测试

第一部分 linux

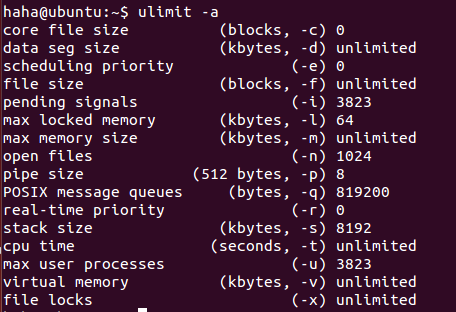
1. 查询当前系统的一些限制

单个程序能能打开的最大文件（最多的tcp连接）

ulimit –n

最大堆栈

ulimit -s



1. netstat 命令详解

–a 显示所有

–l 显示监听相关

-p 显示占用该端口的程序名称

-n 不显示别名

3.查询当前哪些程序使用了处于监听中

netstat –anp | grep TCP

netstat –lnp

4查询某个端口被那个程序占用

netstat –apn | grep 8080

lsof –l:8080

5.查询处于某种状态的端口号

netstat –an | grep LISTENING

5查询某个文件被哪些程序占用

fuser –u filename

fuser –u4 只查询ipv4套接字

ps –ef | grep filename

6查询某个pid所对应的进程名称

lsof –p pid

7查询某个进程的pid

ps –aux | grep name

pidof name

8查询某个进程打开哪些文件（socket）

lsof –c 进程名

lsof –p pid

9查询某个进程占用的内存

top –p pid

top –u user

pmap –d pid

10.找出系统的僵死程序，如何处理

可以使用top判断当前有无僵尸进程（top会显示当前系统的进程统计信息，包括一共有多少进程，处于各种状态的各有多少）

ps aux | grep Z

kill -9 pid

11**显示系统核心版本号、名称、机器类型等**

uname –a

**12查看cpu、内存信息**

**cat /proc/meminfo cat /proc/cpuinfo**

**13查看硬盘信息**

**df –hl**

**14前后台切换**

**程序后面加& 表示后台运行**

**crtl+Z 将当前程序丢入后台**

**fg %n 让后台运行的进程n到前台来**

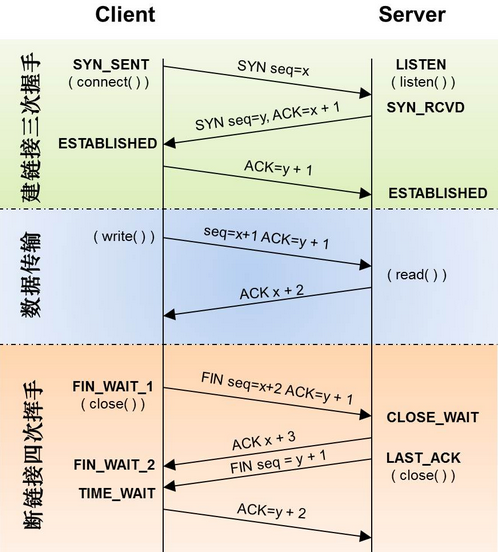
**bg %n 让进程n到后台去**

**15查看当前系统有多少用户在线**

**w或者who**

第二部分 计算机网络

1. 三次握手，四次挥手



1. **为什么么要三次握手**

**为了防止已失效的连接请求报文段突然又传送到了服务端，服务器端建立起无效的链接，一直等待而占用资源。**

**client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。**

1. TIME\_WAIT的作用

**TIME\_WAIT为2MSL，MSL(maximum segment lifetime) 这是一个IP数据包能在互联网上生存的最长时间，，一般为30秒**

 （1）可靠的关闭TCP连接

**主动关闭连接的一方（A）在收到对面发送的FIN，会发送最后一个ack，然后会进入TIME\_WAIT状态，再停留2个MSL时间(后有MSL的解释)，进入CLOSED状态。对方（B）如果丢失了这个ack报文会重新发送fin， 而如果A没有TIME\_WAIT直接进入CLOSED状态则无法响应此报文，B就会一直处于LAST\_ACK状态而无法正常关闭。**

