## 其他

### 断言assert （很重要）

assert (a>5); 断言判断成功，程序继续向下进行，断言失败，程序退出

assert只判断一个条件比较好，程序一旦出错就知道是哪里的问题

一般的运用函数之前可以判断一下，x>10程序继续

assert(x>10);

或者进入函数前判断入参

assert(p != NULL);

### const的作用：

1.限定变量为不可修改。

2.限定成员函数不可以修改任何数据成员。

3.const与指针：

const char \*p 表示 指向的内容不能改变。

char \* const p，就是将P声明为常指针，它的地址不能改变，是固定的，但是它的内容可以改变。

定位问题：CI工程上构建的sha256值与自己编译的不一样（etc/profile环境变量未生效）

网卡数量太多，导致查询带内信息失败。网卡缓冲区不够。

用例查询开机失败：通过pty的方式查询，通过vmtools状态查询。

带内信息：通过程序接口获取的信息

带外信息：通过其他方式获取的信息

在while（） 并且循环里面定义的变量在循环外不能访问，它已经释放了。

每轮循环完成后定义的变量就会被析构，下一轮重新定义的。

但是python的循环中的变量在while之外可以访问。

结构体数组初始化struct a[3]

{

{.x=1,.y=2}, //a[0]的初始化

{.x=1,.y=2}, //a[1]的初始化

{.x=1,.y=2}

}

struct file\_operaters var = { 部分初始化

.name = asdsa,

.age = 12,

};

struct stu p = {1,2,3,4};全部初始化，定义同时才可以这样

结构体传参：

函数参数是栈段分配的，传结构体指针因为指针占用的内存小

如果值传递，系统得在栈中分配很大的空间。所以不管你改不改结构体中的内容，都建议传递结构体指针。

参数是栈段分配的

内存使用原则：小栈大堆

位段结构体：

struct data{

uint year:5;

uint month:4;

uint day:5;

}; 给每个参数规定占位，不浪费

结构体默认4字节对齐，位段结构体也是4字节对齐

### Ps公章：

PS:ctrl+j 把色彩范围的复制出来，按住ctrl+点击抠出来的印章，前景色选成红色，alt+delete填充，ctlr+d取消选区

### PCI总线：

外部设备互联总线，链接cpu与树状pci设备（显卡，网卡），PCI主要链接显卡，网卡。

内核启动时，通过pcibus之前的关系枚举pci\_dev，为每个pci设备创建pcidev结构体，注册到pci树中。

类比内核为字符设备创建cdev结构体，注册到设备树。虚拟总线识别后与驱动绑定。

### Ctags用法：

在/usr/include/目录下，执行ctags -R 生成tags 文件

vim ~/.vimrc

set tags+=/usr/include/tags

### linux中重要的文件

开机自动挂载在/etc/fatab设置

mount -t iso9660 -o ro /dev/cdrom /mnt 挂载方式readonly

/proc

linux中的/proc 是虚拟文件系统，里面的文件大多大小都是0，随时刷新，掉电消失

/proc/processN 进程消失则对应的进程消失

对于进程文件夹中：

/proc/processN/cmdline 启动该程序的完整cmd

environ：当前proc的环境变量

exe：软连接到可启动当前程序的可执行文件

fd:当前进程打开的fd 指向实际的文件的软连接

limits:当前进程的资源受限

task:当前进程中各个线程的相关信息

/proc/version ：当前运行的内核

/proc/partitions: 块设备的每个分区的主次设备号，每个分区的block数量

/proc/modules：当前装入内核的所有模块 也可以通过：lsmod查看

lsmod：

Module size used

A 10 B,C,D 可以知道B,C,D依赖A

/proc/mounts 当前挂载的所有fs

/proc/filesystem: 系统支持的fs列表

其他信息

cat /proc/cpuinfo - CPU

cat /proc/interrupts - 中断

cat /proc/ioports - 设备 IO 端口

cat /proc/meminfo - 内存信息(i.e. mem used, free, swap size)

cat /proc/partitions - 所有设备的所有分区

cat /proc/pci - PCI 设备的信息

cat /proc/swaps - 所有 Swap 分区的信息

cat /proc/version - Linux 的版本号

可以通过修改 proc 修改内核中预定的一些变量

文件句柄的最大数量：

echo 65535 > /proc/sys/fs/file-max

修改网络TTL

echo 128> /proc/sys/net/ipv4/ip\_default\_ttl

修改系统中最大的进程数量

echo 65535 > /proc/sys/kernel/pid\_max

修改普通用户的最大 RTC 频率

echo 128 >/proc/sys/dev/rtc/max-user-freq

### linux中重要的结构体

inode结构体:每个文件（包含字符设备文件）都有自己的inode结构体

cdev结构体：内核空间，每个cdev结构体代表一个设备

file结构体：用户空间每打开文件，在vfs（虚拟文件系统）层，都有对应的file结构体，结构体中有个成员指向的就是属于它操作集（）

inode结构体{

.设备号dev\_t

.cdev结构体{

.struct file\_ops

}

}

task\_struct 进程描述符

内含mm\_struct 内存描述符

mm\_struct描述对于当前进程4G内存是怎么分配的

每个进程对应一个task\_struct

查看进程可用资源：ulimite -a

(open files:1024 最多能同时打开这么多文件)

(stack size：栈大小)

进程栈、用户栈一般是8M：初始化时由编译器链接器计算出，但是会动态增加（即增加页表）内核是8K或4k

用户栈：fun运行提供空间

内核栈：保存register的值，头区域存放当前进程的task\_struct进程描述符

线程栈：（轻量化的进程，与其他线程共享mm\_struct中的资源）

线程栈从进程的地址空间中映射出来一段区域，用完就没了，不会动态增加。对于线程mm是共享的但是线程栈不是共享的。

进程与线程不共享内核栈，为啥每个进程需要单独的内核栈呢？

假设进程a，陷入内核态时刚好休眠让出CPU，进程b也刚好陷入内核态，若内核栈只有1个，切换时内核栈要存进程a的数据，b运行内核栈也需要压栈，很可能溢出，丢失数据

### unit

### unit类型

- target unit 模拟运行级别的 xx.target===runlevelx

- swap unit 标识swap设备

service unit文件通常由3部分组成

- unit 定义与unit无关的通用选项，描述unit，行为、依赖

- 1. description

2. after 当前servoce应晚于哪些unit启动

3. requires 依赖其他的units，强依赖：其他unit不激活，自己也不能用

4. wants 依赖其他units 弱依赖

5. conflicts units间的冲突关系，我启动某些unit就不能启动了

- service 特定类型相关选项

- 1. execstart 指明启动unit要运行的脚本的绝对路径

2. execstop 指明停止unit要运行的脚本的绝对路径

- install 定义由systemctl enable 、systemctl disable实现或禁用时用到

### 任务调度

LINUX任务调度，定时任务 crontab---/etc/crontab

/var/log/cron 定时日志

修改完/etc/crontab文件后要重启crond.service服务你的任务才会生效

定时运行的脚本应将屏幕输出重定向到/dev/null-->root sh xx > /dev/null 2>&1

否则每次的输出会新建文件，很快会占完目录的inode节点

```c

crontab -e 添加任务

crontab -i 查询任务

crontab -r 删除任务 执行成功没有输出log，若再执行就会输出

\*no crontab for linux 因为刚才第一次执行时就已经删了

占位符：---

+ 第一个\*：1个小时的第几分钟（0-59）

+ 第二个\*：一天中的第几小时（0-23）

+ 第三个\*：1月中的第几天（1-31）

+ 第四个\*：1年中的第几月（1-12）

+ 第五个\*：一周中的周几（0-7ps:0与7都代表周日）

例子：

每分钟将ls 当前目录的内容放入/tmp/1.txt

\*/1 \* \* \* \* ls /etc/ >> /tmp/1.txt

每分钟

### virsh命令

virsh有很多有用的功能：

生成磁盘镜像qemu-img create -f qcow2 xxx 40G 这个也可以是磁盘镜像，也可以是虚拟机 看你的xml是什么了

热插磁盘：virsh attach-device 虚拟机名字 xxx.xml（此xml中就是普通xml中的device中的描述磁盘的那部分内容）

动态添加：virsh attach-disk 机名 xx.qcow2 vda

热拔磁盘：virsh detach-device 虚拟机名字 xxx.xml

virsh reboot/shutdown virtname

热插网卡：virsh attach-device virtname net.xml

热插cpu：virsh setvcpus 机名 2 --live

热插内存：virsh attach-device virtname mem.xml

热迁移：就是将HOST1中的virtA在运行状态迁移到HOST2上去运行

本质就是将virtA所在的目录在HOST1与HOST2中共享。共享目录必须包含此虚拟机的磁盘，镜像，iso，不然迁移不过去，

HOST2看不到磁盘，iso的位置了就

xml文件中屏蔽:<!-- xxxxxxx --> z中间内容被屏蔽

virsh domiflist virtname:查看虚拟机的mac，网桥

virsh change-media virtname hdb(xml中cdrom叫什么名字) --eject 先将之前的cdrom拔出来

virsh change-media virtname hdb(xml中cdrom叫什么名字) /xx.iso --insert

### LVM逻辑卷管理器

lvm是建立于磁盘与分区之上，FS之下的一个逻辑层。

```c

物理存储介质：/dev/hda,vda,sda,。。。 最底层的存储单元

物理卷（PV）：物理介质硬盘分区。再建立卷组时确定PE最小单元大小，确定后不可更改(fdisk /dev/sda 执行此命令进行分区)

