Interview 面试

郑华

2018 年 6 月 6 日

目录

1 参考学习网址

http://blog.csdn.net/Hackbuteer1/article/category/1181402

http://blog.csdn.net/qingyuanluofeng/article/category/2544593/4

网络编程过来人：http://blog.csdn.net/Adam040606/article/category/3186109

面试题：http://blog.csdn.net/huaweitman/article/category/1487663

后台开发同道：http://blog.csdn.net/fycy2010/article/category/5648813

2 Linux

2.1 题目

2.1.1 熟练 netstat tcpdump ipcs ipcrm

2.1.2 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

共享内存定义 ：共享内存是最快的可用 IPC（进程间通信）形式。它允许多个不相关的进程去访问**同一部分逻辑内存**。共享内存是由 IPC 为一个进程创建的一个特殊的地址范围，**它将出现在进程的地址空间中**。**其他进程可以把同一段共享内存段“连接到”它们自己的地址空间里去**。所有进程都可以访问共享内存中的地址。如果一个进程向这段共享内存写了数据，所做的改动会立刻被有访问同一段共享内存的其他进程看到。因此共享内存对于数据的传输是非常高效的。

共享内存段紧靠在栈之下，最大限制为 32M

共享内存的使用实现原理 ：要使用一块共享内存，进程必须首先分配它。随后需要访问这个共享内存块的每一个进程都必须将这个共享内存绑定到自己的地址空间中。当完成通信之后，所有进程都将脱离共享内存，并且由一个进程释放该共享内存块。

理解 Linux 系统内存模型可以有助于解释这个绑定的过程。在 Linux 系统中，每个进程的虚拟内存是被分为许多页面的。这些内存页面中包含了实际的数据。每个进程都会维护一个从内存地址到虚拟内存页面之间的映射关系。尽管每个进程都有自己的内存地址，不同的进程可以同时将同一个内存页面映射到自己的地址空间中，从而达到共享内存的目的。

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。

2.1.3 进程内存空间分布情况

内核空间和用户空间 ->

所有 os 分配给每个进程一个独立的，连续的，虚拟的，地址内存空间，该大小一般是 4G（32 位操作系统，**即 2 的 32 次方**），其中将高地址值的内存空间分配给 os 占用，**linux os 占用 1G，window os 占用 2G**；*其余内存地址空间分配给进程使用*；分别称作内核空间和用户空间；

通常情况下，用户进程不能访问内核空间的地址，例外情况是用户进程通过系统调用访问内核空间；

用户空间布局 ->

用户空间**被分成几个段 (Segment)**，从高地址到低地址分别为：Stack, Heap, BSS, Data, Text;

- **Stack**: `local variables`
- **Heap**: `dynamically allocated withnew delete malloc free`
- **BSS**: `unInitialized global and static data`
- **Data**: `initialized global and static data`
- **Text**: *编译后的代码段*

`Notice->`

- Text，ro-data，Data，BSS 四个段的**大小是在程序编译时确定的**；

- Text，ro-data，Data 三个段的内容存储在可执行文件中；

- Text, ro-data, Data, BSS 四个段的内容是**常驻内存的**，即进程从开始运行到僵死它们都一直存在内存，所以访问它们用的是**常量地址**

- Stack Segment 被放在高地址值是有原因的，原因是：调用函数（加帧）是减 esp 的，函数返回（减帧）是加 esp 的，调用在前，所以栈是向低地址扩展的，放在高地址比较合适

多线程程序与普通程序的内存布局的不同 ->

多线程程序和普通程序在内存中的**不同只要表现在栈的不同**，*每一个线程一个栈，所以线程的局部变量不会受到其他线程的影响。*而**.text, .data. .bss 等部分段是各个线程共享的**，所以线程间通信很简单，可以直接在这些共享内存中存取就可以了；

所谓祸福相依，线程内操作这些共享内存的数据时候，要非常小心，以避免有两个或多个线程同时操作同一块内存区域；

其他细节 ->

- 初始化了的：
  - 初始化了的全局变量：`data segment`
  - 初始化了的静态变量：`data segment`
- 未初始化的：
  - 未初始化的全局变量：`bss segment`
  - 未初始化的静态变量：`bss segment`
- 类中静态变量：`data segment`
- 字符串常量：`ro-data segment`
- 局部变量：`stack segment`
- New/malloc ：`heap Segment`

2.1.4 ELF 是什么？其大小与程序中全局变量的是否初始化有什么关系

`Linux ELF ELF = Executable and Linkable Format`，可执行连接格式，是 UNIX 系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的。扩展名为`elf`。工具接口标准委员会 (TIS) 选择了正在发展中的 **ELF 标准作为**工作在 32 位 INTEL 体系上**不同操作系统之间可移植的二进制文件格式**。假定开发者定义了一个二进制接口集合，ELF 标准用它来支持流线型的软件发展。应该减少不同执行接口的数量。因此**可以减少重新编程重新编译的代码**。

