1. **Typedef的使用必须明白！**
2. **基类与派生类的转换：（谭浩强page192）**

* **派生类对象可以向基类对象赋值(过程是单向的)，同一基类的不同派生类之间不能赋值**
* **派生类对象可以代替基类对象向基类对象的引用进行赋值或初始化！（基类对象的引用与派生类中的基类部分共享同一段内存）**
* **如果函数的参数是基类对象或基类对象的引用，相应的实参可以用子类对象！**
* **派生类对象的地址可以赋给指向基类对象的指针变量，也就是说指向基类对象的指针变量也可以用来指向派生类对象！（多态实现）**

**4.在最后的派生类中不仅要负责对其直接基类进行初始化还要负责对虚基类初始化！（这一点和以往的方式截然不同。原因见谭浩强page189）**

5动态库：

* **ldd** file:查看file链接了哪些动态库。（只能查看隐式加载动态库的可执行文件，对于dlopen操作的动态库该命令无效！）
* **ldd -u** file:查看file中无用的被加载的动态库。（现代化的编译器已经做了智能优化！所以该命令已经没用！）
* **-Wl,--as-needed**:编译器选项，编译时不连接无用的动态库！（现代化的编译器已经不需要该选项自动优化了已经！）
* 指定动态库的链接路径使用：**export LD\_LIBRARY\_PATH=**/LIB\_PATH
* **-Wl,rpath=**PATH：编译器选项，编译时指定动态库的位置使得动态库被正确编译进可执行文件，但可执行文件运行时却不一定能够找到动态库的位置，加上此选项使动态库的路径被编译进可执行文件，再也不用担心可执行文件找不到可执行文件！

1. 值初始化和默认初始化的区别
2. 直接初始化和拷贝初始化需要了解二者不同:成员的类型决定了它如何拷贝 : 对类类型的成员, 会使用其拷贝构造函数来拷贝;内置类型的成员则直接拷贝!!在函数调用过程中,具有非引用类型的参数要进行拷贝初始化.类似的,当一个函数具有非引用的返回类型时,返回值会被用来初始化调用方的结果.
3. 派生类中虚函数的返回类型必须与基类函数匹配. 该规则存在一个例外,当类的虚函数返回类型是类本身的指针或引用时,上述规则无效.亦即如果D由B派生得到, 则基类的虚函数可以返回B\*而派生类的对应函数可以返回D\*,只不过这样的返回类型要求从D到B的类型转换是可访问的.
4. 虚基类: 继承间接共同基类时只保留一份成员!虚基类不是在声明基类时声明的,而是在声明派生类时指定继承方式时声明的. 为了保证虚基类在派生类中只继承一次, 应当在该基类的所有直接派生类中声明为虚基类, 否则仍然会出现对基类的多次继承!（如果第二代有两种实现，第三代应该继承第二代的哪个？ 曰：继承第一代的实现！）
5. Qt两枚工具的用法

**moc** file.**h** **-o moc\_**file.**cpp**

**uic** file.**ui** -o **ui\_**file.**h**

1. **Qt UI构图中必不可少的语句：setupUi。它用来初始化ui窗口部件和布局或者理解为使能ui设计。**
2. qmake -project之后需要在.pro中加入：

**QT += core gui widgets** //至于要添加哪一个可以assitant查看头文件简介

greaterThan(QT\_MAJOR\_VERSION, 4) //向下兼容qt4，不添加也无所谓！

1. Qt中信号参数数目可以比槽数据多，如果比槽参数少只能有两种connect格式使用：

* Qt4 connect的SIGNAL和SLOT机制，槽多出的参数必须带有默认参数
* Qt5的Lambda形式connect

为什么Qt5不能使用函数指针的connect！因为**c/c++标准规定，默认参数值只能使用在直接的函数调用中，当使用函数指针时，默认参数是不可见的！**

1. Qt添加对c++11的支持：

* CONFIG += c++11
* QMAKE\_CXXFLAGS += -std=c++11
* QMAKE\_CXXFLAGS += -std=gnu++11

1. 在qt程序中添加调试：

在.pro文件中添加下列语句之一

**1. CONFIG += debug**

**2. QMAKE\_CXXFLAGS += -g**

1. linux下使用ctag跨文件跳转设置：

在.vimrc下添加语句：

set tags=tags; //Debian下无效

set autochdir

1. vim下显示函数列表：
2. sudo apt install excuberant-ctags //现在的ctags已经被excuberant-ctags取代
3. 进入/etc/vim/vimrc输入如下数据：

* set nu
* syntax on
* set autoindent
* set cindent
* let Tlist\_Use\_Right\_Window=1
* let Tlist\_File\_Fold\_Auto\_Close=1

在vim下输入Tlist即可显示函数列表！

1. 64位ubuntu运行32位程序或者使用32位工具链无法运行。使用下面三条语句可以使之兼容：

* sudo dpkg --add-architecture i386 //
* sudo apt update
* sudo apt install libc6:i386 libncurses5:i386 libstdc++6:i386

1. 使用数组初始化vector对象：（始料不及）

vector<int> vec(begin(int\_array), end(int\_array)); //将内置类型数组int\_array中元素保存到vector对象vec中，也可以部分保存。begin(int\_array)和end(int\_array)是c++11标准，这种转换会使得vec.size()等于sizeof(int\_array)！

1. assign代替顺序容器的赋值操作，以解决相容但类型不同的容器之间的赋值问题。
2. emplace等操作必须掌握，注意其与insert,push\_back,push\_front的区别！emplace具有先进性！
3. string支持容器的很多操作，除了push\_front和pop\_front。其行为尤其类似vector！
4. array不支持push、pop、insert、emplace、erase、clear等操作，因为这些操作会改变容器大小。
5. 顺序容器中的assign成员必须掌握。不同类型容器即使元素类型相同也不能直接赋值，必须经过assign适配。另外assign有两种版本!
6. 容器类型不同依然可以通过指定迭代器范围进行赋值操作（无需使用assign函数适配）。
7. linux下可以使用mtrace和muntrace函数来调试内存泄漏等问题！backtrace函数簇用于获取程序的调用堆栈
8. makefile目标文件可以是列表
9. makefile的五种元素：

* 显式规则
* 隐晦规则
* 变量定义
* 文件指示
* 注释

1. makefile中#字符表注释，\#表示#字符
2. include关键字可以把其他makefile包含进来，其格式为：

[-]include <filename> //include前面不能以tab键开始，filename可以包含通配符和路径等，也可以是文件列表。包含-字符可以使得找不到filename时不会报错!