卷组（vg）：至少包含1个物理卷

逻辑卷（LV）：建立在卷组里面，卷组中未分配的空间都可以用于建立新的逻辑卷，最大的好处就是可以动态扩容或缩小，不像物理

硬盘，多大就多大。LVM可以由1或多个VG组成，1个VG也可以创建1或多个LVM。

LE：逻辑区域，是逻辑分区中的可分配的最小单元 === PE，物理区域

fdisk pvcreate vgcreate lvcreate mkfs

/dev/sda介质------>物理分区--------->物理卷--------->卷组---------->逻辑卷------>逻辑分区去挂载

命令：

创建物理卷：pvcreate /dev/sda1 sda1就是个物理卷，

查看物理卷：pvdisplay

卷组：vgcreate lvmdisk -s 16M /dev/sda1 /dev/sdb1 lvmdisk就是卷组名字，用两个物理卷做出来的

创建卷组的时候PE大小就确定了， 这个物理卷可以动态扩容的

逻辑卷：lvcreate 逻辑卷中LE大小==卷组中PE的大小

lvcreate -L 500m vgname

lvcreate -L 5G vgname

在lv中创建文件系统fs：mkfs -t ext2 /dev/lv的path 我们的根文件系统就挂载在这里

创建好fs之后挂载：mount /dev/openeuler/root / 挂载后即可使用，为了开机自动挂载修改/etc/fstab文件

mount /dev/lvpath1 /home

根目录空间不够了：如果此时vg还有剩余的PE，可以：

lvextend -l+1122(空闲PE，可通过vgdiaplay查询) /dev/openeuler/root 扩了硬件

ext2online / 同时扩容fs，扩fs 扩fs的命令有可能是下面的

resize2fs /dev/mapper/open-root 扩展fs

/dev/mapper/openeuler-root ==== /dev/openeuler/root是软连接

还可以这样扩硬件：lvextend L+55G /dev/openeuler/root ext2online /

若卷组空间不够了：

vgextened lvmdisk /dev/sdd1 往卷组中增加物理卷-->之后再扩容lv

```

增加根分区：

lvresize -l + 100%FREE -r /dev/mapper/xxx

linux的硬盘分为：SCSI 类型，现在用的比较广泛

IDE 类型，现已淘汰

UUID：分区初始化后就会有对应的UUID号，是针对分区的不是硬盘的。（分区出事化mkfs后就会有）

sr0-------光驱

\*\*\*\*增加硬盘的方法\*\*\*

1.虚拟机加硬盘SCSI类型的。。xxx.vmdk文件，就是虚拟硬盘--加了重启

lsblk 即可看到多了/dev/sdb 硬盘

2.对硬盘分区fdisk /dev/sdb

m+n(add 分区)+p（主分区）+默认enter+默认enter+w（写入） ok了

lsblk 可以看到/dev/sdb1了

分配的时候默认会占用所有磁盘中剩余空间，如果想多分几个区，可以不要按照默认enter，自己+xxG去分配。

3.格式化（mkfs）

mkfs -t ext4 /dev/sdb1

4.挂载--设置自动挂载（fstab文件）

mount /dev/sdb1 /home/jl/newdisk 挂载成功

最后记得mount -a 启动自动挂载

### mmu

mmu负责进程的虚拟内存与物理内存的映射，每个进程一旦运行，它就开始工作了

mmu的三级映射：

a.out-->页目录-->页表-->物理内存地址 页表就是描述虚拟内存与物理内存的映射

虚拟地址与物理地址的映射(mmu做的然后存到这里)

虚拟地址4g通过页表（映射表）映射到物理内存，页表由os维护，并被内存单元mmu引用，每个进程都有自己的页表

所以，每个进程感觉自己独享4g地址空间，并没有这么多的物理地址，仅代表它可以支配这4g。

slab是内存分配器，malloc的内存就是它分配的

页表是建立虚拟地址与物理地址映射的表，这个映射关系是mmap建立的，新的映射关系一建立就会被放进页表中

### arping

公网服务器如果想跟内服1通信：

1.发arp广播：谁的ip是192.168.1.2呀，请告诉我你的mac地址

TTL：8位最大256 意思就是这个数据包最多跳多少跳就得到达目的地址，每跳TTL值减1，如果减到0还没跳到目的ip，则丢弃此包。

\*\*真正的网络通信最后一定是基于物理层的mac去通信的。

ARP学习：

若A想与同一个局域网的B通信：（底层其实是MAC地址的信息交换）每台机器都有自己的网卡自己的MAC地址

1.若A知道B的ip地址，但是不知道B的MAC地址，可以通过发ARP广播，局域网内的别的机器都能收到此ARP信息：

ARP信息：A的ip，A的MAC，B的IP，空的MAC

只有对应的另端的机器会单独给A返回自己的IP与MAC信息，这下A就寄到自己的路由表中，以后就不用这样询问了

直接从自己的路由表中一查ip与mac的对应关系，直接发消息就可以了

以太网地址-->就是MAC地址（死记住）

静态arp：自己配置好，以后只能配好的mac之间通信，

动态arp：自己生成的，会改变，每次机器重启即使ip改变，ip与mac的对应关系也会相应改变

免费arp：检测同局域网内是否有别的IP与自己一样，如果有，就可以收到返回的信息，如果没有就不会收到。

----

route -n

destination为默认或0，0，0，0 表示默认网关，所有数据都会发送到此

gateway为0，0，0，0表示与本地同一网段，通信时不需要网关 这个就是路由器的ip。gateway就是路由器ip，网关的ip

root@tengxuncloud:~/jl# route -n

Kernel IP routing table

Destination Gateway Genmask Flags Metric Ref Use Iface

0.0.0.0 10.0.12.1 0.0.0.0 UG 100 0 0 eth0

10.0.12.0 0.0.0.0 255.255.252.0 U 0 0 0 eth0

若在 10.0.12.0 子网中传输数据 则直接通过eth0去传输就好，若想往0.0.0.0其他网络中传输数据，则需要通过网关10.0.12.1

traceroute ipaddr检测到达此ip需要经过多少跳-

route [add|del] [-net|-host] destIP [netmask] [gateway] [dev]

-net:目的IP是个网络

-Host：目的IP是个主机

netmask：若选择目的IP是网络才需要设置

网关与dev二选一即可（意思就是通过哪个去介入网络）

exp:

route add -host 1.1.1.1 [dev eth0|gw 2.2.2.2]

route add -net 1.1.1.1/16 [eth0|gw 2.2.2.2]

route add -net 1.1.1.1 netmask 255.255.0.0 [eth0 | gw 2.2.2.2]

route add default via x.x.x.x eth0

## Linux命令：

addr2line 0x11111 -f -e a.out 找地址对应的代码行

addr2line -e 文件名 -f ：显示对应的fun对应的行数

gdb vmlinux-debug： b 0x000000 ，在这里打断点就知道对应的地址的代码行了

ar crs libxx.a x.o y.o 打包静态库

ar -x libxx.a 解压静态库

objcopy -j .text a.out onlytexta.out 只抽出来.text段放到后面的文件中

useradd -m jl;增加用户,在ubuntu中如果没有-m，会出现用户虽然新增了，但是/home中没有新目录

userdel jl；删除用户 之后还要rm /home/jl

userdel -r jl:彻底删除用户,/home/jl 也会被删除

chown root /home/test 把这个目录或文件的属主设为root，别人就不能访问(还看对other的访问权限0664.)

chown jl:jl 1.c 目录或文件属主root group：root

$ sudo umount /mnt/ umount卸载时显示：busy--

fuser -vm /mnt(挂载点) 看看那个进程占用了此目录，kill掉就可以umount了

fuser -cu /mnt kill查到的进程

记住也不能在mnt目录里面进行umount，不然就说busy

df -i:查看inode使用情况

df -haT:显示文件系统与类型

df -h 查看磁盘空间

df -hT 查看文件系统类型

df -i 查看磁盘的inode使用情况，inode用完的话就不可以再创建文件了

du 统计目录大小

du -h --max-depth=n 查看几层目录结构的文件占用

du -h -d x 同上行

du -sh . h以k为单位 -sm 以M为单位

mknod /dev/dev中的驱动名 + c +刚才看的dev号（主设备号）+ 驱动文件中的从设备号 //创建设备文件

tar -czf c--create打包并不会压缩 z--gzip压缩 j-bzip2压缩 f-指定要处理的文件名 - x解压

-v显示解压缩过程 f放最后面+文件名

gzip -d xxx.gz

split -b size(字节) file splitfile 分割大文件

hexdump 将二进制转化为ascii区看，类似vim 中:%!xxd去看

mktemp -d test.XXXX 创建目录，创建于当前目录

mktemp test.XXXX 创建于当前目录，创建文件

mktemp -d 创建临时目录于/tmp目录中

mktemp 创建临时文件于/tmp目录中

sar：

sar -u(cpu) -o(out) test\_sar 5 3：每5s采样一次，连续采样3次，将结果以二进制形式保存到test\_sar文件中

sar -u -f test\_sar 查看以上采集的sar文件，直接cat看不了

%idle 空闲时间占用百分比。如果此值高但是os很卡说明cpu在等内存分配，应增加mem

太低说明CPU处理不过来，应增加CPU

sar -q 查看平均负载

```c

chmod函数。chmod('filename',0777);可以在程序中更改传入的文件权限。

chown改变文件所属用户，所属组。chown('filwname',uid,gid);所有者id，组id----/etc/passwd获得id

改变之后自己就属于其他用户的人，对文件不再有修改权限了。（除非该文件对其他人的权限也是rwx）

int link('oldpath','newpath')---创建硬连接

syslink--创建软连接的函数

stat穿透-跟踪软连接 --lstat不穿透----不跟踪软连接

truncate filename -s 3----把文件大小截断成3，文件中别的内容就不要了、

dup：复制文件描述符，本来一个打开的文件只对应1个文件描述符。现在有两个文件描述符对应同一个文件，★但是文件指针只有一个。

int dup(int oldfd); 直接返回新的文件描述符（文件描述表中没被占用的最小的fd）

int dup2(int newfd, int oldfd); 重定向打开的oldfd的数据到打开的newfd。 inode号都是test1的号。

strace ./a.out 查看a.out中有哪些系统调用（此时的a.out不用+-g参数就可以）显示的整个程序的syscall

strace -o 1.txt ./a.out 将a.out的strace调用写入文件，直接用>重定向是不可以的

backtrace 看当前位置使从main几层过来的，调用栈

strace -e open,close ./a.out 只查看a.out中的open，close这两种syscall

strace ./a.out 查看每个syscall花费的时间

ping谁谁就会给你回消息，a->b,b->a ICMP报文

MTU值大小：网络包的大小。若配置发的包不允许拆包，则MTU为1500的机器不能接收数据包大于1500的包

ping -s 大小（byte） -M do 目的ip 限制一次ping的时候发包的大小

ping -s 大小 目的ip：这样ping的时候会自动把包拆开发

dlopen:man一下

nslookup ip：找对应的网域名 前提得有DNS服务器

nslookup www.baidu.com 根据域名找对应的ip

dmsetup ls:查看当前device mapper信息

dmsetup info：详细信息

dmsetup deps：依赖关系

dmsetup table：表

ss -at:显示tcp连接，类似netstat

-s 显示socket

-p 显示应用程序与端口的对应关系

access 判断文件是否存在的系统调用

ltrace （lib trace）查看库的调用，跟踪库的调用(查看每个函数的来源在那个动态库中，每个函数的位置)