2.1.5 动态链接和静态链接的区别？

静态库与动态库

- **静态库**：函数和数据被编译进一个二进制文件 (通常扩展名为.LIB)。在使用静态库的情况下，在**编译链接**可执行文件**时**，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块 **组合起来创建最终的可执行文件 (.EXE 文件)**

- **导入库 (动态库)**：在使用动态链接库的时候，往往提供两个文件：**一个引入库和一个 DLL**。引入库包含被 DLL 导出的函数和变量的符号名，DLL 包含实际的函数和数据。**在编译链接可执行文件时**，只需要链接引入库，DLL 中的函数代码和数据并不复制到可执行文件中，**在运行的时候**，再去加载 DLL，访问 DLL 中导出的函数。

– 装入：在运行 Windows 程序时，它通过一个被称作“**动态链接**”的进程与 Windows 相接。一个 Windows 的.EXE 文件拥有它使用不同动态链接库的引用，所使用的函数即在那里。

当 Windows 程序被加载到内存中时，程序中的调用被指向 DLL 函数的入口，如果 DLL 不在内存中，系统就将其加载到内存中。当链接 Windows 程序以产生一个可执行文件时，你必须链接由编程环境提供的专门的“导入库 (import library) 库”。这些导入库包含了动态链接库名称和所有 Windows 函数调用的引用信息。

链接程序使用该信息在.EXE 文件中构造一个表，当加载程序时，Windows 使用它将调用转换为 Windows 函数。

静态库与导入库的区别 ->

导入库和静态库的区别很大，他们实质是不一样的东西。**静态库**本身就包含了实际执行代码、符号表等等，而**对于导入库而言**，其实际的执行代码位于动态库中，导入库只包含了地址符号表等，确保程序找到对应函数的一些基本地址信息。

动态链接与静态链接

- 静态链接方法:`#pragma comment(lib, "test.lib")`，静态链接的时候，载入代码就会把程序会用到的动态代码或动态代码的地址确定下来

静态库的链接可以使用静态链接，**动态链接库**也可以使用这种方法链接导入库

- 动态链接方法:`LoadLibrary()/GetProcessAddress()`和`FreeLibrary()`，使用这种方式的程序并不在一开始就完成动态链接，而是直到真正调用**动态库**代码时，载入程序才计算 (被调用的那部分) 动态代码的逻辑地址，然后等到某个时候，程序又需要调用另外某块动态代码时，载入程序又去计算这部分代码的逻辑地址，所以，这种方式使程序初始化时间较短，但运行期间的**性能比不上静态链接的程序**。

简单的说，**静态库**和应用程序编译在一起，在任何情况下都能运行，**而动态库**是动态链接，顾名思义就是在应用程序启动的时候才会链接，所以，当用户的系统上没有该动态库时，应用程序就会运行失败。

动态库：

1. 共享：多个应用程序可以使用同一个动态库，启动多个应用程序的时候，只需要将动态库加载到内存一次即可；
2. 开发模块好：要求设计者对功能划分的比较好。

静态库：代码的装载速度快，执行速度也比较快，因为编译时它只会把你需要的那部分链接进去，应用程序相对比较大。但是如果多个应用程序使用的话，会被装载多次，浪费内存。

2.1.6 32 位系统一个进程最多有多少堆内存

2.1.7 写一个 c 程序辨别系统是大端 or 小端字节序

https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/Dev/Network/BasicSocket/ByteOder.cc

2.1.8 信号：列出常见的信号，信号怎么处理？

2.1.9 i++ 是否原子操作？并解释为什么？

原子操作 原子操作是**不可分割的**，在执行完毕不会被任何其它任务或事件中断，分为两种情况（两种都应该满足）

1. 在单线程中，能够在单条指令中完成的操作都可以认为是” 原子操作”，因为中断只能发生于指令之间
2. 在多线程中，不能被其它进程（线程）打断的操作就叫原子操作

i++ 分为三个阶段：内存到寄存器、寄存器自增、写回内存。这三个阶段中间都可以被中断分离开, 所以不是原子操作

http://blog.csdn.net/yeyuangen/article/details/19612795

2.1.10 说出你所知道的各类 linux 系统的各类同步机制（重点），什么是死锁？如何避免死锁

2.1.11 如何实现守护进程？

2.1.12 linux 的任务调度机制是什么？

2.1.13 标准库函数和系统调用的区别？

2.1.14 系统如何将一个信号通知到进程？

2.1.15 fork() 一子进程程后父进程的全局变量能不能使用？

2.1.16 多线程与多进程的区别

线程安全的条件 ：要确保函数线程安全，**主要需要考虑的是线程之间的共享变量**。

属于**同一进程**的**不同线程**会**共享进程内存空间中的全局区和堆，而私有的线程空间**则主要**包括栈和寄存器**。因此，对于同一进程的不同线程来说，**每个线程的局部变量都是私有的，而全局变量、局部静态变量、分配于堆的变量都是共享的**。在对这些共享变量进行访问时，如果要保证线程安全，则必须通过加锁的方式。