1. make参数”-I”或“—include-dir“用于指定包含的makefile路径！
2. 环境变量MAKEFILES和include的区别与联系！这个环境变量经常会引起问题，不要使用为好!
3. makefile支持三个通配符：\* ? […],所以如果我们的文件名中如有\*应该用\转义
4. makefile中的变量就相当于c/c++中的宏，所以不能在定义变量处使用通配符（不能使用通配符赋值）。例如：object = \*.cpp 那么object的值就是\*.cpp而不是指所有.cpp文件，如果要通配所有的.cpp可以：object :=$(wildcard \*.cpp)。
5. VPATH和vpath用于指定文件的搜索路径。前者是makefile中的特殊变量（不是环境变量），后者是makefile的关键字而且更为灵活，一定要注意这里指定的路径只对makefile有效，对于编译器如gcc、g++等却是不适用的。指定的路径可以是相对路径和绝对路径，可以指定路径列表，不同路径以:字符隔开。
6. 利用伪目标特性可以一次生成多个可执行文件！（其实不用伪目标也可以）
7. 静态模式规则很强大！
8. 自动生成依赖性！简直强大。对于目标文件所依赖的头文件，完全可以不去理会（当然路径如果不在当前，路径则要指定）。只要运行在目标文件的命令处添加一行：gcc -M test.c指令即可（或者是g++等其他编译器）这里的-M默认会输出所有的头文件，-MM默认只输出自定义头文件。自动找到
9. 如果想让上一条命令运用于下一条命令需要将两条命令写在一行并以分号（包括分号斜线[;\]再换行）或者&&隔开。
10. 每个命令运行后make会检测每个命令的返回码。忽略某条命令是否成功执行的方法是在命令前面加上-字符。如果想全局指定可以给make指定参数-i或者—ignore-errors。Make参数-k或—keep-going的功能是：如果某规则中的命令出错了，那么就终止该规则的执行，但继续执行其他规则。
11. 嵌套执行make时，如果想将主Makefile中的变量导入子Makefile中使用关键字：export <variable …>(不包含尖括号)。如果variable列表为空则将变量全部导入子目录。这样导入的变量并不会覆盖子Makefile中的变量除非指定参数-e。如果不想将变量传递到下级Makefile中使用参数unexport <variable …>。需要注意的是有两个变量不管是否export都会导入，他们是SHELL和MAKEFLAGS。 //那么默认导入吗？
12. 变量使用$取出其值，如果要使用真实的$需要使用$$。使用$提取变量时最好带上()或者{}。另外在makefile中使用shell指令取出变量的值也要使用$$这一点和普通的取值操作是不同的。
13. 一般情况下makefile中的变量会被make在命令行上指定的同名变量替代。如果不想被替代可以使用关键字override指定。
14. makefile中有两种条件判断：ifeq、ifneq、ifdef、ifndef。
15. makefile中所有函数的调用都是以$来标识的，其语法格式如下：

$(<function>, <arguments>) 或

${<function>, <arguments>} //dash/sh以{}修饰可以但函数不能高亮

参数之间以,隔开！

1. makefile中常用函数：

字符串处理函数： //text字符串之间以空格分隔

* $(subst <from>,<to>,<text>) //字符串替换函数！
* $(patsubst <pattern>,<replacement>,<text>) //模式字符串替换函数！
* $(strip <string>) //去空格函数，去除开头和结尾的空格（不理解）
* $(findstring <find>,<in>) //查找字符串函数，在in中查找find
* $(filter <pattern…>,<text>) //过滤函数，只保留符合pattern的字符串，可以含有多个pattern
* $(filter-out <pattern…>,<text>) //过滤函数，结果与filter函数相反
* $(sort <list>) //排序函数，升序！（重复的会被去除）
* $(word <n>,<text>) //取单词函数，取text中第n个单词
* $(wordlist <s>,<e>,<text>) //取text中第s到第e个单词串
* $(words <text>) //单词个数统计函数
* $(firstword <text>) //取字符串中第一个单词

文件名操作函数：

* $(dir <names…>) //取目录函数
* $(notdir <names…>) //取目录函数，与上面相反
* $(suffix <names…>) //取后缀函数
* $(basename <names…>) //取前缀函数
* $(addsuffix <suffix>,<names…>) //加后缀函数
* $(addprefix <prefix>,<names…>) //加前缀函数
* $(join <list1>,<list2>) //连接函数，将list2中的单词加到list1中的单词后面，这个很容易误解。易错

foreach函数：

* $(foreach <var>,<list>,<text>) //foreach是个循环函数。该语句意为将list的单词逐一取出放到参数<var>所指定的变量中，然后再执行text所指定的表达式，每一次text会返回一个字符串，循环过程中text所返回的字符串以空格分隔当整个循环结束时text所返回的字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值

if函数：

* $(if <condition>,<then-part>)
* $(if <condition>,<then-part>,<else-part>)//这两个函数非常类似。意为如果condition返回值不为空则返回then-part否则返回else-part。如果else-part不存在则返回空。

call函数：

* $(call <expression>,<para1>,<para2>,<para3>…) //call函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用call函数来向这个表达式传递参数。

origin函数：

* $(origin <variable>) //告诉我们variable变量是哪里来的（无$）

shell函数：

* $(shell <command>) //执行shell命令

控制make的信息：

* $(error <text …>) //产生一个致命错误，<text …>是错误信息
* $(warning <text …>) //警告而不退出

1. makefile中可以通过${CURDIR}获得本makefile的路径，该路径包括本makefile的文件名！
2. makefile中对于shell命令不同的行除了使用;\隔开之外他们之间都是独立的，例如上面shell定义的变量下面的shell命令不能使用。让我想到了$$的用法！
3. 共享内存不会主动释放，同一个程序如果没有清除共享内存，则重新打开该进程共享内存一般不会重新分配。
4. 信号量的使用主要是用来保护共享资源，信号量为正的时候说明她空闲，请求线程/进程锁定而使用她。若为0说明她被占用，请求线程/进程要进入睡眠队列中，等待被唤醒。信号量分为两种：内核信号量和用户态信号量！用户态信号量又分为有名信号量和无名信号量。
5. 只有可以睡眠的函数才可以获得内核信号量，中断处理程序和可延迟函数都不能使用内核信号量。
6. 有名信号量，其值保存在文件中，所以它可以用于线程也可以用于进程间同步，而无名信号量其值保存在内存中。
7. 无名信号量常用于多线程间的同步，同时也用于相关进程（父子进程）间的同步。也就是说，无名信号量必须是多个进程（线程）的共享变量，无名信号量要保护的变量也必须是多个进程（线程）的共享变量，这两个条件是缺一不可的。
8. 有名信号量是位于共享内存区的，那么它要保护的资源也必须是位于共享内存区，只有这样才能被无相关的进程所共享。（有待考证，不是说有名信号量保存在文件中吗？）。
9. select和epoll函数要掌握（epoll是linux下特有的）。 //必须掌握

select三个劣势：

* 完成一次监控如果还要监控需要重新将文件描述符注册进内核，这样频繁的写入消耗了大量的时间。
* 响应来了她并不知道到底是哪个或者哪些描述符发生了事件，需要全部遍历。
* 监控的数目默认比较小一般为1024，随着数目的增加其执行效率将迅速下降。

epoll只需要注册一次。每一个发生的事件都被放入队列维护，不需要全部遍历。

随着描述符数量的增大其效率并不会下降。被监控的文件描述符通过mmap函数来读写。

1. Gdb下执行shell命令：

* shell command
* !command
* make [argu] //执行makefile

1. Unix/Linux系统内部不使用文件名，而使用inode号码来识别文件。对于系统来说，文件名只是inode号码便于识别的别称或者绰号。表面上，用户通过文件名，打开文件。实际上，系统内部这个过程分成三步：首先，系统找到这个文件名对应的inode号码；其次，通过inode号码，获取inode信息；最后，根据inode信息，找到文件数据所在的block，读出数据。
2. ioctl：函数
3. pthread\_create()：函数对于一个线程标识符可以创建很多线程，只是这些线程的线程号是不同的。对于pthread\_join()直到全部线程退出后她才放行！
4. Ubuntu下不能man pthread库函数需要安装：glibc-doc和manpages-posix-dev。
5. 线程函数：

