，
而这些进程或者线程中大部分是被阻塞起来的。由于CPU的核数一般都不大，
比如4个核要跑1000个线程，那么每个线程的时间槽非常短，
而线程切换非常频繁。这样是有问题的。而使用I/O多路复用时，
处理多个连接只需要1个线程监控就绪状态，
对就绪的每个连接开一个线程处理（由线程池支持）就可以了，这样需要的线程数大大减少，
减少了内存开销和上下文切换的CPU开销。
2）整个过程只在调用select、poll、epoll这些调用的时候才会阻塞，
收发客户消息是不会阻塞的，整个进程或者线程就被充分利用起来，这就是事件驱动。

（2）缺点：

单线程模型不能有阻塞，一旦发生任何阻塞（包括计算机计算延迟）
都会使得这个模型不如多线程。另外，单线程模型不能很好的利用多核cpu。

- 右值引用 (Rvalue Referene)
  - 是 C++ 新标准 (C++11, 11 代表 2011 年) 中引入的新特性 , 它实现了转移语义 (Move Sementics) 和完美转发(Perfect Forwarding)。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. 能够更简洁明确地定义泛型函数。
- C++ 中的 inline 用法
  - 在 c/c++ 中，为了解决一些频繁调用的小函数大量消耗栈空间（栈内存）的问题，特别的引入了 inline 修饰符，表示为内联函数。
- inline使用限制：
  inline 的使用是有所限制的，inline 只适合涵数体内代码简单的涵数使用，不能包含复杂的结构控制语句例如 while、switch，并且不能内联函数本身不能是直接递归函数（即，自己内部还调用自己的函数）。
  - inline仅是一个对编译器的建议：
    inline 函数仅仅是一个对编译器的建议，所以最后能否真正内联，看编译器的意思，它如果认为函数不复杂，能在调用点展开，就会真正内联，并不是说声明了内联就会内联，声明内联只是一个建议而已。
  - 建议 inline 函数的定义放在头文件中：
    其次，因为内联函数要在调用点展开，所以编译器必须随处可见内联函数的定义，要不然就成了非内联函数的调用了。所以，这要求每个调用了内联函数的文件都出现了该内联函数的定义。
    因此，将内联函数的定义放在头文件里实现是合适的，省却你为每个文件实现一次的麻烦。
    声明跟定义要一致：如果在每个文件里都实现一次该内联函数的话，那么，最好保证每个定义都是一样的，否则，将会引起未定义的行为。如果不是每个文件里的定义都一样，那么，编译器展开的是哪一个，那要看具体的编译器而定。所以，最好将内联函数定义放在头文件中。
  - 类中的成员函数与inline：
    定义在类中的成员函数默认都是内联的，如果在类定义时就在类内给出函数定义，那当然最好。
    如果在类中未给出成员函数定义，而又想内联该函数的话，那在类外要加上 inline，否则就认为

不是内联的。

- inline 是一种"用于实现的关键字"
  关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。
  - 慎用 inline:
    内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？如果所有的函数都是内联函数，还用得着"内联"这个关键字吗？
    内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

- 浏览器访问网页过程中发生了什么?

  第一步，解析域名，找到主机IP

  （1）浏览器会缓存DNS一段时间，一般2-30分钟不等。如果有缓存，直接返回IP，否则下一步。

  （2）缓存中无法找到IP，浏览器会进行一个系统调用，查询hosts文件。如果找到，直接返回IP，否则下一步。（在计算机本地目录etc下有一个hosts文件，hosts文件中保存有域名与IP的对应解析，通常也可以修改hosts科学上网或破解软件。）

  （3）进行了（1）（2）本地查询无果，只能借助于网络。路由器一般都会有自己的DNS缓存，ISP服务商DNS缓存，这时一般都能够得到相应的IP。如果还是无果，只能借助于DNS递归解析了。

  （4）这时，ISP的DNS服务器就会开始从根域名服务器开始递归搜索，从.com顶级域名服务器，到baidu的域名服务器。

  到这里，浏览器就获得了IP。在DNS解析过程中，常常会解析出不同的IP。比如，电信的是一个IP，网通的是另一个IP。这是采取了智能DNS的结果，降低运营商间访问延时，在多个运营商设置主机房，就近访问主机。电信用户返回电信主机IP，网通用户返回网通主机IP。当然，劫持DNS，也可以屏蔽掉一部分网点的访问，某防火长城也加入了这一特性。

第二步，浏览器与网站建立TCP连接

浏览器利用IP直接与网站主机通信。浏览器发出TCP（SYN标志位为1）连接请求，主机返回TCP（SYN，ACK标志位均为1）应答报文，浏览器收到应答报文发现ACK标志位为1，表示连接请求确认。浏览器返回TCP（）确认报文，主机收到确认报文，三次握手，TCP链接建立完成。