关于线程的堆栈 生成子线程后，它会获取一部分该进程的堆栈空间，作为其**名义上**的独立的私有空间。（为何是名义上的呢？）由于，这些线程属于同一个进程，其他线程只要获取了你私有堆栈上某些数据的指针，其他线程便可以自由访问你的名义上的私有空间上的数据变量。（注：而**多进程**是不可以的，因为不同的进程，**相同的虚拟地址，基本不可能映射到相同的物理地址**）

2.1.17 多线程的各种锁！

1. **原子操作**: 原子操作比普通操作效率要低，因此必要时才使用
2. **自旋锁**: 等待解锁的进程将**反复检查锁是否释放**，而不会进入睡眠状态 (**忙等待**)，所以常用于短期保护某段代码同时，持有自旋锁的进程也不允许睡眠，不然会造成死锁——因为睡眠可能造成持有锁的进程被重新调度，而再次申请自己已持有的锁
3. **互斥量**: 用于线程的**互斥**
4. **信号量**: 用于线程的**同步**

信号量与 mutex 是`sleep-waiting`。就是说当没有获得`mutex` 时，会有上下文切换，**将自己、加到忙等待队列中**，直到另外一个线程释放 *mutex* 并唤醒它，而**这时 CPU 是空闲的，可以调度别的任务处理**。

自旋锁 spin lock 是`busy-waiting`。就是说当没有可用的锁时，就一直忙等待并不停的进行锁请求，直到得到这个锁为止。这个过程中 cpu 始终处于忙状态，**不能做别的任务**

锁介绍http://blog.csdn.net/wilsonboliu/article/details/19190861

自旋锁https://blog.poxiao.me/p/spinlock-implementation-in-cpp11/

2.1.18 缓存淘汰算法-LRU

最近最少使用 (Least Recently Used) 淘汰算法。

http://flychao88.iteye.com/blog/1977653

2.1.19 可重入与不可重入

可重入：概念基本没有比较正式的完整解释，但是它比线程安全要求更严格。根据经验，所谓“重入”，**常见的情况是**，程序执行到某个函数 *foo()* 时，收到信号，于是暂停目前正在执行的函数，转到信号处理函数，而这个信号处理函数的执行过程中，又恰恰也会进入到刚刚执行的函数 foo()，这样便发生了所谓的重入。此时如果 foo() 能够正确的运行，而且处理完成后，之前暂停的 *foo()* 也能够正确运行，则说明它是可重入的。

http://www.cnblogs.com/simplepaul/p/7361032.html

http://blog.csdn.net/yeyuangen/article/details/38318709

2.1.20 crontab 原理

crontab 实际上是启动服务后读取所有配置文件，然后睡一段时间（一般也是一分钟）后运行下一个任务，从链接里看，是**半小时监控一次 crontab 文件是否有改动**。对 suse 而言，crontab 服务启动后，会检查/var/spool/cron/tabs 下所有用户的定时任务，然后加载到内存的队列中，然后每分钟检查一下crontab文件是否有改动。

可以理解为 cron 服务有 2 个线程，一个是调度处理**定时任务**，一个**后台线程**，后台线程检查配置文件是否有变动，如果有变动，则发送信号到调度进程，调度进程再重新读取配置文件更新内存中的任务队列，同理，service cron stop 也是发信号到cron服务（本次确认 cron 服务半僵死就是因为无法响应了）导致调度线程不知道变动，没有更新内存中的队列。从/var/log/messages 观察，依然是修改或删除之前的定时任务在运行。

因此，为了保险，在用户里crontab -r 删除用户的定时任务后，建议重启一下crontab 服务（*service cron stop* 后记得 *service cron status* 查看一下是否停止成功，如果没有 *kill -9* 再 *service cron start*），也就是**清空一下当前定时任务的内存队列**，让 cron 服务重新加载，更重要的是防止 cron 服务挂住没重新加载定时任务队列。

2.1.21 *coreDump

http://blog.csdn.net/tenfyguo/article/details/8159176/

概念 我们经常听到大家说到**程序 core 掉**了，需要**定位解决**，这里说的大部分是指对应**程序由于各种异常或者 bug 导致在运行过程中异常退出或者中止**，并且在满足一定条件下**会产生一个叫做 core 的文件**。

通常情况下，**core 文件会包含**了程序运行时的内存，寄存器状态，堆栈指针，内存管理信息还有各种函数调用堆栈信息等，我们可以理解为是程序工作当前状态存储生成第一个文件，许多的程序出错的时候都会产生一个 core 文件，通过工具分析这个文件，我们可以定位到程序异常退出的时候对应的堆栈调用等信息，找出问题所在并进行及时解决。

coreDump 文件 core 文件**默认的存储位置与对应的可执行程序在同一目录下**，文件名是core，大家可以通过下面的命令看到 core 文件的存在位置：cat /proc/sys/kernel/core_pattern

缺省值是core

如何判断一个文件是 coredump 文件 在类 unix 系统下，coredump文件本身主要的格式也是ELF格式，因此，我们可以通过readelf 命令进行判断.