gdb -p pid 调式已在运行的进程

gdb 中backtrace：查看当前的调用栈。

gdb怎么知道程序在哪里死了呢？（段错误）-->其实就是看的进程死的时候产生的core文件

backtrace:出现好多栈帧

frame 栈帧编号：查看此帧的相关信息

0xnnnnn：in function(m=0xmmmmm(入参的地址),<入参参数>) at 某个文件某个函数的第几行；

head -n file 查看前n行

cal -1 输出本月日历 -3上月本月下月日历

tac filename:将文件内容倒序输出

who 显示当前哪些windows的ip在登录本机器。

w -i显示当前每个console在干啥，who的升级版

tty：终端设备(ctl+alt+F1~F7)前6个是文本吧，第7个是界面。命令行直接输入tty，可以看到当前是ptsx几号终端

pty：虚拟终端(它只是主机的/dev/ptsn在winows的映射)--ssh、serial都会增加这个里面的数量

lsblk 查看当前挂载的磁盘（硬盘）分区

ttyAMA0 arm的串口

ttyS0 x86的串口

brctl show:查看我的网桥连接的哪些网卡 ---桥接

网桥就相当于电脑中的交换机：此电脑中的所有的虚拟机ip与外界通信都通过此网桥的ip

blkid 看磁盘uuid

strings xxx.so 在库中提取字符串并打印出来，对于if，else分支的打印字符串，它只能拿到程序实际运行到的分支中的printf的字符串

dd iflag=xxx oflag=yyy if=uImage of=/dev/sdb seek=1000扇区号，将内核存储到1000扇区

gzip -d xxx.gzip 解压

list --dependencies xx.target

id 查看useid、groupid

hostname -f 查看主机名

hostname 查看主机名

cat /etc/hostname 查看主机名

hostnamectl set-hostname jl：更改主机名为jl，reboot生效

sync：强制把当前内存中的数据写到磁盘。

普通cp一个大文件到某个目录时，命令执行完可能并没有cpy完，然后os会慢慢将当前内存的数据往磁盘中写，若此时断电，大部分数据丢失。

此时应该执行sync，强制刷新mem中的数据到磁盘。

----------

w:查看负载均衡

uptime:查看开机到现在持续了多久，单位是hour+min，也可以看负载均衡

netstat -tnlp 查看网络状态

-t 只列出tcp协议的链接（protocol）

-n 将ip地址从字母组合转化为ip点分十进制，将协议转化为端口号（不加-n可以看一下再+-n去对比）

-l 过滤state列状态为LISTEN列的链接。监听

-p 显示发起链接的进程的pid与进程的名称

只有S端才有监听态：LISTEN

1个电脑可以有多个页面可以连接百度，但是百度给每个页面的返回界面都不一样，因为port不一样。他们发给百度的port都是80

top 后 P：cpu从高到底显示

​ M：内存从高到低显示

​ 1：多个cpu信息都显示出来

top p pid1 p pid2--->动态监控pid1与pid2的状态

ls > 1.txt ==== ls 1>1.txt 标准输出流到1.txt

ls 1>file 2>&1 把标准错误流给标准输出流，把标准输出流 存到 file中

ls 2>file 1>&2 把2给1，把1给file 注意>左侧的1要跟>挨着没有空格，有空格相当于ls 1文件名了

pstree -p 显示pid以父子关系 -u显示所属user

sshd 监听22号端口 ssh -p root@ip 远程登录ip的某个端口(记住关闭防火墙，设置/etc/ssh/sshd\_config PermitRootLogin)

ssh root@ip 远程登录ip

ssh -vvv ip 显示登录的异常

ssh远程链接会断开，修改/etc/ssh/sshd\_config文件

ClientAliveInterval 60----检测一次花60s

ClientAliveCountMax 3600---这个值改大一些,系统检测当前有人操作就不断开了

如果还会断开

在/etc/profile文件中最后export TMOUT 9999

如果链接不断断线，可能因为ip或mac重复了，可以直接在xml中更改mac的几位。

httpd 监听80端口

### scp

scp重复拷贝同一文件名的相同目录会覆盖之前的文件。

### zipper

zypper ar -c file:///iso

zypper refresh：更新repo源

zypper ls:看repo有几个

zypper packages:看repo中提供哪些包

zypper search xx:查看xx软件包是否已安装

### tmux

对于窗口：不太常用

c:新建

&：关闭

l：切换 n:下一个 p:上一个

w:窗口列表 上下键移动+enter切换 \*所在位置就是当前选中的窗口

对于pane：

%：水平增加pane

“：垂直增加pane

x:关闭

；：切换pane o:顺时针切换 ^o:逆时针切换

对于整个会话：

^ b d:挂起 tmux attach -t sename 进入会话

tmux ls:查看挂起的会话有几个

### iptables

iptables == firewalls 防火墙 看下哪些端口是启用的

若只需要转发则不会经过INPUT链

发送至本机的报文：PREROUTING->INPUT链

由本机转发：PREROUTING->FORWARDING->POSTROUTING

由本机发出：OUTPUT->POSTROUTING

表：--》所有规则分4类，存在于这4个表中

filter:过滤，iptables，防火墙

nat:网络地址转换，内核空间是iptables\_nat

mangle:拆解报文，做出修改，并重新封包，对应内核空间iptables\_mangle

raw:关闭nat表启用的连接追踪机制，对应iptables\_raw

每个链存在那些种类的规则：

PREROUTING：raw，mangle，nat

INPUT:mangle，nat

FORWARDING:mangle，nat

OUTPUT:raw，mangle，nat，filter

POSTROUTING:mangle，nat

表的优先级：raw，mangle，nat，filter 前面的优先级最高

添加或删除规则是往 链与表结合 中添加的；\*\* iptables -t filter -D INPUT 3

查iptables规则也是根据 链与表 结合起来去查 \*\* iptables -t filter -vnL INPUT

iptables -t filter -I INPUT -s ip -j DROP 拒绝ip的报文到本机

iptables -t filter -I INPUT -s ip/16 -j REJECT 针对ip所在的子网的包都拒绝

iptables -t filter -I INPUT -s ip1 -d ip2 -j DROP:只拒绝ip1到ip2的包(想让我给你转发，没门)

iptables -t filter -I INPUT -d ip3 -j DROP :只drop目的ip为ip3的报文（并不是说目的ip不是ip3的就不drop）

-p tcp/udp/icmp:只拒绝协议为tcp或udp或icmp的报文，其他类型的不drop

--dport 22：只匹配来源的报文的目的端口是本机的22号端口的报文（注意是本机的22端口）

修改iptables后：service iptables save(临时修改)

/etc/sysconfig/iptables(永久修改)

报文的处理方式有：accept.drop,reject

iptables -F:清空所有的防火墙规则

## shell

ctrl+r:输入最近输过的linux指令，os会默认匹配之前使用过的cmd，直接回车即可执行

在test.sh中用source 1.sh调用1.sh脚本，则1.sh脚本中的$0就是test.sh

source不新建bash，继承之前的环境变量，修改这种变量也会修改父进程的变量

在test.sh中用sh 1.sh调用1.sh脚本，则1.sh中的$0就是1.sh\*\*\*\*

sh会新建bash，不继承之前的变量。

行过滤工具

查找某个时间内的所有记录:grep -n "Jan 29" \*.log /var/log 目中中执行

grep [选项] ‘关键字’ 文件名——》查找文件名中包含关键字的行，并打印出来

选项可加可不加就是-v -f -m 。。。

想要高亮显示：grep --color=auto 'root' passwd在passwd文件中找root关键字

怎么能让grep == grep --color=auto呢？以后就不用巧后面的了

alias grep='grep --color=auto'临时生效

要想永久生效需要改配置文件才行一般都在/etc目录下

source /etc/bashrc 让这个文件重新生效 或者 /root/.bashrc

grep -e 'x' -e 'y' 查带x或带y的行。

-n 显示行号

-i 忽略大小写匹配

^关键字：以关键字开头 '^var' ||

关键字$ :以关键字结尾 'var$'

grep -nB 3 '^www' passwd

grep -nB3 '^www' passwd 匹配开头是www的行再加上3行

grep -nA3 '^www' passwd 匹配开头是www的行再加下3行

grep -nA3 -B3 '^error' passwd

grep -nC3 '^error' passwd 过滤查看错误日志方法，过滤带error的前3行后3行

-v 取反打印

-v -x -n ====-vxn

-w 精确匹配关键词 若行中含有chinese，用-w匹配chin则匹配失败，因为是精确匹配

-o 只打印匹配到的字符 grep -o '[0-9]' 打印出来的只有匹配到的字符，不会整体打印行 可以计算文档中共有多少个关键字

列过滤工具：

grep -w '[0-9][0-9]' 匹配只有两位数字的行

cut：

cut -d':' -f1 passwd:查找以':'分割的第一列字符串

cut -d ':' -f1 passwd |cut -c 1-3

-d:置顶分割符号--默认是\t分割 直接-d: 分割也是可以的

-c:只截取每行中的对应位置的字符 -c m-n

-f:显示第几列 -f1,6显示第一列与第六列

sort工具：字符按照acsii排序升序输出

-u:去除重复行并且排序

-r：降序 ，默认升序

-o:重定向文件

-n:以数字排序 默认以字符排序

-t:分隔符 类似cut的-d参数

-k：第N列 类似cut的-f

-b:忽略前导空格

-R：随机排序

sort -n -t: -k3 1.txt 利用文件中的以冒号分割的第3列数据对文件排序后整体输出

tac filename:将文件内容倒序输出 --相当于reverse

uniq：只去除连续重复的行

-i:大小写忽视 重复行就会变多

-c：统计重复行

-d：只显示连续的重复行

uniq -dc 2.txt:统计每种重复行的个数,不重复的不显示

diff：

-N:将不存在的文件当成空文件

-u:合并格式显示，gitee就是这么用的

touch dir1/file{1..5}

touch dir2/file{1..3}

touch dir2/test{4..5}

diff -q dir1 dir2 ----查看文件夹文件的不同，不对比文件中的内容

- diff -Nur在那个目录执行 打patch时就在那个补录打可以时-p0 在目录里面就开始-p1 -p2

patch -p0 < 1.patch --fuzz=0 不需要指定要将patch往那个目录里面打

patch -RE < 1.patch 退回来打进去的patch

单个文件打补丁：

diff -u file1 file2 > file1.patch

patch file1 file1.patch 或者下面

patch file1 < file1.patch

----tr小工具 行内字符替换

tr ' ' '\n' 空格换成换行符

tr ' ' '\t' 空格换成制表符

command |tr 'A-Z' 'a-z'