* pthread\_exit(void \*retval)：线程退出函数！主线程使用则效同pthread\_join
* pthread\_cancel(pthread\_t tid)：一个线程取消同属于同一进程的tid线程的执行！tid线程收到这个信号后并不一定停止执行，这取决于tid线程的处理方式，pthread\_cancel请求发出后并不会在此等待！
* pthread\_cleanup\_push(…)：
* pthread\_cleanup\_pop(…)：和上面函数配合使用用于清理线程资源。和上面函数必须配对。
* pthread\_detach()：pthread\_detach(pthread\_self())  
  linux线程执行和windows不同，pthread有两种状态joinable状态和unjoinable状态，  
  如果线程是joinable状态，当线程函数自己返回退出时或pthread\_exit时都不会释放线程所占用堆栈和线程描述符（总计8K（使用ulimit -s查看））。只有当你调用了pthread\_join之后这些资源才会被释放。  
  若是unjoinable状态的线程，这些资源在线程函数退出时或pthread\_exit时自动会被释放。

unjoinable属性可以在pthread\_create时指定，或在线程创建后在线程中pthread\_detach自己, 如：pthread\_detach(pthread\_self())，将状态改为unjoinable状态，确保资源的释放。或者将线程置为 joinable,然后适时调用pthread\_join.

1. 进程和线程原语对应关系：

|  |  |  |
| --- | --- | --- |
| fork | pthread\_create | Create a new flow of control |
| exit | pthread\_exit | Exit from an existing flow of control |
| waitpid | pthread\_join | Get eixt status from of control |
| atexit | pthread\_cleanup\_push | Register function to be called at exit from flow of control |
| getpid | pthread\_self | Get id for flow of control |
| abort | pthread\_cancel | Request abnormal termination of flow of control |

1. 线程间同步：（UNIX）

* **Mutexes：**资源被占用则将自己置入等待队列睡觉等待，直到mutex锁被释放，期间cpu可以执行其他任务。
* **Reader-Writer Locks**：Reader-Writer locks are similar to mutexes, except that they allow for higher degree of parallelism.Three states are possible with a reader-writer lock:locked in read mode, locked in write mode,and unlocked.Only one thread at a time can hold a reader-writer lock in write mode,but multiple threads can hold a reader-writer lock in read mode at the same time.
* **Condition Variables**
* **Spin Locks(自旋锁)：**与mutex不同之处在于如果不能获得spin lock则一直忙等待（busy-waiting），忙等待的含义是指如果不能获得lock则一直轮询lock直到获得lock资源。期间cpu一直被占用不能执行其他任务。
* **信号量**
* **Barriers**

1. Mutex机制函数。

* pthread\_mutex\_init();
* pthread\_mutex\_destroy();用于销毁init了的锁。
* pthread\_mutex\_lock();
* pthread\_mutex\_trylock();非阻塞的上锁，如果资源没有被锁上则成功，返回零，否

则失败，返回EBUSY;

* pthread\_mutex\_timedlock();在预定的时间内功能与pthread\_mutex\_lock()相同，超出了时间而没有解锁则出错返回。
* pthread\_mutex\_unlock(); 用于解除lock和trylock的锁。

1. 读写锁函数： //不知道是linux下没有实现还是ubuntu16.04没有实现

* pthread\_rwlock\_init();
* pthread\_rwlock\_destroy();
* pthread\_rwlock\_rdlock();
* pthread\_rwlock\_wrlock();
* pthread\_rwlock\_unlock();
* pthread\_rwlock\_trylock();
* pthread\_rwlock\_timedrdlock();
* pthread\_rwlock\_timedwrlock();

1. 条件变量函数： //

* pthread\_cond\_init();
* pthread\_cond\_destroy();
* pthread\_cond\_wait();
* pthread\_cond\_timedwait();
* pthread\_cond\_signal();
* pthread\_cond\_broadcast();

1. Spin locks函数 //notice the difference between spin lock and mutex lock.

* pthread\_spin\_init(); //ubuntu下没有该函数和…trylock，Debian全无
* pthread\_spin\_destroy();
* pthread\_spin\_lock();
* pthread\_spin\_trylock();
* pthread\_spin\_unlock();

1. Barriers函数 //ubuntu下有七个函数,debian下没有

* pthread\_barrier\_init();
* pthread\_barrier\_destroy();
* pthread\_barrier\_wait();

1. 查看当前网段下已经使用的ip和mac，三种方式：

* nmap -sP 192.1.1.0/24 //进行ping扫描，打印出对扫描做出响应的主机
* nmap -PU 192.1.1.0/24 //使用UDP ping探测主机
* nmap -sS 192.1.1.0/24 //SYN扫描，又称为半开放扫描

上面三枚指令都可以获得ip相关信息，并保存到文件/proc/net/arp中。

1. 同属于一个进程下的线程共享signal。因此如果其中一个线程更改了信号的行为则该行为对于其他线程同样有效。可以使用pthread\_sigmask、sigwait来更改这种共享！
2. break … if …：gdb下设置条件断点！
3. until/u：运行直到退出循环体！ //shell脚本循环，这个真的对吗？
4. 断点相关

* break lineno
* break functionname //c++中可以使用class::function和function(type, type)
* break +offset //在当前行号的下面offset行设置断点
* break -offset
* break filename:lineno
* break filename:functionname
* break \*address //在程序运行的内存地址处停住！
* break //在下一条指令处停住
* break … if … //如果if条件成立则设置断点
* info breakpoints/break [n] //查看断点号为n的断点信息
* clear //清除断点
* delete [breakpoints] [range…] //删除断点
* disable [breakpoints] [range…] //
* enable [breakpoints] [range…] //
* enable [breakpoints] once range //
* enable [breakpoints] delete range… //

1. 为断点设置运行命令：

**break** foo **if** statements

**commands**

command-lists

**end**

1. handle用以处理信号，其几个主要参数如下：

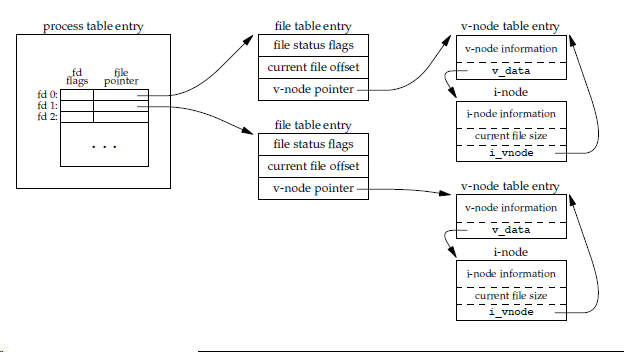
* nostop
* stop
* print
* noprint
* pass
* noignore
* nopass
* ignore

以上参数可以组合

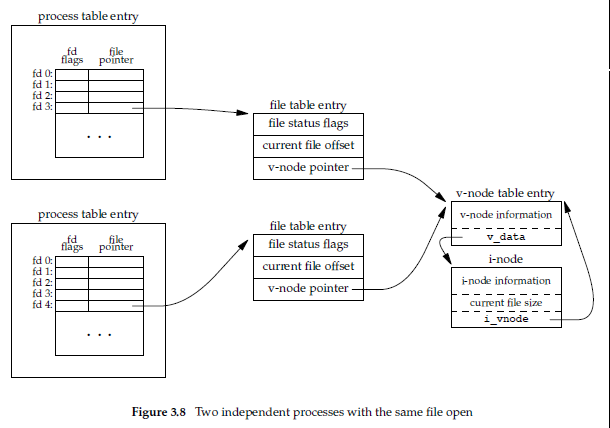
signal SIGNAL:通过gdb给当前程序发送信号SIGNAL

1. 守护进程与终端没有任何联系，也就和标准输入输出和error没有联系，所以我们看不到守护进程的标准输入、标准输出、标准错误是正常的。
2. 守护进程的一些不成文约定：