第三步，浏览器发起GET请求

浏览器向主机发起一个HTTP-GET方法报文请求。请求中包含访问的URL，也就是http://www.baidu.com/，还有User-Agent用户浏览器操作系统信息，编码等。值得一提的是Accep-Encoding和Cookies项。Accept-Encoding一般采用gzip，压缩之后传输html文件。Cookies如果是首次访问，会提示服务器建立用户缓存信息，如果不是，可以利用Cookies对应键值，找到相应缓存，缓存里面存放着用户名，密码和一些用户设置项。

第四步，显示页面或返回其他

返回状态码200 OK，表示服务器可以相应请求，返回报文，由于在报头中Content-type为“text/html”，浏览器以HTML形式呈现，而不是下载文件。
但是，对于大型网站存在多个主机站点，往往不会直接返回请求页面，而是重定向。返回的状态码就不是200 OK，而是301,302以3开头的重定向码，浏览器在获取了重定向响应后，在响应报文中Location项找到重定向地址，浏览器重新第一步访问即可。

- sh .sh source .sh ./a.sh区别

  ./a.sh，sh ./a.sh和sh a.sh是一样的， 实际上是启了一个子shell来执行a.sh, 所以可以看到PPID of this script: 21657

  source ./a.sh ,source a.sh 和. ./a.sh是一样的， 都是在当前shell中执行脚本， 请看进程号

  ../a.sh是万万要不得的，两个点之间没有空格

  最后要说明的两点是：

  1. 用sh和source去执行时， 不要求a.sh有可执行权限， 但单独./a.sh这样去搞时，需要可执行权限

  2. 大家在开发项目时，经常需要设置环境变量， 当然是用source啊， 确保在当前shell生效

信号 中断 异常

- 中 断（也称硬件中断）
  - 定义：中断是由其他硬件设备依照CPU时钟周期信号随机产生的。
  - 分类： 可屏蔽中断 非可屏蔽中断
  - 来源：间隔定时器和I/O
  - 中断处理

    设备产生中断

    PIC（可编程中断控制器）会产生一个对应的中断向量

    和中断向量表中的每一个中断向量进行比较，转到对应的中断处理程序

    中断处理程序进行保存现场，做相关处理，恢复现场

    内核调度，返回用户进程
- 异 常（也称软件中断）
  - 定义：当指令执行时由CPU控制单元产生的。
  - 分类： 处理器探测到的异常 编程异常(也称软中断)

    int指令 内核必须处理的异常(例如：缺页和内核服务的请求-int)

- 异常处理

  当发生异常时，CPU控制单元产生一个硬件出错码。

  CPU根据该中断码找到中断向量表内的对应向量，根据该向量转到中断处理程序。

  中断处理程序处理完之后向当前进程发送一个SIG***信号。

  若进程定义了相应的信号处理程序则转移到相应的程序执行，若没有，则执行内核定义的操作。

- 信号
  - 信号与中断的相似点

（1）采用了相同的异步通信方式；
（2）当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序；
（3）都在处理完毕后返回到原来的断点；
（4）对信号或中断都可进行屏蔽。

  - 信号与中断的区别

（1）中断有优先级，而信号没有优先级，所有的信号都是平等的；
（2）信号处理程序是在用户态下运行的，而中断处理程序是在核心态下运行；
（3）中断响应是及时的，而信号响应通常都有较大的时间延迟。

  - 信号机制

（1）发送信号。发送信号的程序用系统调用kill( )实现；
（2）预置对信号的处理方式。接收信号的程序用signal( )来实现对处理方式的预置；
（3）收受信号的进程按事先的规定完成对相应事件的处理。

  - 对信号的处理

当一个进程要进入或退出一个低优先级睡眠状态时，或一个进程即将从核心态返回用户态时，核心都要检查该进程是否已收到软中断。当进程处于核心态时，即使收到软中断也不予理睬；只有当它返回到用户态后，才处理软中断信号。对软中断信号的处理分三种情况进行：

（1）如果进程收到的软中断是一个已决定要忽略的信号（function=1），进程不做任何处理便立即返回；
（2）进程收到软中断后便退出（function=0）；
（3）执行用户设置的软中断处理程序。

- 拥塞控制，流量控制：
  - 拥塞控制：拥塞控制是作用于网络的，它是防止过多的数据注入到网络中，避免出现网络负载过大的情况；常用的方法就是：（1）慢开始、拥塞避免（2）快重传、快恢复。
  - 拥塞控制详细：https://zhuanlan.zhihu.com/p/37379780
  - 流量控制：流量控制是作用于接收者的，它是控制发送者的发送速度从而使接收者来得及接收，防止分组丢失的。
  - 什么是流量控制？流量控制的目的？