可以通过简单的file 命令进行快速判断文件是什么类型。

coredump 产生的几种可能情况

1. 内存访问越界
2. 多线程程序使用了线程不安全的函数
3. 多线程读写的数据未加锁保护
4. 非法指针
5. 堆栈溢出

利用 gdb 进行 coredump 的定位 堆错误示例：

- 先写一个存在段错误的代码
- 编译成 gdb 文件，需要-g 选项编译
- 运行代码，崩溃，产生 core 文件
- 利用 file core 查看错误信息

2.1.22 线程达到线程池定义个数，怎么办

首先线程池有一个排队策略

排队策略

1. **直接提交**。工作队列的默认选项是 SynchronousQueue，它**将任务直接提交给线程而不保持它们**。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。
2. **无界队列**。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）**将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize**。（因此，maximumPoolSize 的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列
3. **有界队列**。当使用有限的 maximumPoolSizes 时，**有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制**。队列大小和最大池大小可能需要相互折衷：**使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量**。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。**使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。**

2.1.23 Linux 能同时启动多少个线程

对于 32-bit Linux，一个进程的地址空间是 4G, 其中用户态能访问的是 3G，而一个线程的默认栈为 10M，心算便知，一个进程大约最多能同时启动 300 个线程

2.1.24 多线程能提高并发度么？

如果指的是“并发连接数”，不能

假如单纯采用 thread per connection 模型，那么并发连接数大约 300，这远远低于基于事件的单线程程序所能轻松达到的并发数（几千上万），所谓基于事件，指的是用 I/O multiplexing event loop 的编程模型，又称 Reactor 模式

所以多线程并不能提高并发度。

2.2 惊群

https://blog.csdn.net/lyztyycode/article/details/78648798?locationNum=6&fps=1

惊群效应也有人叫做雷鸣群体效应，不过叫什么，简言之，惊群现象就是多进程（多线程）在同时阻塞等待同一个事件的时候（休眠状态），如果等待的这个事件发生，那么他就会唤醒等待的所有进程（或者线程），但是最终却只可能有一个进程（线程）获得这个时间的“控制权”，对该事件进行处理，而其他进程（线程）获取“控制权”失败，只能重新进入休眠状态，这种现象和性能浪费就叫做惊群。

为了更好的理解何为惊群，举一个很简单的例子，当你往一群鸽子中间扔一粒谷子，所有的各自都被惊动前来抢夺这粒食物，但是最终注定只可能有一个鸽子满意的抢到食物，没有抢到的鸽子只好回去继续睡觉，等待下一粒谷子的到来。这里鸽子表示进程（线程），那粒谷子就是等待处理的事件。

解决 accept 加锁、SO_REUSEPORT

3 C++

3.1 题目

https://www.zhihu.com/question/34574154?sort=created

http://www.cnblogs.com/fangyukuan/archive/2010/09/18/1829871.html

http://www.cnblogs.com/Y1Focus/p/6707121.html

http://www.cnblogs.com/LUO77/p/5771237.html

http://www.cnblogs.com/bozhicheng/p/6259784.html

http://www.cnblogs.com/simplepaul/p/6820533.html

3.1.1 文本文件和二进制文件的区别

ASCII 文件也称为文本文件，这种文件在磁盘中存放时每个字符对应一个字节，用于存放对应的 ASCII 码

ASCII 码文件可在屏幕上按字符显示，例如源程序文件就是 ASCII 文件，用 DOS 命令 TYPE 可显示文件的内容。由于是按字符显示，因此能读懂文件内容。

二进制文件是按二进制的编码方式来存放文件的。例如，数 5678 的存储形式为：00010110 00101110 只占二个字节。二进制文件虽然也可在屏幕上显示，但其内容无法读懂。C 系统在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。输入输出字符流的开始和结束只由程序控制而不受物理符号 (如回车符) 的控制。因此也把这种文件称作“流式文件”。

3.1.2 结构体用 memcmp 比较的问题

可以通过`memcmp()` 来比较 2 个相同的结构体变量，但这 2 个变量必须在赋值前进行清零初始化（否则结果不准确）

即比较前需要 `memset(&x, 0 ,sizeof(x))`, 然后再 `memcmp(&t3, &t4, sizeof(CmpTest))`

3.1.3 memcpy 实现

3.1.4 strcpy 实现

3.1.5 strcat 实现

3.1.6 书写给定类的赋值操作符 =

- 形式：`className& className::operator =(const className& rhs)`
- 自我赋值:`if(*this == rhs)`
- 连续赋值:返回 `className &`
- 参数不变性:`const className& rhs`
- 异常安全性:用交换技术