* 锁文件（如果使用锁文件）保存在/var/run/x.pid
* 配置文件（如果支持配置）保存在/etc/x.conf(If a daemon has a configuration file, the daemon reads the file when it starts, but usually won’t look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch SIGHUP and reread their configuration files when they receive the signal. Since they aren’t associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive SIGHUP. Thus they can safely reuse it.)
* 守护进程可以开始于命令行，但他们通常开始于系统的初始化脚本（/etc/rc\* 或/etc/init.d/\*）

1. record lock：linux下记录锁的功能是当多个进程同时操作一枚文件时使用record lock可以锁定文件的某个区域不被其它程序修改。三个函数可以完成这个功能：fcntl、flock、lockf
2. mmap函数提供了内存映射技术，对于io操作使用mmap函数免去了read和write的复制过程（系统调用过程）以此可以提高io效率。可以将文件映射到进程地址空间，可以私有映射，还可以共享映射。通过返回的指针操作该区域的能力！
3. Linux下异步I/O和I/O多路复用的使用以及区别很重要。异步I/O的使用需要和信号结合，但是该信号只能对应一个I/O操作。详情请咨询apue；
4. ps -A -opid,stime,etime,args：查看进程运行时间和启动时间（etime：运行时间。stime：开启时刻。args：进程名。opid：进程号。A显示所有进程信息）（逗号两边没有空格）
5. 对象的移动是我国现阶段发展的迫切需要。（移动构造函数，移动赋值运算符）！
6. setjmp/longjmp是很重要的两个函数！
7. 上图是一个进程打开两个不同文件的过程，下图是一个进程打开一个相同文件的过程。Linux下没有v-node。总结如下：

* 一枚打开操作对应一个file table entry，dup()和dup2()函数创建新的文件描述符指向一个已经打开的文件，但依然共用一个file table entry！对于父子进程其也会共享file table entry！
* 一枚文件对应一枚v-node table entry和i-node，而跟打开它的进程个数无关！



1. **apropos**：根据参数查找解释中包含该字段的命令！
2. fsch：检查和修复文件系统！
3. mkfs：在磁盘上建立文件系统！也就是以某种格式格式化！
4. dumpe2fs：查看分区的文件系统详情！
5. lsusb：查看当前环境下的所有usb设备！
6. dd：转换和拷贝文件、文件系统
7. fdisk：建立分区表（首先格式化磁盘），查看磁盘信息！
8. lsof：查看当前已经打开的文件或者已打开文件和进程的对应关系
9. 可以配置多个默认路由default，只是这些默认路由对应的网关和网卡必须相异！
10. cdecl：用于查看变量定义语法说明！用法：

* 先进入cdecl模式：**cdecl**
* **explain** 声明 //根据声明得到解释
* **declare** <name> **as** <english> //根据解释得到声明

1. C++：参数默认值只能使用在直接地函数调用中。当使用函数指针取地址的时候，默认参数是不可见的。 //到底是默认参数不可见还是其默认值不可见？
2. 返回引用类型都为左值，例如int &func(); func()=16;（可以如此理解：一般返回值返回的是中间变量/对象，是个瞬态量。而返回引用返回的是变量本身其本身不是瞬态量）
3. 掌握目标文件处理工具：

* **readelf**：查看elf文件信息
* **hexdump**：二进制文件查看工具
* **objdump**：显示目标文件信息
* **objcopy**：将目标文件的一部分或者全部内容拷贝到另外一个目标文件中，或者实现目标文件的格式转换
* **nm**：列出文件中的符号 //什么符号？我用grep不能找吗？我想应该是nm可以格式化输出符号和和其相关的东西。
* **strings**：打印文件中可打印的字符串
* **strip**
* **addr2line**：addr2line （它是标准的 GNU Binutils 中的一部分）是一个可以将指令的地址和可执行映像转换成文件名、函数名和源代码行数的工具。**经常用来查看core dump问题出现的行号，**不过这项功能使用gdb调试core文件很容易得到。  
  Eg：**addr2line** **-e** FILE\_NAME IP **-f** -e用来指出异常文件名（FILE\_NAME）。IP是dmesg查到的异常字段
* **ar**：静态库制作工具  
  eg：**ar** **-rcu** libmain.a A.o B.o C.o //将A.o&B.o&C.o打包成libmain.a  
  eg: **ar** **-x** libtest.a //将库文件还原为.o文件
* **as**：汇编语言编译器
* **size**：显示目标文件中各个段的大小，当没有输入文件名是默认为a.out
* **ranlib**：更新静态库的符号表，用于解决ar追加.o文件后因为符号表没有更新而产生的问题。
* **ld**：将目标文件链接起来
* **ldconfig**:动态库管理命令。

ldconfig几个需要注意的地方：

* 往/lib和/usr/lib里面加东西，是不用修改/etc/ld.so.conf的，但是完了之后要调一下ldconfig，不然这个library会找不到。
* 想往上面两个目录以外加东西的时候，一定要修改/etc/ld.so.conf，然后再调用ldconfig，不然也会找不到。
* 比如安装了一个[mysql](http://man.linuxde.net/mysql)到/usr/local/mysql，mysql有一大堆library在/usr/local/mysql/lib下面，这时就需要在/etc/ld.so.conf下面加一行/usr/local/mysql/lib，保存过后ldconfig一下，新的library才能在程序运行时被找到。
* 如果想在这两个目录以外放lib，但是又不想在/etc/ld.so.conf中加东西（或者是没有权限加东西）。那也可以，就是[export](http://man.linuxde.net/export)一个全局变量LD\_LIBRARY\_PATH，然后运行程序的时候就会去这个目录中找library。一般来讲这只是一种临时的解决方案，在没有权限或临时需要的时候使用。
* ldconfig做的这些东西都与运行程序时有关，跟编译时一点关系都没有。编译的时候还是该加-L就得加，不要混淆了。
* 总之，就是不管做了什么关于library的变动后，最好都ldconfig一下，不然会出现一些意想不到的结果。不会花太多的时间，但是会省很多的事。
* 再有，诸如libdb-4.3.so文件头中是会含有库名相关的信息的（即含“libdb-4.3.so”，可用strings命令察看），因此仅通过修改文件名以冒充某已被识别的库（如libdb-4.8.so）是行不通的。为此可在编译库的Makefile中直接修改配置信息，指定特别的库名。
* **ldd:** 查看二进制程序的共享库依赖!（查看程序运行链接的动态库）
* **gprof**： GNU gprof是一款linux平台上的程序分析软件（unix也有prof)。借助gprof可以获得C程序运行期间的统计数据，例如每个函数耗费的时间，函数被调用的次数以及各个函数相互之间的调用关系。gprof可以帮助我们找到程序运行的瓶颈，对占据大量CPU时间的函数进行调优（gprof统计的只是CPU的占用时间，对I/O瓶颈貌似无能为力，耗时甚久的I/O操作很可能只占据极少的CPU时间）。

gprof的使用非常简单，在编译链接的时候加上"-pg"选项，然后按照正常方式运行程序，如果程序正常退出，一个名为gmon.out将会产生。使用gprof可查看gmon.out中的统计结果