如果发送者发送数据过快，接收者来不及接收，那么就会有分组丢失。为了避免分组丢失，控制发送者的发送速度，使得接收者来得及接收，这就是流量控制。流量控制根本目的是防止分组丢失，它是构成TCP可靠性的一方面。

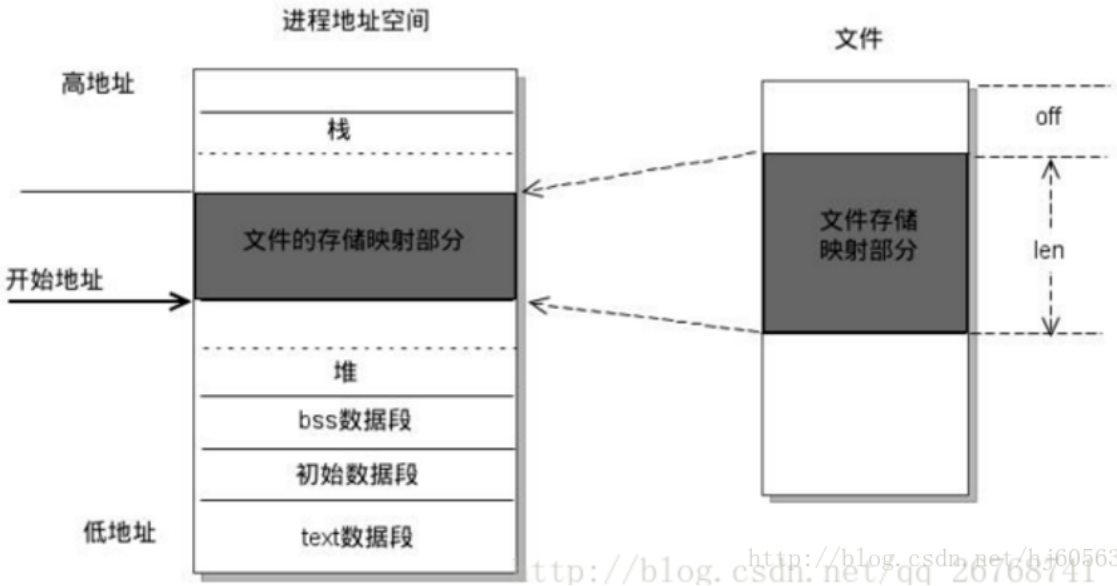
如何实现流量控制？

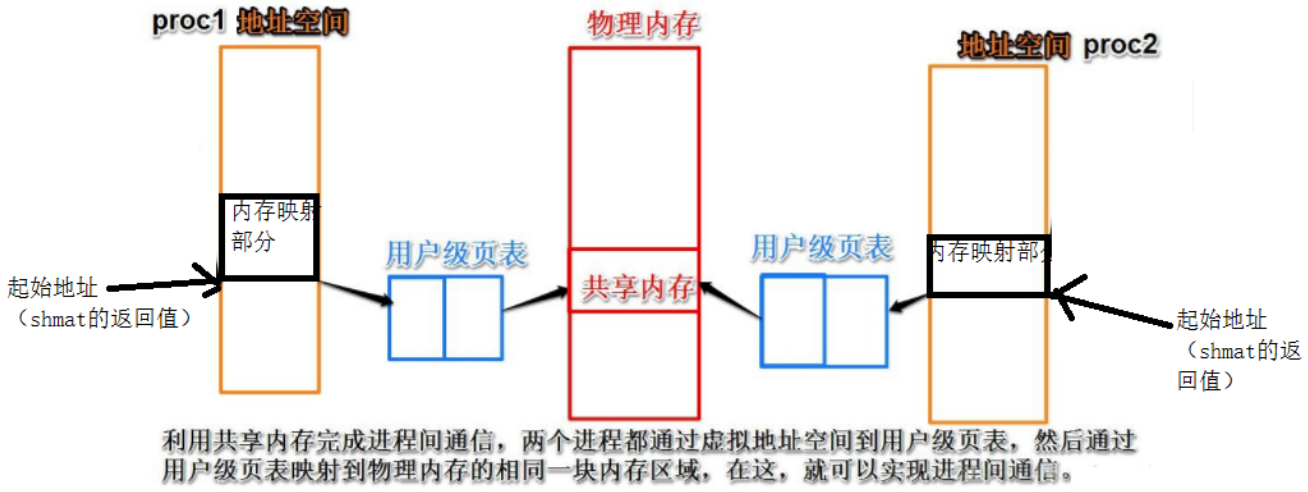
由滑动窗口协议（连续ARQ协议）实现。滑动窗口协议既保证了分组无差错、有序接收，也实现了流量控制。主要的方式就是接收方返回的 ACK 中会包含自己的接收窗口的大小，并且利用大小来控制发送方的数据发送。