（2）防止上一次连接中的包，迷路后重新出现，影响新连接（经过2MSL，上一次连接中所有的重复包都会消失）

**TCP分节可能由于路由器异常而“迷途”，在迷途期间，TCP发送端可能因确认超时而重发这个分节，迷途的分节在路由器修复后也会被送到最终目的地，这个迟到的迷途分节到达时可能会引起问题。在关闭“前一个连接”之后，马上又重新建立起一个相同的IP和端口之间的“新连接”，“前一个连接”的迷途重复分组在“前一个连接”终止后到达，而被“新连接”收到了。为了避免这个情况，TCP协议不允许处于TIME\_WAIT状态的连接启动一个新的可用连接，因为TIME\_WAIT状态持续2MSL，就可以保证当成功建立一个新TCP连接的时候，来自旧连接重复分组已经在网络中消逝。**

1. tcp，udp的区别

基于连接与无连接  
对系统资源的要求（TCP较多，UDP少）  
UDP程序结构较简单  
流模式与数据报模式   
TCP保证数据正确性，UDP可能丢包

TCP保证数据顺序，UDP不保证

1. 如何避免time\_wait状态占用资源

/etc/sysctl.conf

修改 time\_wait 时间net.ipv4.tcp\_fin\_timeout=30

开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭

net.ipv4.tcp\_tw\_reuse = 1

开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭

net.ipv4.tcp\_tw\_recycle = 1

1. 滑动窗口的作用

流量控制：通过调节滑动窗口和接受窗口的大小来实现

可靠传输：TCP的滑动窗口的可靠性也是建立在“确认重传”基础上的。

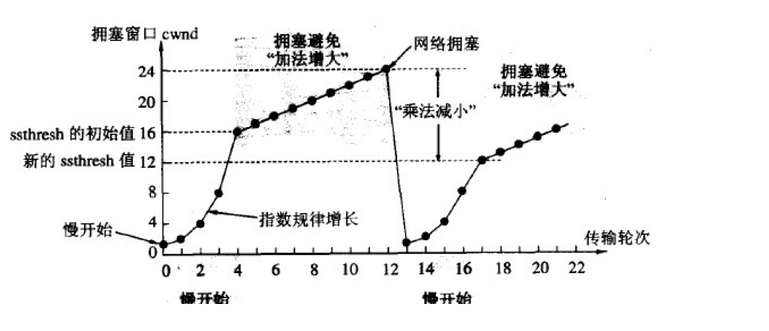
发送窗口只有收到对端对于本段发送窗口内字节的ACK确认，才会移动发送窗口的左边界。

接收窗口只有在前面所有的段都确认的情况下才会移动左边界。当在前面还有字节未接收但收到后面字节的情况下，窗口不会移动，并不对后续字节确认。以此确保对端会对这些数据重传

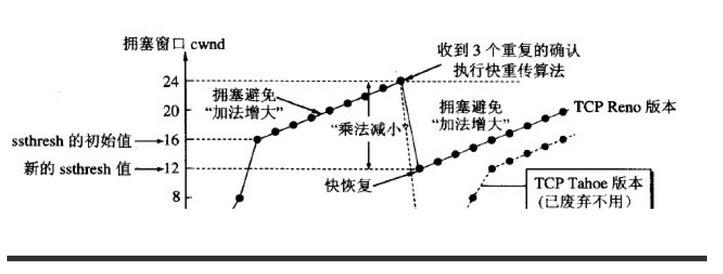
1. tcp流量控制与拥塞处理

流量控制通过滑动窗口机制来实现

慢启动（slow-start）、拥塞避免（congestion avoidance）、快速重传（fast retransmission）和快速恢复（fastrecover）