tr 'A-Z' 'a-z' < filename 字母大小写转化

tr -d 'a-zA-Z :/' < filename :删除文件中的字母、空格、冒号、/

tr -s 'a-z' < file 压缩相同的挨着的小写字母，压成一个

tr -s ' ' < file 压缩相同的挨着的空格

---paste小工具 合并文件行工具

paste -d： file1 file2 并排合并打印，以:分割

paste file1 file2 并排合并打印，以tab分割

patse -s file1 file2 file1内容合并成1行与file2合并成1行做对比

---sed工具：

sed：行处理工具

sed 参数 '脚本语句' filename

sed 参数 -f '脚本文件' filename

a, append 追加'ammm'，每行之后加新行mmm '3annn'第3行后加一行新的nnn

'$akkk'文件最后加一行kkk a后面有无空格都可以

i,insert 在''之外-i 修改源文件 在'i'之内inset行

d,delete 删除

s,substitution 替换

sed -i '2,7d' 删除2-7 行

sed 's/xxx/yyy/g' 把行中带为xxx改成yyy，g表示都换，若没g，只换此行的第一个

sed '10s/xxx/yyy/g' 只想替换第10行的数据，别的行不替换

sed '6,10s/^/#/' 1.txt 给6-10行加注释

删注释:sed -i 's/^#//' 1.c 只会删除开头的#，后面的#不删除

### sed 默认时对每一行都要处理的

sed -i '/xxx/d' file 删除含有xxx的行 sed -i "/$var/d" file--注意要引用变量的时候不能用单引号，必须双引号

sed '/^root/i hello' 在以root开头的行上面插入hello行

sed -i '88 r b.file' filename:在filename的第88行后+b.file的内容

sed '3ixxx' 3行前面新加xxx行

sed -n '/xxx/p' file 只打印匹配到的行，别的地方的行不打印

sed -n -f xx.sedscript file 根据sed脚本对file的内容进行修改输出

sed -[opt] '动作（p,i,a,d,c替换）'

sed '$ c hello' 最后一行换成hello

sed '$ a hello' 最后一行后增加新行 hello

sed '/[0-9]/d' 删除包含数字的行

sed -n 's/\(192.168.1.\)1/\1254/gp' 1.txt 将文件中192.168.1.1替换成1.254

（192.168.1.）1 ==== \1 第一对括号里面包起来的就是 \1

sed -n 's/\(192.\)\(168.\)1.1/\1\2254/gp' 1.txt

（192.）==\1 （168.）==\2

---awk工具：是个语言，很强大 一行一行的处理数据

在协议中数据截取电压后计算：

head -10 youxiao.txt |awk -F' ' '{print $3,$4,$5,$6,$7,$8}' 截取所有电压数据

将对应位置电压数据 高低字节 转化成10进制再合并

head -10 youxiao.txt |awk -F' ' '{print strtonum("0x"$4)\*256+strtonum("0x"$3)}'

截取电压位置的数据

strtonum("0x"$n) 将对应位置的16进制转成10进制 转化成10进制 后可以直接运算

|awk -F' ' '{print strtonum("0x"$4)\*256+strtonum("0x"$3),strtonum("0x"$6)\*256+strtonum("0x"$5),strtonum("0x"$8)\*256+strtonum("0x"$7)}' > vol.txt

echo "4783" |awk '{printf("%x\n",$0)}' 10进制转16进制

echo "47 83" |awk '{printf("%d\n",$1\*$2)}' 10进制显示

echo "47 83" |awk '{printf("%x\n",$1\*$2)}' 计算结果转成16进制显示

i=0;for name in `ls`;do mv $name $i"-"$(($i+9999));let i=i+10000;done 批量重命名命令

awk [选项] '命令部分' filename 注意：引用shell变量需要用双引号引起来

-F：以什么符号分割，默认空格

-v:定义变量并赋值

正则表达式进行地址定位；awk 'NR==1,NR==5{print $0}' 打印1~5行

awk 'NR==1,NR==5;{print $0}' 打印1~5行2次，其他的都打印一次

awk 'NR==1||NR==5{print $0}' 打印第1行或者第5行

awk 'NR>=3&&NR<=5{print $0}' 打印第3行到第5行

1~5行打印1次，第1行满足分号后的条件，打印1次

root@tengxuncloud:~/jl# awk 'NR==1,NR==5;/^root/{print $0}' passwd |cat -n

1 root:x:0:0:root:/root:/bin/bash

2 root:x:0:0:root:/root:/bin/bash

3 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

4 bin:x:2:2:bin:/bin:/usr/sbin/nologin

5 sys:x:3:3:sys:/dev:/usr/sbin/nologin

6 sync:x:4:65534:sync:/bin:/bin/syncr

花括号{}进行语句的执行；

BEGIN..END:

'BEGIN{awk语句}；{处理中}；END{awk语句}'

awk中的相关变量：

$0:当前处理行的所有记录

$1,$2...$n:文件中每行以间隔符号分割的不同字段

NF：当前处理行的字段数--列数量

$NF:最后一列 --awk -F: '{print $NF}' 打印文件最后一列

FNR/NR：行号

FS：定义输入间隔符

OFS：定义输出字段分隔符，默认空格

awk -F: '{print $NF,$(NF-1)}' 打印倒数第一第二列

awk -F: '/root/{print $0}' 打印包含root字符的行

awk -F: 'NR==1,NR==5;/^root/{print $0}' 打印以root字符开头的行

awk 'NR==1,NR==5;{print $0}' 1~5行打印2次，后面的每行打印一次--分号前面就会处理并打印

默认输出结果是以空格分割的，可以在BEGIN中指定输出分隔符，输入分割符：

awk -F: 'BEGIN{OFS="@"};{print $1,$NF}' passwd

awk 'BEGIN{FS=":";OFS="@@@@@@"};{print $1,$NF}'

awk 'BEGIN{FS=":"};{print $1"@@@@@@@@@@@"$NF}' BEGIN中没有定义输出分割符，直接在处理数据中"@@@"也可以分割

awk 'BEGIN{FS=":"};{print "用户名是"$1"bash是"$NF}' 打印的时候前后都可以用""包起来想打印的

print和printf两者的区别：

用print打印，会在打印结束后加上换行符

用printf打印，后面不会添加任何符号

cat 2.txt |awk -F' ' '{for(i=58;i<110;i=i+2){printf $i $(i+1)" "}print""}' 后面的print是为了换行

--date |awk '{print "month:" $2 "\nYear:" $NF}' 将结果分行打印

printf：格式化输出

%s 字符类型

%d 数值类型 16进制转10进制 printf '%x' 0xnnn

- 左对齐，默认是右对齐

%x:10进制转16进制 打印：printf '%x' 10进制num

awk -F: '{printf "%-15s %-10s %-15s\n",$1,$2,$3}' /etc/passwd

awk -v NUM=3 -F: '{print NUM}' passwd -v变量赋值 变量NUM打印的时候不需要带$，带上就错了

awk -F: 'BEGIN{print "logshell\t\tloghome\n\*\*\*\*\*\*\*\*"};{print $NF"\t\t"$(NF-1)};END{print "\*\*\*\*\*\*\*\*"}' passwd 处理文件数据前打印两行+处理数据+处理完数据打印一行

awk与正则结合：

==：等于 !=不等于

~：匹配 !~:不匹配

&& || ! 逻辑与或非

awk -F: '/^root/,/^lp/{print $0}' 筛选条件要以//隔开，中间逗号隔开，从以root开头的行打印到以lp开头的行

awk -F: 'NR==1,/^lp/{print $0}' 从第一行打印到以lp开头的行

awk -F: '/^root/||/^lp/{print $0}' 打印以root开头或者以lp开头的行

awk -F: '/^root/;/^lp/{print $0}' 分号，命令分割，匹配到以root开头打印。匹配到以lp开头打印

awk -F: 'NR>=30 && NR<=39 && $0~/bash$/{print $0}' 匹配30~39行中以bash

ifconfig lo |grep inet6|awk -F'[: ]+' '{print $3}' 前导字符多个或1个空格或冒号 分割

awk编程：高级筛选

awk 选项 '正则，地址定位{awk语句}' 文件名

awk语句：{if(表达式){语句1;语句2;...}}

awk -F: '{if($3>=500 && $3<=60000){print $1,$3}}' 没有正则与地址定位，直接写的awk语句

awk 'BEGIN{if(`id -u`==0){print "is admin"}}' 执行失败，awk语句中不能用``获取命令结果

awk 'BEGIN{if($(id -u)==0){print "is admin"}}' 执行成功，awk语句中能用$()获取命令结果

if--else if--else

awk -F: '{if($3==0){print $1"is 管理员"}else{print $1" is not admin"}}' passwd

awk 'BEGIN{if($(id -u)>=500 && $(id -u)!=65534){print "is normol user"}else{print "is root"}}' 注意不能用&&链接，需要-a才是逻辑与

awk -F: '{if($3==0){print " is root"}else if($3>=1&&$3<=499){print " is system admin"}else{print "is normol user"}}'

awk -F: '{if($3==0){i++}else if($3>=1&&$3<=499){k++}else{m++}};END{print"root="i,"xx="k,"yyy="m}' 统计当前系统系统用户，普通用户，root用户个数

for循环：

awk 'BEGIN{for(i=1;i<=5;i++) {print i}}' 不像shell语句，不需要双括号

awk 'BEGIN{sum=0;for(i=1;i<=10;i++) {sum=sum+i};print sum}' 最后的print不需要加END{}，是处理完文件数据才加" ;END{}"