流量控制引发的死锁？怎么避免死锁的发生？

当发送者收到了一个窗口为0的应答，发送者便停止发送，等待接收者的下一个应答。但是如果这个窗口不为0的应答在传输过程丢失，发送者一直等待下去，而接收者以为发送者已经收到该应答，等待接收新数据，这样双方就相互等待，从而产生死锁。
为了避免流量控制引发的死锁，TCP使用了持续计时器。每当发送者收到一个零窗口的应答后就启动该计时器。时间一到便主动发送报文询问接收者的窗口大小。若接收者仍然返回零窗口，则重置该计时器继续等待；若窗口不为0，则表示应答报文丢失了，此时重置发送窗口后开始发送，这样就避免了死锁的产生。

- 总结mmap和shm:
  1、mmap是在磁盘上建立一个文件，每个进程地址空间中开辟出一块空间进行映射。
  而对于shm而言，shm每个进程最终会映射到同一块物理内存。shm保存在物理内存，这样读写的速度要比磁盘要快，但是存储量不是特别大。
  2、相对于shm来说，mmap更加简单，调用更加方便，所以这也是大家都喜欢用的原因。
  3、另外mmap有一个好处是当机器重启，因为mmap把文件保存在磁盘上，这个文件还保存了操作系统同步的映像，所以mmap不会丢失，但是shmget就会丢失。





- 宏与函数

  宏相比函数而言的优势主要在于：

  1. 宏因为是文本替换，没有函数栈的维护代价
  2. 宏参数不带类型，可以做函数不能做的工作。

  解释一下，第一点 函数调用是利用函数栈帧来实现的，这个需要一定的系统资源。但是宏是直接文本替换，无函数调用过程。
  第二点是因为函数的参数列表要求每个参数必须带类型，当我们想实现把类型也当作参数传入时，函数是无法做到。例如下面的宏：

```
#define MALLOC(size,type) ((type*)malloc(sizeof(type) * (size)))
int * p =MALLOC(10,int);
```

- 数据库保证事务的ACID：
  - 一致性：数据库通过原子性，隔离性，持久性来保证一致性。同时在上层的代码里，代码要符合约束。
  - 原子性：利用innodb的undo log(回滚日志)，当事务回滚时能够撤销所有已经成功执行的sql语句，他需要记录你需要回滚的相关信息，当事务执行失败，可以利用回滚日志中的信息进行回滚
    - (1)当你delete一条数据的时候，就需要记录这条数据的信息，回滚的时候，insert这条旧数据
    - (2)当你update一条数据的时候，就需要记录之前的旧值，回滚的时候，根据旧值执行update操作
    - (3)当年insert一条数据的时候，就需要这条记录的主键，回滚的时候，根据主键执行delete操作
  - 持久性：利用innodb的redo log, mysql会将数据从磁盘加载到内存，在内存中对数据进行修改，再刷回磁盘。如果这时候死机，数据的更改将会丢失。如果直接每次在事务提交前，把页面刷回磁盘，效率很低，所以采用redo log,每当有事务提交时，会对redo log进行刷盘，如果主机掉电重启，会将redo log中的内容恢复到数据库中。

    - 隔离性：锁和MVCC机制。MVCC是行级锁的变种，他在很多情况下避免了加锁的操作。--->非阻塞的读操作，写操作也只锁定必要的行

- MVCC实现：
    - 通过保存数据在某个时间点的快照来实现的。InnoDB是通过在每行记录后面保存两个隐藏的列来实现。一列保存行的创建时间，一列保存行的删除时间。存储的不是真实的时间，而是系统版本号（递增）。事务开始时刻的**系统版本号**会作为**事务的版本号**，用来和查询到的每行记录的版本号作比较。
    - SELECT

      > InnoDB会查找版本小于当前事务版本的数据行，行的删除版本要么未定义，要么大于当前事务的版本号。这样可以确保事务读到的行，在事务之前未删除。

    - INSERT

      > InnoDB为新插入的一行保存当前系统版本号作为行版本号

    - DELETE

      > InnoDB为删除的每一行保存当前系统版本号作为删除标志

    - UPDATE

      > InnoDB把当前系统版本号作为当前行版本号，同时将当前版本号保存为该行的删除版本号

- 数据库并发控制：
    - 悲观锁（PCC）：先取🔐再访问，保证数据安全。
    - 乐观锁（OCC）：假设用户处理的并发事物之间不会产生相互影响，在事务提交以前，每个事务检查该事物读取数据后，有没有其他事务又修改了数据。如果有其他事务更新的话，正在提交的事务会进行回滚。一般实现乐观锁的方式是记录数据版本。
    - MVCC
- 解决哈希冲突
    - 开放地址法
        - 线性查找
        - 平方查找
        - 双哈希函数探测法：探测的步长值是同一关键字另外一个hash函数的值
    - rehash: 就是构造多个hash函数，当第一个发生冲突的时候调用第二个计算，直到不再有冲突