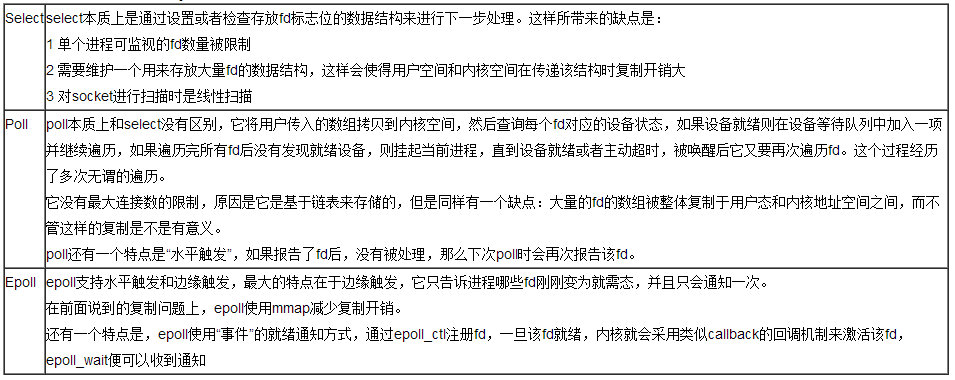


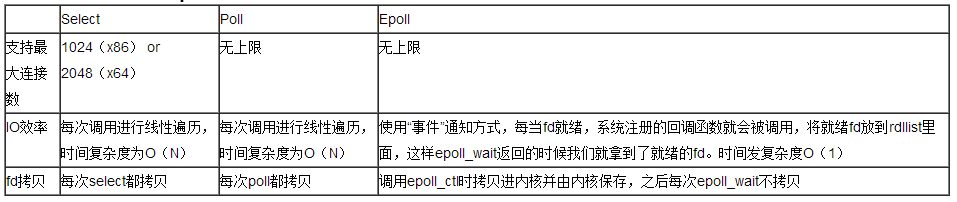
慢启动和拥塞避免



快速重传和快速恢复

1. I/O多路复用select和poll，epoll的区别





select 是采用内核轮询方式，每次调用都需要轮询 FD\_SET，默认最多可以接受 1024 个fd，可更改为更大，但是随着数量的增多，轮询周期的变长，性能会急剧下降；

poll 是 select 的改进版，将 FD\_SET 该造成由（ fd，监听事件类型，实际事件类型 ）为节点组成的链，解除了1024 的限制，其他并无大的区别，当 fd 多时，同样会造成效率下降；

epoll 将 轮询机制 改造为 事件触发机制，给每一个 fd 附上一个 callback，当监听事件发生时，就将 fd 链接到 就绪链表，调用 epoll\_wait 时，只用检查就绪链表就可以了，而不需要像 select 和 poll 一样进行轮询。

另外，select 和 poll 是将存有 fd 的结构或者数组在每次调用的时候都复制到内核态，然后调用完再复制回用户态，而无所谓是否有意义。epoll 使用内存映射，减去了这部分的data-copy操作。

再者，从触发方式上来看，select 和 poll 都只有 条件触发（也可以叫水平触发），epoll 则有条件触发 和 事件触发（也可以叫边缘触发）两种。

在选择使用哪种方式的时候，需要根据 fd 的多少和活跃程度来判断。当fd 数量较少，且都比较活跃的时候，使用 select 或者 poll 反而有可能效率更高，因为毕竟 epoll 要有多次的回调函数。

1. epoll哪些触发模式，有啥区别

**水平触发**：内核会通知你一个fd是否就绪，然后才可以对这个就绪的fd进行I/O操作。就算你没有任何操作，系统还是会继续提示fd已经就绪

**边缘触发**：当fd从未就绪变为就绪时，内核会通知fd已经就绪，并且内核认为你知道该fd已经就绪，不会再次通知了，除非因为某些操作导致fd就绪状态发生变化。如果一直不对这个fd进行I/O操作，导致fd变为未就绪时，内核同样不会发送更多的通知，因为only once

LT可以理解为水平触发，只要有数据可以读，不管怎样都会通知。而ET为边缘触发，只有状态发生变化时才会通知，可以理解为电平变化。

1. I/O多路复用和多线程多进程的区别

与多线程和多进程相比，I/O 多路复用的最大优势是系统开销小，系统不需要建立新的进程或者线程，也不必维护这些线程和进程

1. select返回可读，结果只读到0字节，什么情况

连接正常断开

1. http 与 https的区别

http是HTTP协议运行在TCP之上。所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。  
  
https是HTTP运行在SSL/TLS之上，SSL/TLS运行在TCP之上。所有传输的内容都经过加密，**加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密**。此外客户端可以验证服务器端的身份，如果配置了客户端验证，服务器方也可以验证客户端的身份