----shell 中的通配符：

\*：匹配0-n个任意字符

？：匹配任意单个字符

[list]：匹配[list]中任意单个字符，或一组单个字符

[abc] [a-z] [123] [0-9]

file[1-13]:表示1-1 3共2个字符中的一个字符

file[2-13]:表示3共1个字符中的一个字符

[0-9,10,22]:表示0-9+','中的任意1个字符

[!list]：匹配除了[list]中任意单个字符

{str1,str2,...}:匹配多个字符串

file{1,2,10,11}.jpg

bash中的引号：

root@tengxuncloud:~/jl/dir1# echo "$(date +%F)" 命令得用双引号包含起来

2022-09-02

root@tengxuncloud:~/jl/dir1# echo '$(date +%F)' 用单引号括起来的命令执行结果不会当成命令执行结果

$(date +%F)

双引号中允许通过$引用其他变量值

单引号中不允许通过$引用其他变量值，bash中的特殊字符都会被当成普通字符

``：反撇号，相当于$() echo "`ls -la`"可以正常运行

echo "`echo "`ls`"`" ``不允许嵌套，因为``本身不分顺序，会把" `echo "` ls `"` "命令解析错误

echo "$(echo "$(ls)")" $()允许嵌套

echo "$(echo "`ls -la`")" $()中套``允许

----mkisofs -o xx.iso xx.tar.gz(压成tar包再做iso)

shell脚本如果./test.sh不给你说哪一行有问题。

所以建议用解释器运行：bash test.sh---会给你说哪一行有问题

取变量值-> $a 直接加变量名也可以，${a}也可以

赋值=两边不能有空格。否则不能成功赋值

unset var---取消此变量

`uname -r` 就是得到命令执行的结果

a=$(uname -r) 也可以得到命令结果，并保存到a中

$a 与 ${a}的区别： A=123456

echo ${A:2:3} 切片：从2位置开始切3个

交互：

read -p ”提示“ 变量名 选项都是可有可无的

read -n 定义字符数量

read name：将输入给name变量

read -n 5 -p "input your name" name：输入字符串长度不能超过5个。

从文件读入赋值：&&\*\*

read -p 'inptu ip' IP < ip.txt 把ip.txt中的内容赋值给IP变量。

read -s var : 不显示用户输入内容，密码就是这么做的

read -t 5 A ：5S内不输入则自动返回，给A赋值为空

read var1 var2 < 1.txt 从文件中获取数据给var1 var2赋值

### shell函数传参

定义时：fun（）{$1xxx $2xxx}

fun()

{

echo "$\* xx yy " $\*就表示每个入参都打印

}

返回值只能返回函数执行状态0成功非0失败。要其他的话用定义全局变量接受

调用时fun xx yy,返回数字用return，返回值接收用$? 或者 res=`fun $1 $2` 或者 res=$(fun $1 $2)

返回多个值的字符串用echo 'xxx'; echo 'yyy'

res=$(fun)，返回值是多个字符串中间以空格分开

read -p 'xxxxx' varname

shell脚本第一行

set -x :就说后面的程序执行的时候类似外界sh -x 执行一样

set +x:关闭这种提示

set -e 之后的命令出错立即退出

set +e 之后的命令出错不退出

变量分类：

变量：

declare -i var :当前中断变量var声明成int类型，不能赋值字符串了

-r：声明称只读变量 unset不了，只能重启终端了

export var:将定义好的普通变量var 临时导成环境变量

本地变量->只对当前进程有效 定义在当前终端（是个进程），只能在当前终端使用

环境变量—>当前终端能用，子进程也能使用

env：查看当前用户的环境变量（只是当前用户的）

set：查看当前用户的本地变量与环境变量

全局变量：/etc/bashrc 所有用户都生效 建议（函数与别名都放这里面）

/etc/profile 系统与所有用户都生效

~/.bashrc 当前用户生效

重要：系统变量：$? -> 上条命令执行结果是0 说明上条命令运行成功了 ping 11.1.1.1也可以

ls --> echo $? 结果==0

lll -> echo $? 结果127说明上条命令出错了非0就是出错

$1 --$9 脚本后面的位置参数 ${10}得包住，不然会当成$1

$@ $\*-->脚本后面的所有参数

$# -->脚本后面传参个数，不包含./xx.sh

echo "\$0=$0" 前面的$0是字符串，后面的是变量

$$:当前进程的进程号

$!:当前终端后台运行的最后一个进程号

数组：

declare -a 定义普通数组；查看普通数组

declare -A 定义关联数组；查看关联数组

普通数组：只能使用整数作为数组索引

关联数组：可以使用字符串作为数组索引

普通数组：

赋值：

array=(var1 var2 var3 ..) 引用：${array[1]} 不加{}不行

array[x]=var3 单个赋值

array=(`cmd`) 将cmd结果赋值给array 空格 换行符都会将元素分开

array4=(1 2 3 4 "hello world" [10]=linux) 双引号可以将长字符串接到一起，直接[]可以给固定下标赋值

读取：

array[\*]、array[@] 获取数组里所有元素

#array[\*] 获取数组里元素个数

!array[@] 获取数组所有的元素索引下标--有的数组中间部分没赋值

${names[@]:0:10} 获取所有元素后，切片显示0下标开始共10个

对于数组：array=([0]="var1" [1]="var2" [2]="var3" [10]="sad")

root@tengxuncloud:~/jl# echo ${!array[@]} 所有下标不是连续的

0 1 2 10

root@tengxuncloud:~/jl# echo ${array[\*]} 查看所有有效元素时又是连续显示

var1 var2 var3 sad

root@tengxuncloud:~/jl# echo ${array[\*]:0:4} 切片显示4个却又将所有元素都显示出来了

var1 var2 var3 sad

关联数组：

赋值：

先得declare -A asso\_array 声明是关联数据，不然会被当成普通数组

asso\_array[linux]=one

asso\_array=([name1]=harry [namen]="hello world") 多个同时赋值

读取方法与普通数组一样。下标是无序的

let asso\_array[java]++ 相当于array[i]++，必须有let

变量内容的删除替换：\*:所有 .:分隔符号 .\*:去的是.与后面的\*内容 \*.:去的是左边的内容与.

%:从右往左去掉一个/key/

%%:从右往左最大去掉/key/

#:从左往右去掉一个/key/

##:从左往右最大去掉/key/

root@tengxuncloud:~/jl# url=www.taobao.com.test

root@tengxuncloud:~/jl# echo ${#url}

19

root@tengxuncloud:~/jl# echo ${url#\*.}

taobao.com.test

root@tengxuncloud:~/jl# echo ${url##\*.}

test

root@tengxuncloud:~/jl# echo ${url%.\*}

www.taobao.com

root@tengxuncloud:~/jl# echo ${url%%.\*}

www

四则运算：只支持整数运算

$(()) 符号 echo $((1+1))

$[] 符号 echo $[1+1]

expr 程序计算 expr 10 / 5 \*\*expr 用的时候‘运算符’左右要加空格 乘号\*用的时候要加转义字符\\*

let 程序 n=1;let n+=1

echo 1+1.5|bc bc可以帮忙处理数学运算

bc 工具可以计算数学 直接enter后算 quit+回车退出

条件判断语句：

test+表达式

[ 表达式 ] 不支持正则（2边要有空格）字符串必须用""抱起来

[[ 表达式 ]] 支持正则（2边都要有空格）字符串不需要用""包

判断2个字符串是否相等： test $s1 = $s2 等号两边都有空格，没空格是赋值

[ $s1 = $s2 ] 等号两边都有空格，[]中括号也是

if [ xxx ]：if后面也有‘ ’的，要注意

-z :字符串为空

-n ：字符串不为0

linux怎么判断两个文件一样：看inode号->ls -li file--->[ a -ef b ]

判断文件是否为空[ ! -s file ]

不会了man test关键字查 判断的内容

case语句：

case var in

pattern 1) 或者

hello|world|h)

cmd1

;;

pattern 2)

cmd2

;;

\*)----default语句

cmd\*

;;

easc 表示case结束

案例：给脚本传不同值做不同事情

#!/bin/bash

case $1 in 判断$1传的入参是什么

start|S) 入参是start或S，则进入此case

service apache start &>/dev/null &echo "start ok"

;;

stop|P)

service apache stop &>/dev/null &echo "stop ok"

;;

reload|R)

service apache reload &>/dev/null &echo "reload ok"

;;

\*)

echo "输入参数"

