# 目录相关命令

## 命令行

pwd 打印当前工作目录的名字

cd

mkdir

rmdir

ls

touch

cp

mv

rm

cat

more

less

head

tail

du 查看某个目录大小

tree 以树状形式查看指定目录内容

df 检测文件系统的磁盘空间占用和空余情况

find /home -name “a.txt”

grep -r "hello，world" /home/itcast

ps aux | grep 10001 管道，左边的输出作为右边的输入

tar -cvf 创建归档文件

tar -xvf 解除归档文件(还原)

chmod 777 test/ -R

chown root txt

ln 源文件 链接文件 创建硬链接

ln -s 源文件 链接文件 创建软链接

## c函数

open

close

read

write

lseek

stat 查看文件状态信息

access 测试文件是否有某个属性

chmod

chown

link 硬链接

symlink 软链接

unlink

rename

dup 复制一个文件描述符

dup2 可以指定具体的文件描述符

fcntl(int fd, int cmd, ... /\* arg \*/)

fcntl函数有5种功能：

1) 复制一个现有的描述符（cmd=F\_DUPFD）

2) 获得／设置文件描述符标记(cmd=F\_GETFD或F\_SETFD)

3) 获得／设置文件状态标记(cmd=F\_GETFL或F\_SETFL)

4) 获得／设置异步I/O所有权(cmd=F\_GETOWN或F\_SETOWN)

5) 获得／设置记录锁(cmd=F\_GETLK, F\_SETLK或F\_SETLKW)

getcwd 获取当前进程的工作目录

chdir 修改当前进程(应用程序)的路径

opendir

closedir

readdir

# 进程管理相关命令

## 命令行

ps aux 显示所有进程及其具体状态

top 用来动态显示运行中的进程

kill [-signal] pid 信号值从0到15，其中9为绝对终止

kill all 通过进程名字杀死进程

## c函数

getpid

getppid

getpgid

fork

wait 等待任意一个子进程结束，如果任意一个子进程结束了，此函数会回收该子进程的资源。

waitpid 等待指定的子进程终止，如果子进程终止了，此函数会回收子进程的资源。

exec 在进程中执行别的进程

pipe

mkfifo

mmap 一个文件或者其它对象映射进内存

munmap 释放内存映射区

kill(pid\_t pid, int sig) 给指定进程发送指定信号(不一定杀死)

raise(int sig) 给当前进程发送指定信号(自己给自己发)

abort 给自己发送异常终止信号

alarm(unsigned int seconds) 定时闹钟

//将set集合置空

int sigemptyset(sigset\_t \*set);

//将所有信号加入set集合

int sigfillset(sigset\_t \*set);

//将signo信号加入到set集合

int sigaddset(sigset\_t \*set, int signo);

//从set集合中移除signo信号

int sigdelset(sigset\_t \*set, int signo);

//判断信号是否存在

int sigismember(const sigset\_t \*set, int signo);

signal(int signum, sighandler\_t handler) 信号处理函数，handler是程序入口sigaction

getsid 获取进程所属的会话ID

setsid 创建一个会话，并以自己的ID设置进程组ID，同时也是新会话的ID。调用了setsid函数的进程，既是新的会长，也是新的组长。

pthread\_mutex\_init

pthread\_mutex\_destroy

pthread\_mutex\_lock

pthread\_mutex\_unlock

pthread\_rwlock\_init

pthread\_rwlock\_destroy

pthread\_rwlock\_rdlock

pthread\_rwlock\_wrlock

pthread\_rwlock\_unlock

pthread\_cond\_init

pthread\_cond\_destroy

pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex) 如果cond条件不满足，阻塞并立刻释放mutex；如果条件满足，则继续。

pthread\_cond\_signal 唤醒至少一个阻塞在条件变量上的线程

pthread\_cond\_broadcast 唤醒全部阻塞在条件变量上的线程

sem\_init

sem\_destroy

sem\_wait P操作，会阻塞

sem\_trywait P操作，不会阻塞，立刻返回

sem\_post V操作

# gcc/g++工作流程

第一步: 预处理（头文件展开，宏替换，注释去掉）

gcc -E 1hello.c -o 1hello.i

第二步: 编译，生成汇编文件（c文件->汇编文件）

gcc -S 1hello.i -o 1hello.s

第三步: 汇编，生成目标代码（汇编文件->二进制文件）

gcc -c 1hello.s -o 1hello.o

第四步: 链接，生成可以执行文件（将函数库中相应的代码组合到二进制（目标）文件中）

gcc 1hello.o -o 1hello

# Makefile

$@: 表示规则中的目标

$<: 表示规则中的第一个条件

$^: 表示规则中的所有条件, 组成一个列表, 以空格隔开,如果这个列表中有重复的项则消除重复项。

wildcard – 查找指定目录下的指定类型的文件

src = $(wildcard \*.c) //找到当前目录下所有后缀为.c的文件,赋值给src

patsubst – 匹配替换

obj = $(patsubst %.c,%.o, $(src)) //把src变量里所有后缀为.c的文件替换成.o

**伪目标声明:** **.PHONY:clean**

声明目标为伪目标之后，makefile将不会该判断目标是否存在或者该目标是否需要更新。

SRC=$(wildcard \*.c)

OBJS=$(patsubst %.c, %.o, $(SRC))

TARGET=test

$(TARGET):$(OBJS)

gcc $(OBJS) -o $(TARGET)