3.1.7 什么是“引用”？申明和使用“引用”要注意哪些问题？

1. 引用就是某个**目标变量**的“别名”(alias)，对应用的操作与对变量直接操作效果完全相同。
2. 申明一个引用的时候，切记要对其进行初始化。
3. 引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把该引用名作为其他变量名的别名。
4. 声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。

5. 不能建立数组的引用。

6. 引用效率比指针会更高一些，因为引用不需要判断是否合法，因为是引用则必然存在其真实变量。

3.1.8 将“引用”作为函数参数有哪些特点？

- 传递引用给函数与传递指针的效果是一样的，直接操作原实参。
- 在内存中并没有产生实参的副本，效率和空间占用更好。
- 用更容易使用，更清晰，易于程序员阅读。

3.1.9 在什么时候需要使用“常引用”？

既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变

```
string foo( );
void bar(string&s)
// 那么下面的表达式将是非法的:
bar(foo( ));
bar("hello world");
```

原因在于foo( ) 和"hello world" 串都会产生一个临时对象，而在 C++ 中，这些**临时对象**都是const 类型的。因此上面的表达式就是试图将一个const 类型的对象转换为非const 类型，这是非法的。

引用型参数应该在能被定义为const 的情况下，尽量定义为const。

3.1.10 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
}
```

- **好处**: 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，产生 runtime error!

- **须遵守的规则**

1. 不能返回局部变量的引用
2. 不能返回函数内部new 分配的内存的引用
3. 可以返回类成员的引用，但最好是const

3.1.11 引用与指针的区别是什么？

指针通过某个指针变量指向一个对象后，对它所指向的变量**间接操作**。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。

3.1.12 什么时候需要“引用”？

1. 赋值操作符= 的参数
2. 拷贝构造函数的参数
3. 赋值操作符= 的返回值
4. 流操作符<< 和>>

3.1.13 结构与联合有和区别？

结构和联合都是由多个不同的数据类型成员组成
区别:

1. 但在任何同一时刻, 联合中**只存放了一个被选中的成员**（所有成员共用一块地址空间）, **而结构的所有成员都存在**（不同成员的存放地址不同）。

2. **对于联合的不同成员赋值, 将会对其它成员重写**, 原来成员的值就不存在了, 而对于结构的不同成员赋值是互不影响的。

3.1.14 已知 strcpy 的函数原型：char *strcpy(char *strDest, const char *strSrc) 其中 strDest 是目的字符串，strSrc 是源字符串。不调用 C++/C 的字符串库函数，请编写函数 strcpy

答：

```
#include <assert.h>
#include <stdio.h>
char*strcpy(char*strDest, constchar*strSrc)
{
    assert((strDest!=NULL) && (strSrc !=NULL)); // 2分
    char* address = strDest;     // 2分
    while( (*strDest++=*strSrc++) !='\0' )            // 2分
    NULL;
    return address ;       // 2分
}
```

3.1.15 不用中间变量实现交换 swap 的问题