;;

esac

案例2：

----echo的另一种 写法

root@tengxuncloud:~/jl# cat---本身就是你输入什么 它查看什么

hello

hello

world

world

^C

root@tengxuncloud:~/jl# cat <<END 看到END才结束，结束的时候把输入的所有内容打印出来

> hello

> world

> END

hello

world

root@tengxuncloud:~/jl# cat exp.sh 脚本中向向屏幕echo大量内容 不需要一个一个echo

#!/bin/bash

cat <<end

\*\*\*\*\*\*chose\*\*\*\*\*

h help

m map

s select

\*\*\*\*\*\*\*\*

end

su jl <<END 跑到jl用户的家目录创建file文件，然后返回到当前用户

touch file

END

计算传入的参数累加和：

root@tengxuncloud:~/jl# cat sum.sh 1 2 3 0 4

#!/bin/bash

sum=0

while [ $# -ne 0 ] 脚本传入的参数个数不为0才 左移计算

do

let sum=$sum+$1 始终是$1 第一个参数

shift---->传入参数左移 shift 2参数一次左移两位

done

echo $sum

#!/bin/bash

sum=0

for i---->自动获取个数 便利参数

do

let sum=$sum+$i

done

echo $sum

循环：

for((i=1;i<=5;i++));do echo $i;done 双括号写法

for i in {1..5} 打印5次你输入的第一个参数

do

echo $1--------$i打印1到5，i每次都在变化

done

---------

sum=0 sum是变量，$sum是个数字，是保存在变量中的值

for i in {1..100..2}

do

sum=$[$sum+$i]

done

i=1;while(($i<=5));do echo $i;i=$[$i+1];done 涉及判断 四则运算

批量修改win中特殊的文件名文件为普通命名的文件：

i=1;for name in `ls`;do mv $name $i".txt";let i=i+1;done 不管文件名是什么，直接mv ---rename 带''的文件，会失败

if 条件：a else b fi == 条件 && a || b

至多有1个（<=1）------(>1)至少2个

至少1个（>=1）--------(<1)1个也没有

至少2个（>=2）--------(<2)最多1个

至多2个（<=2）--------(>2)最少3个

至少有1个不发生--------1个也没有不发生------所有的都发生。

充分：

a--

c-- 3个同时成立b才成立

d--

必要：a+c+d-----b abc都是b的必要条件得同时满足b才能成立

1、如果p是q的充分条件，那么q一定是p的必要条件。

2、如果p是q的必要条件，q一定是p的充分条件。

问题：

批量在多个服务器创建同一个用户： ip.txt里面放的每台主机ip 密码

for i in `wc -l ip.txt |cut -d' ' -f1` 根据ip.txt文件行数确定循环次数

do

done

---

while read ip passwd

do

/usr/bin/expect <<-END &>/dev/null #中间交互打印都给到null不显示

spawn ssh root@$ip

expect空格{

"yes/no"空格{send "yes\r";exp\_continue} #登录时看到yes no选项，输入yes回车

"passwd:"空格{send "$passwd\r"} #见到passwd后输入密码回车

}

expect "#" {send "useradd yy1;exit\r"} #登录成功执行命令

expect eof #退出

END #END中间用的expect语言，见到END表示退出这种语法，回到shell脚本中

done < ip.txt 将文件中的ip 密码分别赋值给循环变量ip passwd，有多少行就赋值多少次

root@tengxuncloud:~/jl# cat except.sh

#!/bin/bash

set -x

while read ip passwd

do

/usr/bin/expect <<-END &>/dev/null

spawn ssh root@$ip

expect {

"yes/no" {send "yes\r";exp\_continue}

"password:" {send "$passwd\r"}

}

expect "#" {send "useradd yy1;exit\r"}

expect eof

END

done < ip.txt

root@tengxuncloud:~/jl# cat ip.txt

127.0.0.1 jl02011318

root@tengxuncloud:~/jl#

案列1：

将当前机器普通账户的公钥拷贝给别的root主机，则以后此普通用户可以直接免密登录这个root主机

指令：ssh-copy-id root@x.x.x.x 拷贝需要密码，拷贝后 root主机的.ssh目录会多一个authorized\_keys

无交互生成公钥方法:ssh-keygen -P '' -f id\_rsa 交互的密码都是空，-f：密钥文件名

案例2：

写一个脚本，将跳板机上yunwei用户的公钥推给所有它可以ping通的机器上,主机与密码文件提供 ip:passwd

---fail code

#!/bin/bash

#判断yunwei账号是否存在

{

id yunwei

[ $? -ne 0 ] && useradd -m yunwei && echo 123|passwd --stdin yunwei #给yunwei设置密码

} &> /dev/null #多行命令都需要重定向结果到/dev/null时可以直接用{}括起来执行

#expect是否存在

rpm -q expect

[ $? -ne 0 ] && yum install expect && "expect 安装ok"

#判断yunwei的密钥对是否存在，不存在就创建

su - yunwei #切换到运维账户再去创建密钥对 ------程序的执行过程中不能跨用户 ，这种代码有问题

[ ! -f $/home/yunwei/.ssh/id\_rsa.pub ] && ssh-keygen -P '' -f id\_rsa

---success code--yunwei只用创建一次，直接创建好了之后用yunwei账户去写代码

#!/bin/bash

#判断yunwei的密钥对是否存在，不存在就创建

[ ! -f $/home/yunwei/.ssh/id\_rsa.pub ] && ssh-keygen -P '' -f .ssh/id\_rsa

#循环检查网络能否ping通 并 直接公钥推送 ---能用1个循环解决的事情就不要用2个循环

ip\_txt=/~/ip.txt

while read ip passwd 因为ip.txt中的内容是以:分割的，所以不用while做，还是用for

do

done<$ip\_txt while的输入用于读的文件中数据以空格分割的文件

---

#!/bin/bash

#生成公钥 -f 加的绝对路径

[ ! -f $/home/yunwei/.ssh/id\_rsa.pub ] && ssh-keygen -P '' -f /home/yunwei/.ssh/id\_rsa

for i in `cat ip.txt` i就是遍历的每行数据

do

ip=`echo $i|cut -d: -f1`

pass=`echo $i|cut -d: -f2`

ping -c1 $ip &>/dev/null

if[ $? -eq 0 ];then ping成功，加公钥到此ip

/usr/bin/expect <<-END &>/dev/null

spawn ssh-copy-id root@$ip

expect{

"(yes/no)" {send "yes\n";exp\_continue}

"password:" {send "$pass\n"}

}

expect eof

END

else ping失败的ip加到某个txt文件中

echo $ip>>ip\_down.txt

fi

done<ip.txt

测试是否拷贝成功

ssh root@$ip hostname 不想跳转到别的机器，只想ssh后执行cmd后再回来

echo $? ==0 说明密钥拷贝成功了，!=0说明拷贝失败

------------

$(( xxx )) 双层括号内可以进行数学计算

cat > $filename << EOF

XXX

XXX

EOF 可以将中间的内容放到文件当中

操作文件中的行

while read line

do

op $line

done < file.txt

函数：

程序入口在哪里？

1. 没有函数入口，直接在shell中先source一下脚本名，然后直接在命令行 调用函数就可以 只对当前终端有效

若要return return的数字必须是0-255之间，不能随便return

怎么返回字符串 直接echo $mystr即可，外面用变量接收就可以收到str

echo $? 就可以读到自己定义的函数的返回值

调用：

当前终端命令行调用 source x.sh

全局生效---将函数写到/etc/bashrc --所有用户生效 /~/.bashrc 当前用户生效

脚本中调用---在别的脚本中调用写好的函数

本文件中函数调用，直接在本文件中 函数以外 的地方直接调用就能执行，不需要main入口

随机数：

$RANDOM--0~65535 $(($RANDOM%n)) ---0~n-1 $(($RANDOM%n+1))---1~n

电话号码切片显示：139\*\*\*\*${var:7:4} ${var:7:4} 从7开始数4个字母

查看文件某行数据：head -n file |tail -1 选取文件前n行再取最后一行

获取cmd结果：str=`cmd`

### man指令每章对应的内容：

man 手册9章分别对应的内容

Executable programs or shell commands // 命令

System calls (functions provided by the kernel) // 系统调用，比如 man 2 open

Library calls (functions within program libraries) // 函数库调用

Special files (usually found in /dev) // 特殊文件, 比如 man 4 tty

File formats and conventions eg /etc/passwd // 文件格式和约定, 比如man 5 passwd

Games // 游戏

Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7) //杂项

System administration commands (usually only for root) // 系统管理命令

Kernel routines [Non standard] // 内核例程

### 进程组的概念：

只有父进程的进程组是自己，子进程的进程组是父进程。

kill -9 -pgid 一次杀死所有进程组的进程,'-'pgid （区别父进程与进程组的概念）

getpgrp() :获取当前进程的进程组id

getpgid(pid) ：获取指定进程的进程组id 成功0，失败-1置errno

setpgid(pid,pgid);改变进程默认所属的进程组， 成功0，失败-1置errno

普通用户不能改变root用户的进程的进程组

## 正则表达式：

### Egrep的正则：linux命令

基础正则：

扩展正则：egrep

egrep “[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}” filename 在filename查找ip地址

注意“ 位数 值范围 仅分割意思 .在正则中有别的意思

\：在这里是转义字符 因为'.'是匹配任意1个字符的

[]匹配括号中任一字符 [abc]d =ad/bd/cd

^：位于中括号内的时候-[^lxy] 匹配除lxy以外的任意1个字符

位于中括号外的时候- ^con 匹配开头是con的行

数字限定符：

？：[0-9]？\.[0-9] 前面0-9要不要都可以(紧跟在它前面的单元匹配0次或1次)

$:匹配行末尾 ;$ 匹配行末是；的行

^$：空行

\bth：表示以th开头的行

th\b：表示以th结尾的行

？：紧跟在它前面的单元匹配0或1次ex: [0-9]?\.[0-9]-->x.[0-9] x可有可无 ‘.’需要转义

正则中‘’ 与“”一样的

+：紧跟在它前面的单元匹配1或多次

\*：紧跟在它前面的单元匹配0或多次

{N}：紧跟在它前面的单元匹配N次

{N,}:紧跟在它前面的单元匹配至少N次

{M,N}：紧跟在它前面的单元匹配至少M次，至多N次

### python中的正则：

re.findall("正则式子"，输入的字串)

re.search("xx",f.read());只找第一次出现的结果

re.findall("1[3|5|8|9]\d{9}"，输入的字串) 匹配手机：1开头 第二个数字是3/5/8/9 其他还有9个数字

\d表示是数字，可以写成：re.findall("1[3|5|8|9][0-9]{9}",s)

re.findall("1[3589][0-9]{9}",s) []中间的|有没有都行

pattern = "https://movie.douban.com/subject/[0-9]+/"

urls=re.findall(pattern,html) 从数据很多的网页把链接提取出来

“[^abc]” 匹配abc以外的其他字符

>>> s="ahhhb12345cchhaajjjbb"

>>> re.findall("[^abc]{5}",s) 需要连续有5个才行

['12345']

“^abc$” 以abc开头并结尾的字符串

“a\*” 匹配\*号前的字符0次或多次

“（ABC）{3,5}” 匹配abc字符串，最少3次最多5次

“s” 匹配空白字符。‘\n \t \r’

“((0-9){4})([0-9]{3}) ([0-9]{4})” 分组匹配输入的字符串

例如，将身份证分组识别出来：re.search("([0-9]{4})([0-9]{3}) ([0-9]{4})","14262319950201131d")