%.o:%.c

gcc -c $< -o $@

.PHONY:clean

clean:

rm -rf $(OBJS) $(TARGET)

# 静态链接和动态链接

## 静态链接

由链接器在链接时将库的内容加入到可执行程序中。优点是对运行环境的依赖性较小，具有较好的兼容性，缺点是生成的程序比较大，在装入内存时会消耗更多的时间，库函数有了更新，必须重新编译应用程序。

## 动态链接

连接器在链接时仅仅建立与所需库函数的之间的链接关系，在程序运行时才将所需资源调入可执行程序。优点是在需要的时候才会调入对应的资源函数，有着较小的程序体积，缺点是严重依赖动态库，不能独立运行。

前面我们编写的应用程序大量用到了标准库函数，系统默认采用动态链接的方式进行编译程序，若想采用静态编译，加入-static参数。

# gdb

## 生成调试信息

gcc -g hello.c -o hello

g++ -g hello.cpp -o hello

## 启动gdb

* 启动gdb：gdb program

program 也就是你的执行文件，一般在当前目录下。

* 设置运行参数

set args 可指定运行时参数。（如：set args 10 20 30 40 50 ）

show args 命令可以查看设置好的运行参数。

* 启动程序

run： 程序开始执行，如果有断点，停在第一个断点处

start： 程序向下执行一行。

## 断点操作

### 简单断点

break 设置断点，可以简写为b

* b 10 设置断点，在源程序第10行
* b func 设置断点，在func函数入口处

### 多文件设置断点

C++中可以使用class::function或function(type,type)格式来指定函数名。如果有名称空间，可以使用namespace::class::function或者function(type,type)格式来指定函数名。

* break filename:linenum -- 在源文件filename的linenum行处停住
* break filename:function -- 在源文件filename的function函数的入口处停住
* break class::function或function(type,type) -- 在类class的function函数的入口处停住
* break namespace::class::function -- 在名称空间为namespace的类class的function函数的入口处停住

### 查询所有断点

info b

info break

i break

i b

## 条件断点

一般来说，为断点设置一个条件，我们使用if关键词，后面跟其断点条件。

设置一个条件断点：

b test.c:8 if Value == 5

## 维护断点

delete [range...] 删除指定的断点，其简写命令为d。

如果不指定断点号，则表示删除所有的断点。range表示断点号的范围（如：3-7）。

比删除更好的一种方法是disable停止点，disable了的停止点，GDB不会删除，当你还需要时，enable即可，就好像回收站一样。

* disable [range...] 使指定断点无效，简写命令是dis。

如果什么都不指定，表示disable所有的停止点。

* enable [range...] 使无效断点生效，简写命令是ena。

如果什么都不指定，表示enable所有的停止点。

## 调试代码

* run 运行程序，可简写为r
* next 单步跟踪，函数调用当作一条简单语句执行，可简写为n
* step 单步跟踪，函数调进入被调用函数体内，可简写为s
* finish 退出进入的函数
* until 在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体,可简写为u。
* continue 继续运行程序，停在下一个断点的位置，可简写为c
* quit 退出gdb，可简写为q

## 数据查看

### 查看运行时数据

print 打印变量、字符串、表达式等的值，可简写为p

p count 打印count的值

## 自动显示

你可以设置一些自动显示的变量，当程序停住时，或是在你单步跟踪时，这些变量会自动显示。相关的GDB命令是display。

display 变量名

info display -- 查看display设置的自动显示的信息。

undisplay num（info display时显示的编号）

delete display dnums… -- 删除自动显示，dnums意为所设置好了的自动显式的编号。如果要同时删除几个，编号可以用空格分隔，如果要删除一个范围内的编号，可以用减号表示（如：2-5）

disable display dnums…

enable display dnums…

disable和enalbe不删除自动显示的设置，而只是让其失效和恢复。

## 查看修改变量的值

* ptype width -- 查看变量width的类型

type = double

* p width -- 打印变量width 的值

$4 = 13

你可以使用set var命令来告诉GDB，width不是你GDB的参数，而是程序的变量名，如：

set var width=47 // 将变量var值设置为47

在你改变程序变量取值时，最好都使用set var格式的GDB命令

## 查看栈信息

当程序调用了一个函数，函数的地址，函数参数，函数内的局部变量都会被压入“栈”（Stack）中。

### 调用栈信息

backtrace [n/-n]

bt [n/-n]

* 如果不加n查看所有调用栈信息
* n查看栈顶上n层信息
* -n查看栈底n层信息

frame [n]

f [n]

* 不加n查看当前的堆栈帧信息
* n切换当前的栈，f 0表示切换到栈顶堆栈帧

up <n>

表示向堆栈帧的上面移动n层（这里的向上是指内存中堆栈帧向上（低地址向高地址），实际上是栈顶往栈底的方向），可以不打n，表示向上移动一层。

down <n>

表示向堆栈帧的下面移动n层，可以不打n，表示向下移动一层。

info args

打印出当前堆栈帧函数的参数名及其值。

info locals

打印出当前堆栈帧函数中所有局部变量及其值。

info catch

打印出当前堆栈帧的函数中的异常处理信息。

# 设置文件描述符为非阻塞

//获取原来的flags

int flags = fcntl(fd, F\_GETFL);

// 设置新的flags

flag |= O\_NONBLOCK;

// flags = flags | O\_NONBLOCK;

fcntl(fd, F\_SETFL, flags);