* **cpp**：c preprocessor的缩写。
* **c++filt**：  
  C/C++语言在编译以后，函数的名字会被编译器修改，改成编译器内部的名字，这个名字会在链接的时候用到。如果用[backtrace](http://www.gnu.org/software/libc/manual/html_node/Backtraces.html)之类的函数打印堆栈时，显示的就是被编译器修改过的名字，比如说\_Z3foov 。 那么这个函数真实的名字是什么呢？
* 每个编译器都有一套自己内部的名字，这里只是针对linux下g++而言。  
  以下是基本的方法:  
  每个方法都是以\_Z开头，对于嵌套的名字（比如名字空间中的名字或者是类中间的名字,比如Class::Func）后面紧跟N ， 然后是各个名字空间和类的名字，每个名字前是名字字符的长度，再以E结尾。(如果不是嵌套名字则不需要以E结尾)
* 比如上面的\_Z3foov 就是函数foo() , v 表示参数类型为void .  
  又如N:C:Func 经过修饰后就是 \_ZN1N1C4FuncE, 这个函数名后面跟参数类型。 如果跟一个整型，那就是\_ZN1N1C4FuncEi
* 另外在linux下有一个工具可以实现这种转换，这个工具是[C++](http://lib.csdn.net/base/cplusplus)filt , 注意不是c++filter.
* $ c++filt \_ZN1N1C4FuncEi
* N::C::Func(int)

1. gdbserver必须掌握
2. 标准库定义了三个顺序容器适配器：stack，queue和priority\_queue.适配器（adaptor）是标准库中的一个通用概念。容器，迭代器和函数都有适配器。本质上一个迭代器是一种机制，能使某种事物的行为看起来像另外一种事务一样。一个容器适配器接受一种已有的容器类型使其行为看起来像一种不同的类型。
3. 根据可执行文件如何得到源文件文件名：readelf file\_name -s | grep ABS 。或者readelf file\_name -a | grep ABS（如果源文件的文件名为汉字则无法使用grep显示）
4. Linux下静态库制作步骤：

* 将源文件生成目标文件： gcc source\_file -c object\_file.o
* 将.o文件打包生成库： ar cr libtarget.a objectfile0 objectfile1 …

1. Linux下动态库的制作步骤：

* gcc/g++ file0 file1 file2 … -fPIC -shared -o libtarget.so //其中file文件是源文件，而不是目标文件
* shared：用来指定生成动态库
* -fPIC：表示编译为位置独立的代码；

不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的；

1. 大小不一样的数组即使类型一样也属于不同类型。另外对于形参，如果形参是引用类型则实参数组不会被转换为指针（这也算是指针和引用的区别之一了）
2. 计算机启动流程：

* 加载BIOS：包含了cpu信息，设备启动顺序信息，硬盘信息，内存信息，时钟信息，pnp特性等。在此之后计算机心里就有数了，直到应该去读取那个硬件设备了。
* 读取MBR：磁盘上第0磁道第1扇区称为MBR（Master Boot Record），其大小为512字节，里面存放了预启动信息，分区表信息。系统找到BIOS指定的硬盘MBR后，将其复制到某一指定的物理地址处，其实复制的内容就是Boot Loader！
* Boot Loader：初始化硬件设备，建立内存空间映射图。从而将系统的软硬件环境带到一个合适的状态以便为最终调用操作系统内核做好一切准备。
* 加载内核：根据Boot Loader设定的内核映像所在路径，系统读取内核映像，并进行解压缩操作。

1. 制作交叉编译工具链需要使用bash的shell版本，另外对目标的命名也有要求！
2. Linux下vmstat：记录关于处理器，内存，页，块io， traps，磁盘和cpu活动等的基本信息！
3. zImage：大小不超过512K，和bzImage一样都是gzip压缩的。而uImage是uboot专用格式的内核映像文件，uImage和zImage比较仅仅是增加了一个64字节的头，其他完全一样。
4. .cpp生成.o文件的阶段使用的是gcc编译器，指定编译版本-std=c++11也是在这个阶段完成的，包括其他的参数，而由.o生成可执行文件的过程只完成连接的功能！
5. Gdb下list的使用：

* **list** lineno //显示第lineno附近的源码
* **list** func //显示函数func附近的源码
* **list** //显示上次list结束行开始的源码
* **list** - //向前显示
* **list** fist,last //显示first到last之间的源码
* **list** ,last //显示当前行到last之间的源码
* **list** + //向后显示源码

1. gdb中的watch功能很重要，用来监视某一变量或者某一指定内存中的状态变化！
2. Linux下的目录结构：

* /bin ：包含二进制可执行文件（命令）。
* /sbin ：同样包含二进制可执行文件。但是其通常由系统管理员使用，对系统进行维护 //可以理解为super bin！
* /etc ：配置文件，包含所有程序所需的配置文件。也包含了用于启动/停止某个程序的启动和关闭shell脚本eg：/etc/resolv.conf、/etc/logrotate.conf
* /dev ：设备文件，这些设备包括终端设备、usb设备或连接到系统的任何设备
* /proc ：进程信息，包含系统进程的相关信息。这是一个虚拟的文件系统，包含有关正在运行的进程的信息。
* /var ：变量文件，这个目录下可以找到内容可能增长的文件。包括日志文件，数据库文件，电子邮件，打印队列等等
* /tmp ：临时文件，包含系统和用户创建的临时文件。当系统重新启动后，该目录下的文件都将被删除。
* /usr ：用户文件，包含二进制文件，库文件，文档和二级程序的源代码。
* /home ：所有用户用home目录存放其个人文档
* /boot ：引导加载程序文件，包含引导加载程序相关的文件。
* /lib ：系统库包含支持/bin和/sbin下二进制文件的库文件。
* /opt ：可选的附加应用程序，opt代表可选的。包含从个别厂商的附加应用程序
* /mnt ：挂载目录，临时安装目录，系统管理员可以挂载文件系统
* /media ：可移动媒体设备，用于挂载可移动设备的临时目录。Eg：挂载CD-ROM的/media/cdrom,挂载软盘驱动器的/media/floppy
* /srv ：服务数据，srv代表服务。

1. sed和awk命令必须掌握
2. 分治策略：

* **分解**：将问题划分为一些子问题，子问题的形式与原问题一样，只是规模更小。
* **解决**：递归地求解子问题，如果子问题的规模足够小，则停止递归直接求解。
* **合并**：将子问题的解组合成原问题的解。

1. 递归的理解是我国现阶段面临的主要问题。Hanoi问题应该理解一千遍！
2. 递归最重要的一点就是假设子问题已经解决了，现在要基于已经解决的子问题来解决当前问题。
3. SIGKILL和SIGSTOP信号不能被忽略，也不能被捕捉。
4. Vim用法：

* vim -on file1 file2 … //上下分栏打开n个文件
* vim -On file1 file2 … //左右分栏打开n个文件
* ctrl + w + h/j/k/l //切换到前下上后栏
* ctrl + w + 箭矢 //同上
* ctrl + w + c //松开ctrl再按c否则无效,关闭除当前光标所在窗口的其他窗口!
* ctrl + w + n+ //上下扩大n行
* ctrl + w + n- //上下缩小n行

1. 下载arm交叉编译工具的地址

* <https://launchpad.net/gcc-arm-embedded/+download> //裸板用计！
* <https://launchpad.net/gcc-arm-embedded> //裸板用计！
* <https://www.linaro.org/downloads/> //arm（含高通）平台专用