按照key，value的方式存储：

re.search("(?P<province>[0-9]{4})(?P<city>[0-9]{2})(?P<birthday>[0-9]{4})","371481199306143242").groupdict()

-{'province': '3714', 'city': '81', 'birthday': '1993'}

re.split("[0-9]{2}","alex11name22jack33nike")

以2个数字为分割符，将此字符串分成4段 返回列表

### c中正则

1-转换

int regcomp(regex\_t \*compile,const char \*pattern,int cflags);

1.struct (编译-子串-->结构体) c语言不认识正则串，可以用struct存起来，这就是编译了

2.正则串

3.标志：扩展正则--忽略大小写--识别换行符号

2-

int regexec(regex\_t \*compiled,char \* string,size\_t len,regmatch\_t add[],int eflags)

1.compiled是上面转换后的结构体指针

2.待正则的表达式

3+4：存正则的数组

## Io

全缓存：填满IO缓存 才进行实际的IO操作

行缓存：在输入/输出中遇到‘\n’ 就进行IO操作

不带缓存：直接进行IO操作 不需等待

缓冲区大小：1024---可以写程序测出来

if(i>=100)

fprintf(stdout,"%d",i);---3字节打印

else if (i>=10)

fprintf(stdout,"0%d",i);--3字节打印

else if (i>=0)

fprintf(stdout,"00%d",i);--3字节打印

行缓存的函数：fgets 返回为null表示出错或到文件末尾 fputs 返回EOF（-1）表示出错，成功返回非负值

怎么区别读取错了还是读到文件尾（feof） int feof（FILE\*s）；到尾返回非零，没到尾返回0

全缓存的函数：fread fwrite

无缓存的：stderr ex:fputs("hello linux",stderr);还是会显示到屏幕上，虽然没有换行符

stdin与stdout是 行缓存的。

以a+方式打开文件：即使进行fseek重定位到开头，写进去的东西还是会放到文件末尾。？？？不知道why

以w+方式打开文件：可以重新定位，但是写进去的东西会把之前的覆盖，还是不能实现将1个文件中的内容倒着写到另1个文件中。

实现方法：

char ptr[200];

FILE \*fp=fopen("test.txt","r");

FILE \*fpnew=fopen("testnew.txt","w");

int ret;

int ch=0;

int offset=0; 先将指针指到最后

while((ret=fseek(fp,offset,SEEK\_END)) ==0) 可以偏移成功--fseek失败则退出

{

ch=fgetc(fp); 读取当前位置数据

offset--; 指针向前偏移

fputc(ch,fpnew); 将源文件指针位置的数据 放到新文件中

}

fclose(fp);

fclose(fpnew);

实现方法：r+ 打开，直接重定位指针，读一下往前偏移下，读的数据存入另一个文件

怎么实现 往源文件加一行数字，读源文件，到某行的时候新加一行，后面接着读----数据存入新文件，最后改文件名即可

{

char ptr[200];

FILE \*fp=fopen("test.txt","r");

FILE \*fpnew=fopen("testnew.txt","w");

char \*ret;

int row=1;

while((ret=fgets(ptr, 100, fp))!=NULL)

{

if(row==4) 到了第四行，先插入第四行的数据 在将第五行的数据写进去

{

fputs("4\n",fpnew);

fputs(ptr,fpnew);

break;

}else

{

fputs(ptr,fpnew);

}

row++;

}

fclose(fp);

fclose(fpnew);

}

重定向：设置输入不从键盘输入，输出不在屏幕显示（例：ls > /dev/pts/5） tty /dev/pts/x 另一个窗口，每次新建都会加1个

平时打开一个文件，系统就会分配1个对应的文件描述符（从2往后排）

./a.out > 1.txt 可以将本来printf到stdout的数据重定向到1.txt文件

cmd 1>a 2>a a 会被打开两次，数据会被覆盖，先放的stdout再放stdeerr

cmd 1>a 2>&1 a只需要被打开一次即可

cmd >&fds 结果输出至fdx --->cmd 1>&fds,将stdout给fds，存到fds中

>filename 清空文件 如果清空的是l类型的软连接，最后请的还是源文件

>>filename 不覆盖的创建新文件。如果源文件已存在，并不会覆盖。

格式化输出：都是行缓存的：

fprintf(file\*,"格式"，参数)； 可以显示到屏幕，也可以写到指定的文件中根据FILE\*。更像是1个格式化写入的函数

sprintf(char \*数组指针，格式，参数)；

fgetc与fput 有缓存但不是行缓存函数，是全缓存的

fopen：可以发现用fopen函数打开文件时，位置指针都是处在文件开头处，包括用a和a+方式打开，只是在第一次写时位置指

针会移动到文件末尾处。并不是有2个指针：读的时候从头读，写的时候从末尾写。

fwrite(buf, size, 1, fp);fread 成对使用并且一般用于结构体数组，因为是一块一块的

\*\*\*\*\*\*\*返回值不是字节数， 是读成功（写成功）的块数

## 库的使用：

### 静态库：

1：gcc -c \*.c 生成的.o与位置有关

2:打包生成库 ar crs libtest.a \*.o

库的使用： gcc test.c -L. -l test (不写lib .a) 或：gcc test.c ./lib/libtest.a

优点：依赖静态库编译自己的程序。程序运行的时候不在需要静态库。程序运行速度快。

缺点：若库要改变了，程序需要重新编译，可能很耗时间，很麻烦

### 动态库：

1：gcc -fPIC -Wall -c hello.c (会生成忽略位置的hello.o）

2：gcc -shared -o libtest.so hello.o bye.o 生成库

LD\_LIBRARY\_PATH:动态库的环境变量

优点：客户的程序不用改变，只需要改变动态库里面的实现即可，客户的程序不用重新编译，直接调用新编译的库就行了

### 差别

动态库运行时会调用库，如果库不在了，程序不能正常运行

动态库不方便移植，生成的代码体积小，需要下载库。

两种库编译的时候都要指定库的位置，编译时都要用到库，其中对于动态库，编译时只会简单引用，不会编译进目标a.out

动态库，执行时要用要库。

1>>把此库位置放入环境变量/lib和/usr/lib（系统库，用户库）的位置

2>>环境变量设置vim ./.bashrc export LIBRARY\_PATH=$LIBRARY\_PATH:修改后重启终端才会有效。

$:source ~/.bashrc （更新设置好的环境变量）

3>>找动态连接器的配置文件/etc/ld.so.conf 写入动态库的路径--ldconfig更新连接器。之后就会自动链接动态库

ldd a.out查看a.out运行需要哪些库，如果库链接不到，程序就无法运行。

用户需要.h+动/静态库就可以用了，不需要源代码。.h里面写了库里面有哪些函数。

## 进程与线程

进程间通信：在用户空间是不可能的。进程在用户空间是相互独立的,得通过内核中的对象。因为进程都可以访问同一内核的。

内核对象都不在用户空间：

管道特点：

管道中数据不可反复读取。一旦读走，管道中不再存在。

采用半双工通信方式（对讲机），数据只能在单方向上流动

无名管道：无文件名 pipe 管道的数据读一下就会消失

所以必须用标准io才能进行通信，因为操作的是文件描述符，不是流指针。

得是父子进程或有亲缘关系的进程，因为要得到fd[]才能通信。此时用的同一个管道。

if(childid==0)

{

while(1)

{

close(pipdef[0]);//close read子进程只获取交互界面数据写到管道的写端

fgets(buf,100,stdin);

write(pipdef[1],buf,strlen(buf));

printf("in child pro,id=%d,buf=%s\n",getpid(),buf);

}

}else if(childid > 0)

{

while(1)

{

close(pipdef[1]);//close write父进程只读写端传来的数据并显示

read(pipdef[0],buf,100);

printf("in parent pri,id=%d,buf=%s\n",getpid(),buf);

}

}

有名管道：有文件名存在于文件系统中

管道文件：p类型------ mkfifo 创建的文件不占磁盘大小，系统会分配inode号

mkfifo函数并没有在内核创建出来管道，只有open myfifo文件的时候才会在内核空间创建管道

要打开myfifo得两个进程都open-myfifo才能打开，只有1个进程不能成功打开myfifo

两个进程要通讯，都必须打开同一个mkfifo才可以。管道的数据读一下就会消失

写进程，不写就阻塞了 读进程，读不到阻塞

pipdef = open("./myfifo",O\_WRONLY); | pipdef = open("./myfifo",O\_RDONLY);

while(1) | while(1)

{ | {

fgets(buf,100,stdin); | read(pipdef,buf,100);

write(pipdef,buf,strlen(buf)); | printf("rcv:%s\n",buf);

} | }

信号： 通过执行1个进程调用（kill函数 ）给另1个进程发信号

信号是用户空间的概念，无法为内核空间的线程传递信号----重要知识点

发送：kill（信号类型，进程号） -9终止进程的时候不会刷新缓冲区的

raise（信号类型）把信号发给自己，相当于kill(9,getpid());

alarm（5）；定时5s后给自己发信号。只会给自己发。

信号的接收：pause（）；让程序到了这里直接暂定运行。

信号的处理：1.signal注册

2.signal（14,SIG\_ING）;忽略此信号

3.signal（14，SIG\_DFL）;默认方式处理此信号

IPC对象：终端命令：ipcs -m/-q/-s 查看内核ipc对象

ipcrm -m id 删除

通过宏IPC\_PRIATE创建的key始终为0

shmget（IPC\_PRIVATE，128,0777）,这种只能实现有亲缘关系的通信

通过ftok创建的key再去shmget（key，128,IPC\_CREAT|0777）; 这种可以实现无亲缘关系的通信。

void shmat（int shmid,const void\*shmaddr,int shmflahg）;

NULL表示系统自动完成映射

shmflag：默认是0，表示共享内存可读可写。

成功返回映射后的地址，失败返回NULL；

共享内存创建后，一直存在于内核中，直到被删除或系统关闭。

共享内存与管道不一样，读取内存中的数据之后，内容仍在内存中。

shmdt：将进程中用户空间的映射地址删除。成功0 失败-1。删除之后这个地址就成空的了，不能再操作了

shmctl:删除内核中的ipc对象（即刚才创建的内核中的共享内存）

int shmctl(shmid,cmd,\*buf);

cmd:IPC\_STAT

IPC\_SET

IPC\_RMID

buf:当cmd为stat或者set时，用于保存要设置的属性。cmd为IPC\_RMID时不需要这个参数

字符串定义的时候可以直接赋值。char a[] = 'sadas';