```
void swap(int& a, int& b)
{
    a += b;
    b = a - b;
    a -= b;
}
```

3.1.16 # include<file.h> 与 # include "file.h" 的区别？

前者是从Standard Library 的路径寻找和引用<file.h>，而后者是从当前工作路径搜寻并引用"file.h"

3.1.17 面向对象的三个基本特征，并简单叙述之？

- **封装**：将客观事物抽象成类，每个类对自身的数据和方法实行 protection(private, protected,public)
- **继承**：
- **多态**：系统能够在运行时，能够根据其类型确定调用哪个重载的成员函数的能力，称为多态性

3.1.18 重载（overload) 和重写 (overried，有的书也叫做“覆盖”）的区别？

重载：是指**允许**存在**多个同名函数**，而这些函数的**参数表不同**（或许**参数个数**不同，或许**参数类型**不同，或许两者都不同）。
重写：是指**子类重新定义父类**虚函数的方法。

3.1.19 多态的作用？

概念：

- 首先多态的意思是：在面向对象语言中，接口的多种不同的实现方式，多态性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，赋值以后，父对象就可以根据当前赋值给他的子对象的特性以不同的方式运作。

- 比如有一个父类superClass，它有 2 个子类subClass1，subClass2。superClass 有一个方法func(), 两个子类都重写了这个方法。那么我们可以定义一个superClass 的引用obj，让它指向一个子类的对象，比如superClass obj = new subClass1(); 那么我们调用obj.func() 方法时候，会进行**动态绑定**，也就是obj 它的实际类型的func() 方法，即subClass1的func() 方法。同样你写superClass obj = new subClass2();obj.func() 其实调用的是subClass2的func() 方法。**这种由于子类重写父类方法，然后用父类引用指向子类对象，调用方法时候会进行动态绑定，这就是多态。**

- 多态对**程序的扩展**具有非常大的作用，比如你要再有一个subClass3，你需要改动的东西会少很多，要是使用了配置文件那就可以不动源代码了。

作用：

- 隐藏实现细节，使得代码能够模块化；扩展代码模块，实现代码重用；

- 接口重用：为了类在继承和派生的时候，保证使用家族中任一类的实例的某一属性时的正确调用。

3.1.20 virtual 指针与对象初始化的区别

对象，有无虚析构无关要紧，都会调用父类析构函数

如果是用父类指针的话，调用子类初始化，由于当父类指针指向子类对象，父类析构不是虚函数，则执行时，不能执行到子类的析构函数。

3.1.21 New delete 与 malloc free 的联系与区别?

都是在堆 (heap) 上进行动态的内存操作。

用malloc 函数需要指定内存分配的字节数并且**不能**初始化对象，new **会**自动调用对象的构造函数。delete **会**调用对象的destructor，而free **不会**调用对象的destructor.

new 类型安全、返回的是某种数据类型指针，malloc 非类型安全、返回的是void 指针

关于**内存分配失败**：

- new 会抛出bad_alloc 异常

- malloc 则会返回NULL 指针

3.1.22 有哪几种情况只能用 intializationlist 而不能用 assignment?

1. 当类中含有const 成员变量
2. 当类中含有reference 成员变量
3. 基类的构造函数都需要初始化表

3.1.23 C++ 是不是类型安全的？

不是。两个**不同类型**的指针之间**可以强制转换**

3.1.24 指针函数与函数指针

http://blog.csdn.net/ameyume/article/details/8220832

指针函数：就是返回值是一个地址！

3.1.25 请说出 const 与 #define 相比，有何优点？

const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行**类型安全检查**。而对后者只进行**字符替换**，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

有些集成化的调试工具可以对const 常量进行**调试**，但是**不能**对宏常量进行调试。

3.1.26 简述数组与指针的区别？

- **数组**要么在静态存储区被创建（如全局数组），要么在栈上被创建。**指针**可以随时指向任意类型的内存块。
- 修改内容上的差别

```
char a[] = “hello”;
a[0] = ‘X’;
char *p = “world”; // 注意p 指向常量字符串
p[0] = ‘X’; // 编译器不能发现该错误，运行时错误
```

- 用运算符sizeof 可以计算出数组的容量（字节数）。sizeof(p),p 为指针得到的是一个指针变量的字节数，而不是p 所指的内存容量

```
char a[] ="hello world";
char*p = a;
cout<<sizeof(a) << endl; // 12 字节
cout<<sizeof(p) << endl; // 4 字节
```

- C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

```
void Func(char a[100])
{
    cout<<sizeof(a) << endl; // 4 字节而不是100 字节
}
```

3.1.27 类成员函数的重载、覆盖和隐藏区别？

成员函数被重载的特征

1. 相同的范围（在同一个类中）；
2. 函数名字相同；
3. 参数不同；
4. virtual 关键字可有可无。

覆盖是指派生类函数覆盖基类函数，特征是

1. 不同的范围（分别位于派生类与基类）；
2. 函数名字相同；
3. 参数相同；
4. 基类函数必须有virtual 关键字。

“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下

1. 如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。
2. 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）

3.1.28 如何打印出当前源文件的文件名以及源文件的当前行号？

```
cout << __FILE__ ;
cout<<__LINE__ ;
```

__FILE__和__LINE__ 是系统预定义宏，这种宏并不是在某个文件中定义的，而是由编译器定义的

3.1.29 关于 ___stdcall 和 ___cdecl 调用方式的理解

http://blog.csdn.net/myjisgreat/article/details/46477769

案例理解 stdcall 就是“我”把书（参数）依次放在了书堆顶 (压入堆栈)，事先与服务员约定好我已经放了 N 本书（N 个字节），“服务员”按照约定查看 N 本书（我压入的参数），做好他的工作后，他按照约定的书本个数 N，来把 N 本书从书堆上移除（清理堆栈，释放那几个参数的空间，而不是释放整个堆栈）

cdelcall 是“我”把书（参数）依次放在了书堆顶，没有与服务员约定书的多少，服务员查看我给的书，然后呼叫我，我自己把刚刚放上的书移除

清理区别 ->

cdecl 是调用者恢复堆栈的，假设有一百个不同的函数调用函数 a 那么内存中就有一百端恢复堆栈的代码，是不是很浪费空间呢？

stdcall 是函数恢复堆栈，只有在函数代码的结尾出现一次恢复堆栈的代码，所以节约空间

```
//cdecl :内存中“实际”代码，指令都是保存在内存中，看到a和c分别调用b，内存中就有两处恢复堆栈的代码
   b函数
   {
      do something
      return
   }

   a函数调用b函数
   {
      push 参数(参数进栈)
      call 函数
      sub esp…(恢复堆栈) // 调用者清理
   }

   c函数调用b函数
   {
      push 参数(参数进栈)
      call 函数
      sub esp…(恢复堆栈) // 调用者清理
   }