1. 常见的加密算法

对称加密算法：公钥和私钥相同 DES AES

非对称加密算法：公钥和私钥不相同 RSA DSA ECC。**公钥负责加密，发布给其他人，私钥负责解密，只有自己拥有。**

第三部分 C++

1. const 与 valitale

volatile关键字可以用来提醒编译器它后面所定义的变量随时有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。如果没有volatile关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。

1. extern c

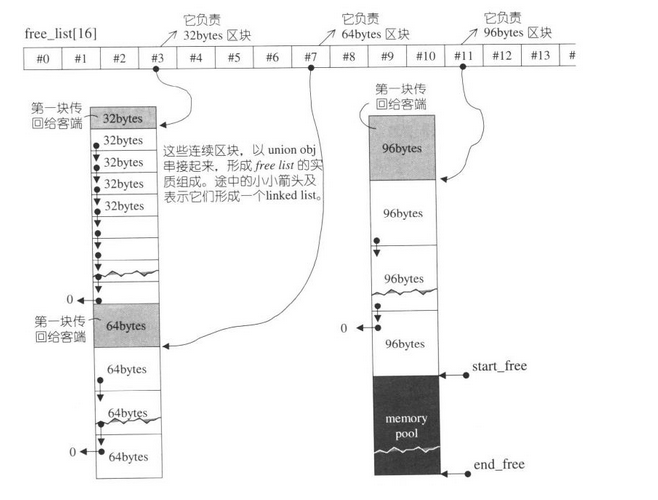
extern "C"，表明它按照类C的编译和连接规约来编译和连接，而不是C++的编译的连接规约. 由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名，也就是所谓的name manging。

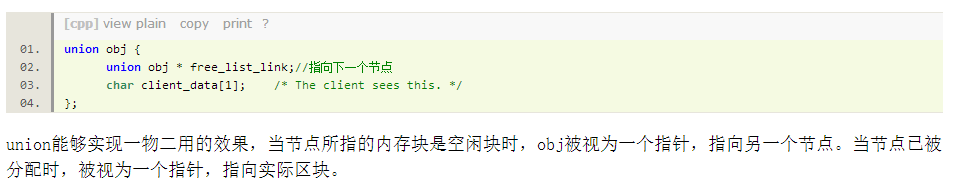
1. 内存分配器allocator

分配器就用于处理容器对[内存的分配与释放](https://zh.wikipedia.org/wiki/%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86)请求。换句话说，分配器用于封装[STL](https://zh.wikipedia.org/wiki/STL)容器在内存管理上的低层细节. 默认情况下，C++标准库使用其自带的通用分配器，但根据具体需要，程序员也可自行定制分配器以替代之

一级空间配置器：对new和delete的包装

二级空间配置器：减少小区块造成的内存碎片（128字节），使用free\_list结构





1. 虚函数的实现原理

虚函数指针及虚表（在构造函数中完成）

1. new和malloc的区别

malloc分配的是裸内存

new 则先分配内存，再接着在此内存上构造对象

1. 使用迭代器需注意的地方

遍历容器的时候对容器修改将使当前迭代器失效

删除迭代器指向的对象时，返回的是下一个元素的迭代器

删除容器中的某个元素的正确写法

for (auto it = container.bigen();;it!=container.end())

it = container.erase(it)

1. RAII

Resource acquisition is initialization 资源获取就是初始化，它提供了一种资源自动管理的方式，当产生异常、回滚等现象时，RAII可以正确地释放掉资源

原理：利用stack上的临时对象生命期是程序自动管理的这一特点，将我们的资源释放操作封装在一个临时对象中，利用构造函数来获取资源，利用析构函数来释放资源，移动语义就可以用来进行资源的转移.