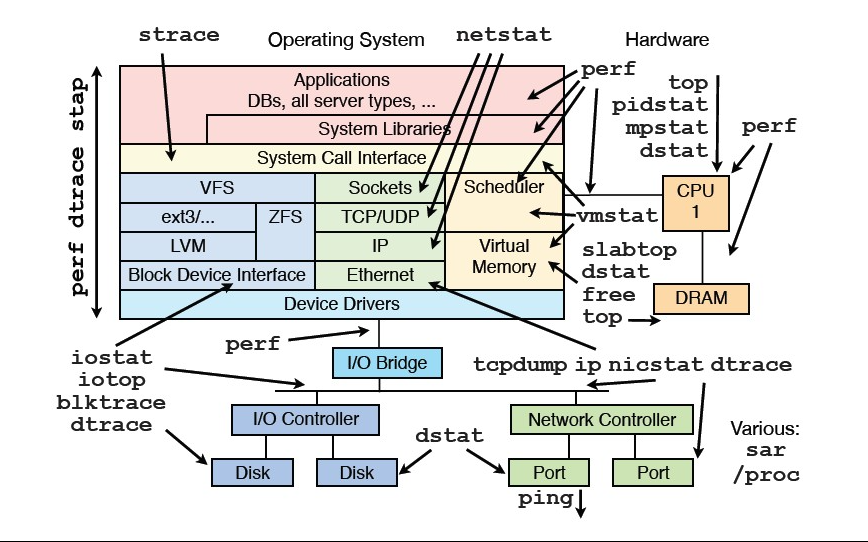
1. Vim下在行首插入行号: :%s /^/\=line(“.”)/g
2. linux下可以使用backtrace函数簇获取运行时堆栈信息.其细节如下:

* backtrace:获取当前线程的调用堆栈
* backtrace\_symbols:将backtrace获取的信息转化为一个字符串数组.
* Backtrace\_symbols\_fd:功能同上,只是信息被写入fd中.

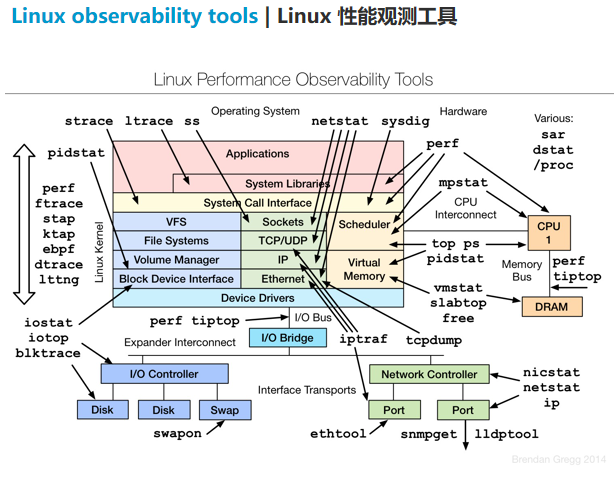
1. catchsegv:该命令用来捕获段错误的出错信息!
2. 根据dmesg查找程序崩溃原因:

* dmesg //打印崩溃信息
* addr2line -e filename ip -f //ip是dmesg打印的ip字段地址.-e用来指定产生异常的可执行文件名，-f用来显示异常处所在的函数。

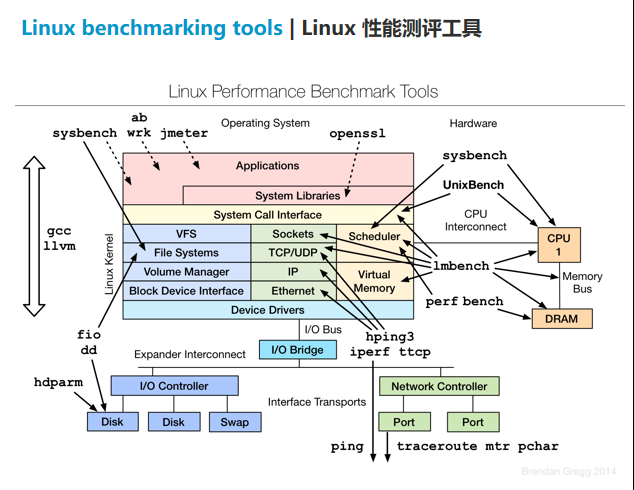
1. Linux:



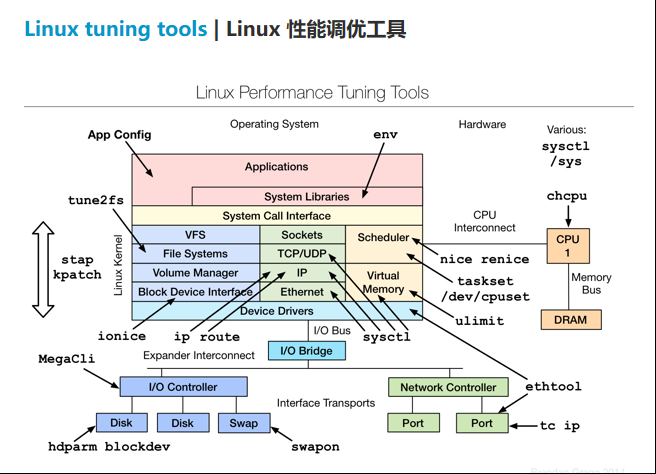
性能观测工具：



性能测评工具：



性能调优工具：



更多内容访问：http://www.brendangregg.com/linuxperf.html

1. linux下当连接断开，还发数据的时候，不仅send()的返回值会有反映，而且还会向系统发送一个异常消息，如果不作处理，系统会出BrokePipe，程序会退出，这对于服务器提供稳定的服务将造成巨大的灾难。为此，send()函数的最后一个参数可以设MSG\_NOSIGNAL，禁止send()函数向系统发送异常消息
2. Qt中qmake -project命令默认生成app(应用),如果想生成动态库只需将”app”字段改为”lib”即可, 也可以添加上:QT -= gui优化(因为生成库不需要图形接口)!
3. 信号槽和事件有时是冲突的，例如鼠标按压事件会覆盖点击信号
4. Qt提供了处理事件的五种方式:

* 重新实现特定的事件处理器:重新实现mousePressEvent(),keyPressEvent(),paintEvent()这些qt事先已定义的事件处理器是进行事件处理的最简单方式.
* 重新实现QObeject::event()函数: 通过重新实现event函数, 可以在事件到达特定的事件处理器之前截获并处理他们. 这种方法用来覆盖已定义事件的默认处理方式, 也可以用来处理Qt中尚未定义特定事件处理器的事件. 当重新实现event()函数时,如果不进行事件处理,则需要调用基类的event()函数.
* 在QObject中注册事件过滤器 : 如果对象使用installEventFilter()函数注册了事件过滤器, 目标对象中的所有事件将首先发给这个监视对象的eventFilter()函数.
* 在QApplication中注册事件过滤器 : 如果一个事件过滤器被注册到程序中唯一的QApplication对象, 应用程序中所有对象里的每一个事件都会在他们被送达到其他事件处理器前,首先抵达这个evnetFilter()函数. 他可以用来处理一些通常被QApplication忽略的事件, 如发送给失效窗口的鼠标事件等.
* 继承QApplication并重新实现notify()函数 : Qt调用QApplication来发送一个事件. 重新实现notify()函数是在事件过滤器得到所有事件之前获得他们的唯一方法.但事件过滤器使用更为便利, 因为可以同时有多个事件过滤器, 而notify()函数只有一个.