//stdcall :可以看到，只有b函数内部有一份恢复堆栈的代码，调用函数的地方不再有重复代码
   b函数
   {
      do something
      恢复堆栈 // 只有一处清理，自己负责清理
      return
```

```
}

a函数调用b函数
{
    push 参数(参数进栈)
    call 函数//这里没有恢复堆栈的代码
}

c函数调用b函数
{
    push 参数(参数进栈)
    call 函数
}
```

参数区别 ->

cdecl 的优势在于他**可以不定参数个数**，参考 printf 函数. 原因在于是调用者存入参数，调用者释放参数占有的空间，都是调用者完成的，所以有参数个数的自由性

stdcall 在结束函数时，回复的空间是编译时决定的，函数负责释放，但他无法知道你实际压入几个参数，于是 stdcall 在编译时就规定了参数个数，**无法实现不定个数的参数调用**

3.1.30 腾讯-输出是什么

```
main()
{
    int a[5]={1,2,3,4,5};
    int *p=(int *)(&a+1);
    printf("%d",*(p-1));
}
```

1. &a a 是一个数组名，也就是数组的首地址。
2. 对 a 进行取地址运算符，得到的是一个**指向数组的指针！**也就相当于 int (*p) [5] = &a;
3. a 是一个指针，它指向的是一个包含5个int 元素的数组！
4. 执行a+1 后，p 的偏移量相当于 a + sizeof(int) * 5 ！
5. 程序中强制将指针p 转换成一个int* 那么 p -1 其实就是 p - sizeof(int)

4 STL 组件与使用

4.1 题目

4.1.1 使用过哪些组件？

使用过哪些组件？其实现原理及复杂度, 迭代器的失效场景..

http://blog.csdn.net/rainkop/article/details/8423732

4.2 体会

固然我们强调工作中不要重新发明轮子，但是，作为一个合格的程序员，应该具备自制轮子的能力。**非不能也，是不为也！**

在深入理解 STL 实现后，运用 STL 自然手到擒来. 并能自动避免一些错误和低效的用法

5 Boost 组件与使用

5.1 题目

boost 的网络库 ASIO boost 的网络库 ASIO 使用过么，有什么体会

6 网络、服务器编程

6.1 参考 C++_NetProgram

6.2 题目

http://www.cnblogs.com/nancymake/p/6516933.html

6.2.1 多线程和多进程的区别

1. **进程**数据是分开的: 共享复杂，需要用 IPC，同步简单；**多线程**共享进程数据：共享简单，同步复杂
2. **进程**创建销毁、切换复杂，速度慢 ；**线程**创建销毁、切换简单，速度快
3. **进程**占用内存多，CPU 利用率低；**线程**占用内存少，CPU 利用率高
4. **进程**编程简单，调试简单；**线程** 编程复杂，调试复杂
5. **进程间**不会相互影响 ；**线程**一个线程挂掉将导致整个进程挂掉
6. **进程**适应于多核、多机分布；**线程**适用于多核

线程所私有的 -
线程 id、寄存器的值、栈、线程的优先级和调度策略、线程的私有数据、信号屏蔽字、errno 变量、

6.2.2 多线程锁的种类有哪些？

6.2.3 自旋锁和互斥锁的区别？

6.2.4 进程间通信和线程间通信

6.2.5 多线程程序架构，线程数量应该如何设置？

6.2.6 什么是原子操作，gcc 提供的原子操作原语，使用这些原语如何实现读写锁？

6.2.7 网络编程设计模式，reactor/proactor/半同步半异步模式？

- **Reactor 模式**：同步阻塞 I/O 模式，注册对应读写事件处理器，等待事件发生进而调用事件处理器处理事件。 proactor 模式：异步 I/O 模式。Reactor 和 Proactor 模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor 中需要应用程序自己读取或者写入数据，Proactor 模式中，应用程序不需要进行实际读写过程。

  主线程往 epoll 内核上注册 socket 读事件，主线程调用`epoll_wait` 等待`socket` 上有数据可读，当`socket` 上有数据可读的时候，主线程把`socket` 可读事件放入请求队列。睡眠在请求队列上的某个工作线程被唤醒，处理客户请求，然后往 epoll 内核上注册 socket 写请求事件。主线程调用`epoll_wait` 等待写请求事件，当有事件可写的时候，主线程把 socket 可写事件放入请求队列。睡眠在请求队列上的工作线程被唤醒，处理客户请求。