单例模式，智能指针中都有用到

1. STL容器

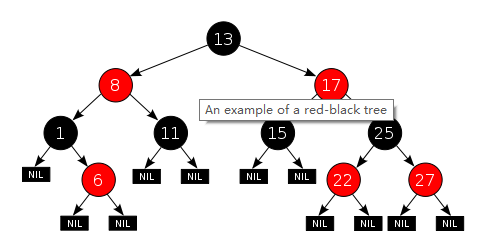
vector：实现原理（动态增长）

list：双向链表，高效的插入和删除

map：红黑树，是一种[自平衡二叉查找树](https://zh.wikipedia.org/wiki/%E8%87%AA%E5%B9%B3%E8%A1%A1%E4%BA%8C%E5%8F%89%E6%9F%A5%E6%89%BE%E6%A0%91)，[O](https://zh.wikipedia.org/wiki/%E5%A4%A7O%E7%AC%A6%E5%8F%B7)(log n)时间内做查找，插入和删除

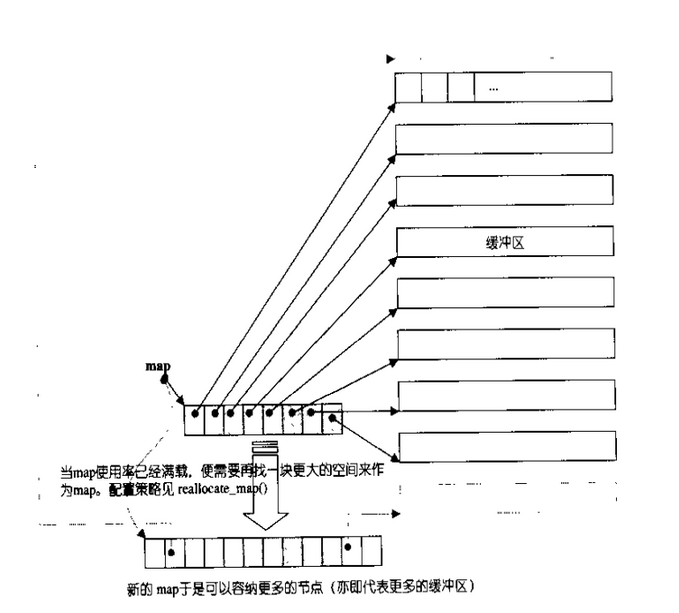
**从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的**

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是NIL节点）。
4. 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）
5. 从任一节点到其每个叶子的所有[简单路径](https://zh.wikipedia.org/wiki/%E9%81%93%E8%B7%AF_(%E5%9B%BE%E8%AE%BA))都包含相同数目的黑色节点。



heap：默认最大堆，堆排序算法

deque:双向队列

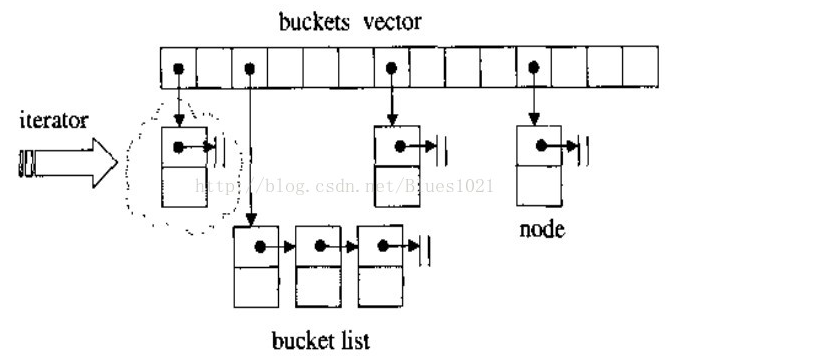


deque结构图

queue：单向队列，底层实现为deque

priority\_queue：优先级队列，底层实现为heap

unorder\_map：无序map（哈希map），底层实现为Hash



取值过程：

1. 得到key
2. 通过hash函数得到hash值
3. 得到桶号(一般都为hash值对桶数求模)
4. 比较桶的内部元素是否与key相等，若都不相等，则没有找到。
5. 取出相等的记录的value。
6. 能不能在构造函数中做复杂操作？