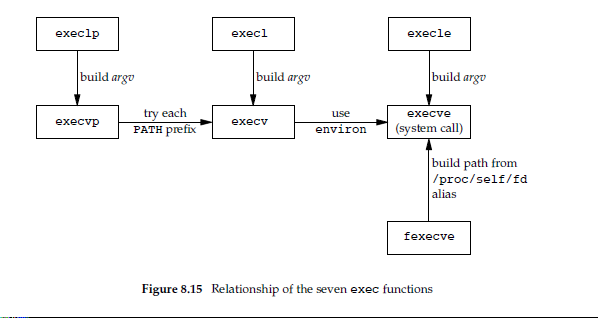
1. UUID（全局唯一标识符）：UUID是一个标帜你系统中的存储设备的字符串，其目的是帮助使用者唯一的确定系统中的所有存储设备，不管它们是什么类型的。它可以标识DVD驱动器，USB存储设备以及你系统中的硬盘设备等。UUID在/etc/fstab文件中指定，被指定的磁盘会被系统启动，如果想添加一个外部磁盘只需在这里添加一个item，重启之后自动挂载！
2. 子网掩码的作用是：划分网络地址和主机地址！仅仅如此！划分多个网段的子网掩码配置，判断多个ip是否从属于一个网段需要掌握
3. 如果没有指定库路径， 则编译不会报错， 运行的时候才会报错， 提示dlopen失败。
4. Qt的对象模型必须掌握，这涉及到进程运行的流程和某些情况下析构发生错误的原因。详见学习之路2（10）
5. find . -mtime -n -type f -print //查找n\*24小时之内被修改的文件，关于find按时间搜索文件的使用应该查看man手册，功能远远不止这么简单！
6. 操作-字符开头的文件：加参数—(两个减号)即可！
7. Mmap函数必须掌握。Mmap函数的功能是把文件的内容映射到内核，然后返回一个指针，通过memcpy等的操作直接可以改变文件内容，效率比起read/write高很多！但是我们无法更改其长度（难道还不能追加写入？那要mmap干什么！有待验证！），因为要映射的长度在调用mmap时就已经确定了！
8. 共享内存创建流程及使用说明

* shmget() //创建共享内存
* shmat() //将共享内存映射到进程地址空间
* shmdt() //断开共享内存和进程的联系
* shmctl() //可以在这里删除共享内存！

不删除共享内存进程重启映射的还是这块内存。以上是system v平台的实现Posix也有共享内存，原理差不多只是其是对文件描述符的操作，使用mmap进行控制，效率上不如System V

1. 静态成员变量可以在类内声明而不能定义(即不能初始化，初始化必须发生在类的外部)，在类的外部可以定义和初始化。C++11提供了关键字constexpr使得可以类内初始化！
2. C++使用构造函数对数据成员初始化时，初始化顺序是按照变量声明的顺序的，而不是按照参数初始化列表的前后顺序！
3. 类成员赋值/初始化三种情况下必须使用参数初始化列表：

* 如果有一个类成员，它本身是一个类或者是一个结构，而且这个成员它只有一个带参数的构造函数，而没有默认构造函数，这时要对这个类成员进行初始化，就必须调用这个类成员的带参数的构造函数，如果没有初始化列表，那么他将无法完成第一步，就会报错。
* 当类成员中含有一个const成员时
* 当类成员中含有一个引用时

1. malloc分配内存并不一定是连续的，通常是链表连起来的内存区！calloc函数也能分配内存，而且内存会被初始化，只是这种分配是在栈帧上进行的，一旦外围函数终止则内存会自动释放（存在的意义呢？）。
2. 父子进程共享file table entry！（一旦发生了写操作file table entry会复制吗？）
3. 

wait函数的原理是父进程捕获子进程终止/停止运行时发出的SIGCHLD信号，任何进程结束都会发出该信号，只是这个信号默认是被ignore的！(Whenever a process terminates or stops, the SIGCHLD signal is sent to the parent. By default, this signal is ignored, so the parent must catch this signal if it wants to be notified whenever a child’s status changes.The normal action in the signal-catching function is to call one of the wait functions to fetch the child’s process ID and termination status.)

1. 查看各信号的含义使用：man 7 signal（不要看中文版手册/注解）
2. 子进程不会继承父进程的子线程！子进程会继承父线程的各种锁，不过一旦我们调用exec函数则这些锁便失效了！
3. Pread和pwrite函数将定位和read/write作为一个原子操作完成，避免了文件共享带来的写入和读取谬误。尤其是多线程下！
4. readv/writev函数是两个有情趣的函数（分散读，聚集写）！
5. Linux下L和I分别用来指定链接库和头文件的绝对路径！
6. GCC专有语法中允许在结构体中使用匿名结构体或共用体，使用时可以当匿名结构体或共用体不存在，直接使用其成员！
7. 消息队列中msgsnd函数的para1第一枚成员值必须大于0，para2是para1参数第二枚成员的长度而不是para1总长度！非常易错！
8. Openat函数和open很多情况下功能是相同的，只是相比于open他有两个优点：

* 在多线程下，使用open则所有的线程只能工作在一个目录下，而openat可以工作于不同目录！
* It provides a way to avoid time-of-check-to-time-of-use(TOCTTOU)errors.

1. od //以某种具体格式显示文件内容如字符格式：od -c test.c
2. 第一次执行某个操作后相关的数据会保存在系统cache中，下次运行相同的程序则直接从该内存中读取该cache而不需要从磁盘上，这是为什么第二次会比第一次顺畅的原因！
3. 子进程会继承父进程的文件描述符及其特性，子进程和父进程共用file table entry结构体，所以一旦子进程改变了offset则父进程的offset也会被改变，想改变这种窘境只需要先fork再打开文件！
4. Vfork和fork的不同：

* Copy-on-write！
* Child run first until the child calls exec of exit！

1. Linux下库的制作：

动态库：

* gcc -c source.c-o file.o //源文件制作成目标文件(\*.c 🡪 \*.o)
* gcc -o libmylib.so file.o -fPIC -shared //根据.o文件生成库文件
* gcc -o target sourcefile.c -L`pwd` -lmylib -Wl,-rpath=`pwd` //编译源文件sourcefile.c生成可执行文件target，-fPIC和-shared是生成动态库的特殊指令，-L用来指定mylib的路径（假设在当前路径），-lmylib没有后缀。以上能够正确编译，但是不能正常运行，因为运行时在系统默认路径下找不到动态库，解决此问题的方式有两种：
* 编译可执行文件时指定动态库路径（如上），只需添加参数：-Wl,rpath=`pwd`(假设mylib就在当前路径下)rpath是只能指定动态链接库路径！
* 通过LD\_LIBRARY\_PATH导入路径

也可将第一步和第二部合并：直接由源文件生成库文件不需要目标文件中继！

静态库：

* gcc souce.c -c //和动态库不同，生成目标文件的步骤不能少！
* ar cr libmylib.a source.o
* gcc -o target sourcefile.c -L`pwd` -lmylib //假设静态库存在于当前目录