- **Proactor**: 主线程调用`aio_read` 函数向内核注册`socket` 上的读完成事件，并告诉内核用户读缓冲区的位置，以及读完成后如何通知应用程序，主线程继续处理其他逻辑，当`socket` 上的数据被读入用户缓冲区后，通过信号告知应用程序数据已经可以使用。应用程序预先定义好的信号处理函数选择一个工作线程来处理客户请求。工作线程处理完客户请求之后调用`aio_write` 函数向内核注册`socket` 写完成事件，并告诉内核写缓冲区的位置，以及写完成时如何通知应用程序。主线程处理其他逻辑。当用户缓存区的数据被写入`socket` 之后内核向应用程序发送一个信号，以通知应用程序数据已经发送完毕。应用程序预先定义的数据处理函数就会完成工作。

- **半同步半异步模式**：上层的任务（如：数据库查询，文件传输）使用同步 I/O 模型，简化了编写并行程序的难度。而底层的任务（如网络控制器的中断处理）使用异步 I/O 模型，提供了执行效率。

6.2.8 有一个计数器，多个线程都需要更新，会遇到什么问题，原因是什么，应该如何做？如何优化？

6.2.9 如果 select 返回可读，结果只读到 0 字节，什么情况？

6.2.10 connect 可能会长时间阻塞，怎么解决?

6.2.11 keepalive 是什么东西？如何使用？

6.2.12 socket 什么情况下可读？

6.2.13 udp 调用 connect 有什么作用？

6.2.14 socket 编程，如果 client 断电了，服务器如何快速知道？

6.2.15 怎么清理僵尸进程

6.2.16 系统调用函数

wait fork

6.2.17 socket 的阻塞和非阻塞的概念

6.2.18 信号与信号量之间的区别

6.2.19 TCP 头大小，包含字段？三次握手，四次断开描述过程，都有些什么状态。状态变迁图。TCP/IP 收发缓冲区

6.2.20 使用 udp 和 tcp 进程网络传输，为什么 tcp 能保证包是发送顺序，而 udp 无法保证？

6.2.21 epoll 哪些触发模式，有啥区别？

6.2.22 tcp 与 udp 的区别（必问）为什么 TCP 要叫做数据流？

6.2.23 流量控制和拥塞控制的实现机制

Nagle 交互控制
慢启动- 拥塞窗口
滑动窗口
拥塞避免算法

6.2.24 滑动窗口的实现机制

滑动窗口机制，窗口的大小并不是固定的而是根据我们之间的链路的带宽的大小，这个时候链路是否拥护塞。接受方是否能处理这么多数据了。　滑动窗口协议，是 TCP 使用的一种流量控制方法。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。

6.2.25 epoll 和 select 的区别?

6.2.26 网络中，如果客户端突然掉线或者重启，服务器端怎么样才能立刻知道?

6.2.27 TTL 是什么? 有什么用处，通常那些工具会用到它? ping? traceroute? ifconfig? netstat?

6.2.28 linux 的五种 IO 模式/异步模式.

6.2.29 请说出 http 协议的优缺点.

6.2.30 NAT 类型，UDP 穿透原理

6.2.31 大规模连接上来，并发模型怎么设计

6.2.32 流量控制与拥塞控制的区别，节点计算机怎样感知网络拥塞了?

7 多线程编程

7.1 参考 C++_Advanced

8 算法

8.1 数据结构

快速排序优化：https://blog.csdn.net/insistgogo/article/details/7785038

- pivot 选择优化
-
-

9 程序设计

9.1 设计模式

9.2 UML

10 OpenGL

10.1 参考 OpenGL

11 DirectX

11.1 参考 DirectX9

12 PC 游戏试玩记录

12.1 坦克世界

12.2 生死狙击

连跳 https://zhidao.baidu.com/question/873824108590170252.html http://news.4399.com/gonglue/ssjj/yxgl/wf/675362.html

1. 首先 W 往前助跑
2. W 跳出去
3. 空中松开 W
4. 在落地瞬间按蹲（虽然说是瞬间，可每个人的键盘都可能会有延迟，所以要多跳尝试，我就是在空中 2/3 的时候就要按蹲了）
5. 然后再滑一次轮滚（鼠标中间的）
6. 然后同样在落地瞬间按蹲, 即一直重复一个动作

升级跳

12.3 Dota2

1. 起步-新手 对于新手这些东西确实挺不好理解的，比如说什么被动，冷却时间，出什么装备.. 等，这块要是加个对于这些名词的解释会有更好的效果...

2. 准备-新玩 了解了一般名词后，对于 Dota2 的游戏玩起来还是比较有吸引力，集中在设置了金钱的诱惑陷阱，和何时出击击杀英雄与占塔.., 最主要的是... 每局比赛都需要将近半个小时的时间，而这则是对游戏中英雄技能了解和学习的阶段.. 不是通过试玩，而是与人机对战.. 除此，游戏中的英雄种类繁多，这是他长存的另一核心..

12.4 CrossFire

12.5 League Of Legends

12.6 文明 5

12.7 极品飞车-系列

12.8 QQ 飞车–鹏鹏卡丁车