构造函数中只进行那些没有实际意义的（trivial，译者注：简单初始化对于程序执行没有实际的逻辑意义，因为成员变量的“有意义”的值大多不在构造函数中确定）初始化，可能的话，**使用Init()方法集中初始化为有意义的（non-trivial）数据。**

**定义：**在构造函数中执行初始化操作。

**优点：**排版方便，无需担心类是否初始化。

**缺点：**在构造函数中执行操作引起的问题有：

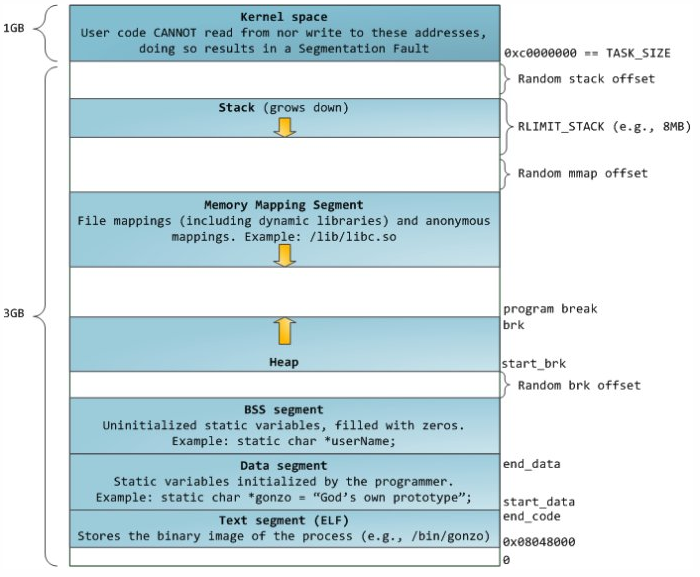
1) **构造函数中不易报告错误，不能使用异常。(构造函数中使用异常将导致析构函数不能执行，如果在抛出异常之前有申请动态内存，将引发内存泄漏)**

2) 操作失败会造成对象初始化失败，引起不确定状态。

3) **构造函数内调用虚函数，调用不会派发到子类实现中**，即使当前没有子类化实现，将来仍是隐患。

第四部分 操作系统

1. 程序内存分布



1. 堆和栈的区别

**大小**：堆的大小决定于系统的虚拟内存空间 32 4G，栈则较小，可以使用ulimits -s 查看并修改

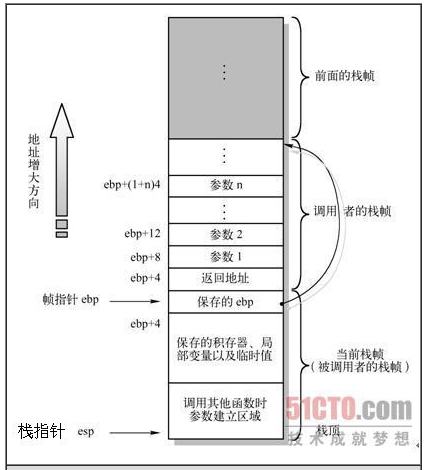
**管理方式：**堆手动管理，声明周期不受函数作用域限制，栈由系统管理，生命周期和函数作用域一致

1. 定位内存泄露

mtrace mtrace的原理是记录每一对malloc-free的执行，若每一个malloc都有相应的free，则代表没有内存泄露，对于任何非malloc/free情況下所发生的内存泄露问题，mtrace并不能找出来。

也就是对于New的内存，只能检测出有泄露，但不能定位

1. C语言函数调用过程



ESP是栈顶指针，EBP是存取堆栈指针，可理解为EBP为基地址，ESP为偏移地址

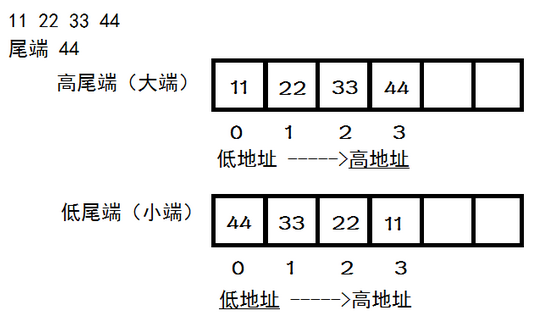
1. 判断系统是32位还是64位 编译器环境位数

sizeof(void\*)