第三步可拆分如下：

* gcc sourcefile.c -c
* gcc -o target sourcefile.o -L`pwd` -lmylib

1. unset命令用来清除指定的环境变量！ eg:unset LD\_LIBRARY\_PATH
2. fork与信号：

当一个进程调用fork时，因为子进程在开始时复制父进程的存储映像，信号捕捉函数的地址在子进程中是有意义的，所以子进程继承父进程的信号处理方式（When a process calls fork, the child inherits the parent’s signal dispositions）。  
特殊的是exec，因为exec运行新的程序后会覆盖从父进程继承来的存储映像，那么信号捕捉函数在新程序中已无意义，所以exec会将原先设置为要捕捉的信号都更改为默认动作。

1，fork后子进程会继承父进程的**信号屏蔽字**，再继续exec后仍会继承这个信号屏蔽字。同样地，直接调用system后子进程也会继承父进程的信号屏蔽字。

2，fork后子进程会继承父进程的**信号处理设置，**再继续exec后就不会继承这个信号处理设置了。

3，fork后子进程会继承父进程的**控制终端**，且子进程在父进程的进程组和会话组中；再继续exec后仍会继承这个控制终端，仍在父进程的进程组和会话组中。同样地，调用system后子进程会继承父进程的控制终端，且子进程在父进程的进程组和会话组中。

4，Ctrl+c产生的SIGINT信号会发送给父进程、fork后的子进程以及继续exec的子进程；同样地，也会发给system调用运行的子进程。

当一个进程调用fork时，因为子进程在开始时复制父进程的存储映像，信号捕捉函数的地址在子进程中是有意义的，所以子进程继承父进程的信号处理方式。  
特殊的是exec，因为exec运行新的程序后会覆盖从父进程继承来的存储映像，那么信号捕捉函数在新程序中已无意义，所以exec会将原先设置为要捕捉的信号都更改为默认动作。

1. linux下效率：

* record lock > XSI semaphore
* POSIX semaphore >> XSI semaphore

1. 对于不同para0和不同para1，ftok可能产生相同key！
2. 对于IPC，主机和客户端建立连接有三种方式：

* 服务端（假设服务端创建了IPC）将key写入文件，客户端读取文件（这要求客户端和主机端具有匹配的文件系统）！或者父子进程间可也！
* 主机端和服务端使用用户自定义key值，比如使用同一个定义了key值的头文件！
* 使用ftok，para1是int型，但只有低八位可用！ 并且正如上面所言不同para0和para1可能产生相同key！

1. 效率：message queue > 全双工管道 > socket！
2. Git常用命令：

* git diff file //比较最后一次提交和当前文件的异同
* git diff version –- file //比较版本version和当前文件的异同
* git reflog [--pretty=oneline] //查看所有版本
* git reset –hard HEAD^ //版本回退，回退到上一个版本，^的个数表示回退的级数，也可用HEAD~n表示回退100级！注意和checkout区分！
* git reset –hard commit\_id //功能同上，以SHA码代替HEAD
* git checkout [HEAD~n] – file //将工作区的修改恢复到n级提交的状态，也就是说恢复的来源是版本库
* 已添加到缓存区恢复方式！
* git reset HEAD file //撤销提交到暂存区的修改！
* git checkout – file //撤销工作区修改（尚未提交到暂存区）/从版本库回复到最新file

git checkout HEAD~n – file //从版本库恢复到n级提交！

* git rm file //删除版本库中的file必须和下面命名配合
* git commit [-m] file //工作区的文件也会被删除
* git rm –cache file //删除暂存区的file，不删除工作区file！

1. git分支管理：

* git checkout -b BRANCH //创建分支BRANCH并进入之

等价于如下两步：

* git branch BRANCH //创建分支BRANCH
* git checkout BRANCH //进入分支BRANCH
* git branch //查看所有分支
* git merge BRANCH //将BRANCH合并到当前分支
* git branch -d BRANCH //删除分支BRANCH
* git log –graph [–pretty=oneline –abbrev-commit] //查看分支合并情况
* git merge –no-ff [-m ‘statesment’] BRANCH //合并分支并禁用fast forward！

1. Git提交中文乱码，执行以下命令可破解： git config –global core.quotepath false
2. Delete一个指向nullptr的指针不会崩溃！网友语：delete会对指针进行检查，当发现指针被指向nullptr时即会跳过delete！
3. 执行gdb时显示源文件两种方式(gdb版本7.0及以上)：

* gdb -tui executable-file //启动gdb时指定显示源文件！
* focus //启动gdb后执行显示源文件！用法同下
* layout //同上！
* next //显示寄存器和汇编！
* prev //显示寄存器和源文件！
* ctrl+x a //关闭layout！
* Ctrl+x 1 //单串口模式！
* Ctrl+x 2 //双窗口模式！
* Enter键 //在各种显示之间切换！

1. gdb知识：

* 开启gdb会产生一枚进程，如果运行调试程序会产生新的进程
* **path**指定运行路径，**directory**指定源文件搜索路径！
* **set args**参数传递给main函数的argc和argv！进入main函数后就无效了！
* **show args**和**info args**是不同的！
* **Info** **program** //查看进程号，进程状态，暂停原因等！
* **condition** //用来更改停止点（断点breakpoint/监测点watch）条件！
* **b** line/function **if** condition **commands** statements **end** //设置条件断点！
* **until**/**u** 执行循环直至结束！
* **Info source** //显示源文件相关信息！

1. Gdb生成的core文件调试过程中依然会读取源文件。也就是说生成core文件的源文件一旦修改可能定位的行号就不准确了！
2. 普通数组也可以使用迭代器处理，使用begin(array)和end(array)分别获取数组array的首位位置，通过迭代器的加减运算移动之！
3. 早期c++版本允许结果为负值的商（/操作）向上或向下取整，c++11新标准规定商一律向0取整！
4. 浮点数不能参与%运算！
5. 自增/自减运算符前置的效率高于后置！
6. 大多数用到数组的表达式中，数组自动转换成指向数组首元素的指针，当数组被用作decltype关键字的参数，或者作为取地址符、sizeof及typeid等运算符的运算对象时，上述转换不会发生，同样的，如果用一个引用来初始化数组，上述转换也不会发生。
7. 单字节组合多字节时不能使用+只能使用|，除非使用中间变量！
8. Gcc编译指定宏(定义的宏会被传递到被编译的源文件中)：

* -Dmacro=string 等价于在头文件中定义：#define macro string
* -DTRUE=true 等价于在头文件中定义：#define TRUE true
* -Dmacro 等价于在头文件中定义：#define macro 1
* -DLINUX 等价于在头文件中定义：#define LINUX 1

1. 标准IO的缓存特性可以修改(setbuf和setvbuf函数)
2. 并不是所有命令使用sudo都万无一失，比如ubuntu系统版本升级命令:

$ do-release-upgrade

如果加上sudo则执行失败！

1. Vim下查找字符大小写敏感：

* :set ignorecase //不区分大小写
* :set noignorecase //区分大小写！

1. Static修饰的函数：

* 在头文件中用static声明则源文件中的定义可以不带static
* 在头文件中声明不带static，则源文件中声明也不能使用static，即使在源文件中用static重新声明也不行！

1. Gcc参数：

* -I PATH //指定头文件路径，I和PATH之间也可以没有空格，使用这个参数则源文件中的头文件路径不需要指定，这个过程发生在编译阶段（即生成.o的过程中！），所以makefile中不能使用自动推导，必须包含.o文件的生成过程！
* -L PATH //指定链接库的路径，该路径的优先级高于系统路径
* -lfile //链接库libfile.so/libfile.a。默认链接动态库使用-static链接静态库！