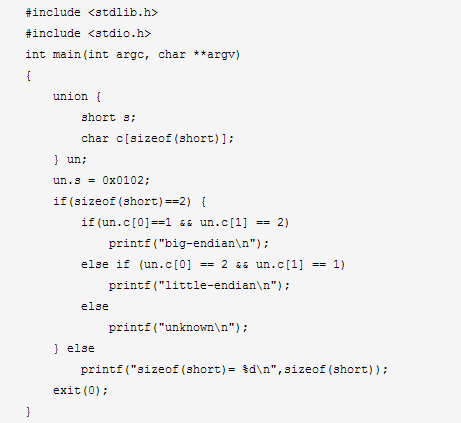
编译器环境位数 与os位数

1. 大端和小端，写程序判断

判断低地址（尾端）存的数字的高位（大端）还是低位（小端）



windwos为小端，网络字节序为大端



1. linux的内存管理机制

虚拟地址空间（线性地址）：隔离不同进程使用的内存地址空间

提高内存的使用率

扩展内存，即运行所需内层大于物理内存的程序

决定程序运行地址（重定位问题）

寻址过程：逻辑地址->线性地址->物理地址

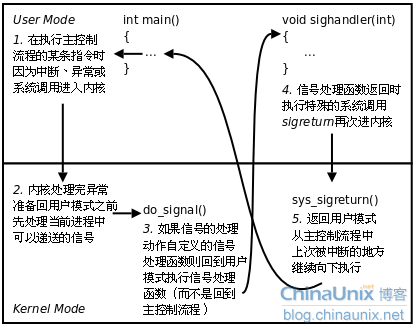
分页机制：MMU(Memory Management Unit)

1. linux的任务调度机制

0(1)调度算法：核心是每个CPU都有一个优先级数组，每个优先级下都挂着具有该优先级的进程链表。在维护一个优先级位图，用于快速找到第一个不为空的优先级，取出它下面的第一个程序进行调度。同时系统维护一个运行进程数组及过期进程数组，一个程序时间片用完后就将其加到过期数组中，当运行数组为空时就切换过期数组和运行数组。

CFS完全公平调度算法：O(1)算法是根据进程的优先级来选择调度进程的，而CFS是根据进程的虚拟运行时间来进行调度。此处利用红黑树来进行查找

1. 系统如何将一个信号通知到进程



1. 进程间通信方式

**文件**

**信号**：一种[异步](https://zh.wikipedia.org/w/index.php?title=%E5%BC%82%E6%AD%A5&action=edit&redlink=1)的通知机制，用来提醒[进程](https://zh.wikipedia.org/wiki/%E8%BF%9B%E7%A8%8B)一个事件已经发生

**信号量**：是在多线程环境下使用的一种设施, 它负责协调各个线程, 以保证它们能够正确、合理的使用公共资源

**套接字：**

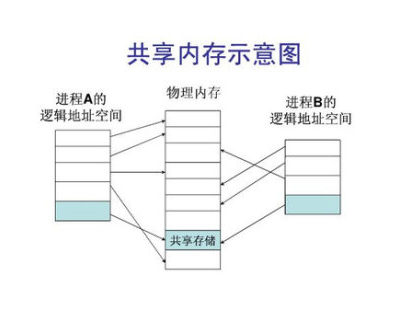
**消息队列**

**管道、命名管道**

**共享内存**

1. **共享内存的实现原理**

**进程A的某一块逻辑地址空间和进程B的某一块逻辑地址空间映射到同一快物理内存上**



1. **动态链接和静态链接的区别**

**静态链接：链接的过程中拷贝库文件到自己的程序中，独占，占用系统资源，不存在冲突。**

动态链接: 把库函数推迟到程序运行时期载入,如果内存里已有此库函数的拷贝了则共享那一个拷贝；没有才链接载入。节省了系统的内存资源。却必须保证库函数为可再入入函数。

提供更为灵活的程序控制方式，只需修改DLL就可以完成程序功能的改变，而无需重新编译源程序。

1. **一个进程的多个线程间资源访问问题（哪些共享，哪些独占）**

共享：进程代码段、进程打开的文件描述符、信号的处理器、进程的当前目录、全局变量、heap，static变量

独占：寄存器、栈（局部变量）

1. **linux系统的各类同步机制**

1）锁：自旋锁spinlock（苦等，并不挂起程序，用于短代码段的互斥，例如某个计数器的修改）

读写锁

RCU （read-copy update）

2）互斥：

原子操作：不可中断的。需硬件支持，例如atomic\_read

信号量：与自旋锁差别：能导致睡眠

3）等待队列

1. **开多少线程比较合适（不确定）**

**2个CPU四个核心，开4个线程**

1. **同步异步，阻塞非阻塞**

1）同步与异步  
**同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)**所谓同步，就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。  
换句话说，就是由\*调用者\*主动等待这个\*调用\*的结果。  
  
而异步则是相反，\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在\*调用\*发出后，\*被调用者\*通过状态、通知来通知调用者，或通过回调函数处理这个调用。  
  
典型的异步编程模型比如Node.js  
  
举个通俗的例子：  
你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下"，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。  
而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。  
  
2）阻塞与非阻塞  
**阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.**  
  
阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。  
非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。  
  
还是上面的例子，  
你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟check一下老板有没有返回结果。  
在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

第五部分数据结构和算法

排序：

1. 插入
2. 冒泡
3. 选择
4. 归并
5. 快速
6. 堆
7. 桶
8. 计数排序
9. 基数排序

树:

1. 二叉树的前中后序遍历（非递归版本）
2. 哈夫曼树（左右权值不一样如何处理？）
3. 二叉搜索树（建立，查找，删除）
4. 层次遍历
5. 红黑树
6. 平衡二叉树AVL
7. 最近祖先节点
8. 判断一个树是不是另外一棵树的子树
9. 二叉树的镜像
10. 求和为指定数的路径
11. 反转二叉树(递归与非递归版)
12. 字典树trie
13. Suffix trie后缀树

链表：

1. 求2条链表的公共节点
2. 判断链表有无环，环的位置
3. 链表的克隆(链表节点中存储的有任意的节点指针)
4. 反转链表
5. 快排链表版

字符串

1. 搜索（KMP）
2. 最长无重复子串
3. 最长回文串
4. 构造回文串：如何删除任意字符使得字符串为最长回文串
5. 正则表达式（。表示任意单字符 \*表示任意个前一个字符）
6. 正则表达式（？表示任意单字符 \*表示任意序列）
7. 有效的括号匹配个数
8. 包含指定字符的最短子串
9. 最长公共子串（顺序一致，并要求连续）
10. 最长公共子序列（顺序一致，不要求连续）
11. 将a变成b的最短编辑距离
12. 插入一个字符
13. 删除一个字符
14. 替换一个字符

图

1. 图的遍历（广度，深度）
2. 最短路径（ Dijkstra，Frued）
3. 最小生成树
4. 有向无环图的建立与输出（有效路径）
5. Union find

|  |  |
| --- | --- |
| 2016年9月 | 四川大学研究生二等奖学金 |
| 2015年9月 | 四川大学研究生二等奖学金 |
| 2016年6月 | 四川大学校园文明监督队优秀队委 |
| 2014年12月 | 兰州大学学生标兵 |
| 2014年12月 | 兰州大学国家励志奖学金 |
| 2013年12月 | 兰州大学学生标兵 |
| 2013年12月 | 兰州大学优秀学生二等奖学金 |
| 2012年12月 | 兰州大学学生标兵 |
| 2012年12月 | 兰州大学国家励志奖学金 |

|  |  |  |
| --- | --- | --- |
| 2016.02-2016.06 | 四川大学 | 校园文明监督队东园支队干部 |
| 2015.09-2016.01 | 四川大学 | 校园文明监督队东园支队干事 |
| 2012.09-2013.06 | 兰州大学计算机学院 | 学生会秘书组织部部长 |
| 2011.09-2012.06 | 兰州大学计算机学院 | 学生会秘书组织部干事 |