或者struct node mynode={

.name = "123",

.mystr = "sdfgds", 结构体定义的时候字符串也可以直接赋值。

}

\*\*如果主线程调用pthread\_exit函数退出，这样的话，main进程会等所有线程结束时才会终止。不然一直阻塞住

要想主线程退出其他线程不退出，主线程退出的时候应该用pthread\_exit

return ：返回调用它的函数

exit：退出当前进程-->底层调的\_exit()

线程退出是值是（void\*）的类型

如何避免僵尸线程：（僵尸进程是杀不死的，kill不掉）

pthread\_join

pthread\_detach

pthread\_create之前先设置线程的分离属性

新线程可能在pthrad\_create返回之前就已经结束了的。不一定说函数有了返回值新线程才开始运行。\*\*\*

初始化线程属性：(应先初始化线程属性，再去create线程)-----可以通过修改线程的属性使线程分离，不需要detach了

int pthread\_attr\_init(pthread\_attr\_t \* attr);attrubute(属性)成功0失败errno

int pthread\_attr\_destroy(pthread\_attr\_t \* attr) 销毁线程属性所占用的资源

线程回收：1.分离

2.pthread\_join 如果子线程return或pthread\_exit退出，这个可以收到退出码。如果子线程被取消，就收不到退出码了。

阻塞等待子线程退出、

如果之前的线程用了datach就不要在去join了，都已经自己回收了，用join会收谁去呀？会出错的。

线程杀死：

pthread\_cancle(pthread\_t pid)---->对应进程中的kill函数--成功0，失败errno

线程中得有syscall才能取消，有取消点（函数的底层会调syscall）可以自己加一行pthread\_testcancle();取消点，这个函数是系统调用，本身没啥作用

杀死的进程得回收资源，此时join到的线程的返回值是-1.(固定的，死记住)

线程的退出：在线程函数中不要调用exit函数，会导致整个进程结束。

\*\*\*\*线程可以被同一进程中的其他线程取消

调用pthread\_exit(void\*val)与return x；这2种类似

线程信号：pthread\_kill

线程清理函数：

void \*clean(void \* arg)

{

printf("cleanup:%s \n",(char \*)arg);

return (void\*)0;

}

void \*thr\_fun1(void \* arg)

{

printf("thread 1 start\n");

pthread\_cleanup\_push((void\*)clean,"thread1 first handle");

pthread\_cleanup\_push((void\*)clean,"thread1 second handle");

printf("thread 1 push complete\n");

if(arg)//为1，则直接退出，pop不会执行，则push进去的clean不执行

{

return (void\*)1;

}

pthread\_cleanup\_pop(1);

pthread\_cleanup\_pop(1);

return (void\*)2;

}

void \*thr\_fun2(void \* arg)

{

printf("thread 2 start\n");

pthread\_cleanup\_push((void\*)clean,"thread2 first handle");

pthread\_cleanup\_push((void\*)clean,"thread2 second handle");

printf("thread 2 push complete\n");

if(arg)

{

return (void\*)3;

}

pthread\_cleanup\_pop(1);//先弹出的是后push的函数，所以本函数执行只能打印出cleanup:thread2 second handle

pthread\_cleanup\_pop(0);//第一次push的clean没有被执行，因为只有pop的时候才会被执行到

return (void\*)4;

}

int main()

{

void \* tret;

res = pthread\_create(&tid1,NULL,thr\_fun1,(void\*)1); //线程1的pthread\_cleanup\_pop不会执行则push进去的函数就不会执行

res = pthread\_create(&tid2,NULL,thr\_fun2,(void\*)0);

res = pthread\_join(tid1,&tret);

printf("thread exit code:%d \n",(int)tret);

res = pthread\_join(tid2,&tret);

printf("thread exit code:%d \n",(int)tret);//tret本身是void\*类型的数据，我们返回的数值就是void\*类型的2，直接打印即可

return 0; //如果写成\*（int\*）tret去打印就会报错，因为我们没权限访问地址为2 的地址的数据

}

打印结果：

thread 2 start

thread 2 push complete

cleanup:thread2 second handle

thread 1 start

thread 1 push complete

thread exit code:1

thread exit code:4

线程间通信：可以在用户空间就通信，通过全局变量。

进程：

ps -aux 中的S+状态表示：正在睡眠随时会到R+状态，因为时间片刚好过去。并不能表示程序真的卡住了

R代表正在运行或readyrun状态

I/S代表可中断睡眠，一般最多啦（等待socket链接，等待read，write等等）

O不可中断睡眠，内核态原子操作等等，kill -9都杀不死的进程

T状态：处于gdb调试中的进程状态（暂停或跟踪态），被调试的进程也会进入t状态，不会一直run。

sigstop信号，sigcontinue将此状态变成R，例如GDB调试时等待输入。

D 处于D状态的进程可以通过查看/proc/pid/stack 文件下的调用栈

d状态是进程执行syscall之后在内核空间还未返回，阻塞住了，此时这种进程无法被杀死，因为kill只能给用户空间的

进程发信号，无法给内核空间的线程发信号，而此时进程已经陷入了内核态了

僵尸进程是Z状态，也是杀不死的。D状态也是杀不死的

ps -axo stat pid,ppid,pgid==pgrp,comm 查看进程组号，组id可能与ppid一样也可能不一样

ps j也能查pgid号

ps -p 进程id -o pgrp 看进程所属组id

ps -Lf pid:查看进程中的线程

nice -n 2 ./test 改变优先级（只针对没有运行的进程）

renice 可以更改正在运行的 进程优先级

jobs 查看所有的后台进程

fg 1 将1号后台进程转为前台进程（foreground）

ctrl+z 将当前前台进程转为后台

bg 2 将后台挂起的2号进程重新变为运行态（backgrou）

父子进程，子进程继承父进程所有内容，但是有独立的内存空间。都有变量i，但是I的地址是不一样的

\*\*\*所以不能通过全局变量进行数据交换。

一个进程的进程号在生命周期不会改变，但是父进程可能改变，（父进程死了，子进程变为后台进程，子进程交给init）

进程结束exit 会刷新流的缓冲区 \_exit不会刷新 （这两个包含的头文件不一样）

return和exit()的另一个区别在于，即使在除main()之外的函数中调用exit()它也将终止程序。

exec函数族 让当前进程执行另外的程序（当前内容被指定程序替换）

让父子进程执行不同的程序。

l = list形式,v = vector形式【"ls","-a"】，e = environment自己穿进去环境变量

p = 函数自己根据命令在默认环境变量中查。带p的exec穿进去的第一个参数就不需要带路经了

execl（指定程序名称char \*含路径，）

execlp("ps","ps","-ef",null)

execl("/bin/ps","ps","-ef",null)

char \* arg[] = {"ls","-a","/etc",NULL} #看的是/etc下的文件

execv("/bin/ls",arg)

execvp("ls",arg)

int main() 如下--当前进程执行execl后 ，后面的printf就不会被打印了，此程序执行的变成了ps -au

{

if(execl("/bin/ps","ps","-au",NULL)<0)

perror("execl error!");

//此行永远不会被打印

printf("this message will never be printed!\n");

}

进程回收 wait（&status） 成功返回子进程进程号，失败返回eof。wait会阻塞等子进程退出它才退，

不然就一直等儿子，子进程的exit（10），父进程的status里面就会存10.

也可以wait（null）表示释放pcb就好，不接收返回值。

wait --等任何子进程状态的改变

WIFEXITED(status) 子进程必须通过exit return结束，返回true

WEXITSTATUS(status) 得到子进程返回的exit值

WIFSIGNALED(status) 子进程被信号打断，返回true

WTERMSIG(status) 得到子进程被那个信号终止

火狐浏览器一打开：就可以看到分配了几十个线程（线程池）

nLwp:多少个线程

Lwp：线程号（线程号不是线程id，cpu轮换就是根据线程号轮换的

线程id：进程内区分pthread\_self()获取线程号）

线程共享的资源：

文件描述符表；

signal的处理方式 尽量不要将线程与signal混这用，比较麻烦，不稳定

当前进程的工作目录

用户id与组id

私有的资源：

errno变量

线程id

栈空间，栈指针

信号屏蔽字

调度优先级（跟进程的优先级一样）

ps -aux //WIFSTPEPED(status) 子进程被信号停止，返回true

才能看 //WSTOPSIG(STATUS) 返回让子进程停止的信号

不能printf 出来 //WIFCONTINUED(STATUS) 如果子进程被信号重启，返回true

waitpid（pid，status，option）

pid=-1 任何子进程状态改变

pid>0 等待此进程号对应的子进程状态改变

pid=0 等待和当前进程同组子进程的改变

option = 0 阻塞

option =WNOHANG 不阻塞，指定子进程没结束马上返回0

option = WUNTRACED 子进程暂停，不阻塞，马上返回0

线程：pthread\_self()获取线程号

线程是在进程中创建的，如果main进程结束，线程也会end。

阻塞等待pthread\_join （tid，void \*\*ret）；

传出参数给空的ret，返回后ret里面放的pthread\_exit（ret）

线程通信：同步互斥 p同时信号量-1 v同时信号量+1（读写操作，生产完才能消费）

线程同步的概念： sem\_wait(\*sem) sem\_post(\*sem)

信号量：p操作：申请资源 信号-- v操作 信号量++ 声明在全局变量，便于线程A,B都能访问

2类信号量：无名信号（存在于内存中，主要用于进程的线程间通信，全局变量）

有名信号（线程、进程间通信都可以）

信号初始化：int sem\_init(\*sem,int pshared,int val) sem是信号量对象

0线程间1进程间 初始化量大小

有名信号量 无名信号量

要对一个读进程一个写进程进行PV操作，需要2个信号量，sem\_r初始化为0 sem\_w初始化为1

信号也可以当锁实现，信号是0不能操作，信号是1才可以操作。

【有读信号才能读，有写信号才能写】

函数sem\_wait( sem\_t \*sem )被用来阻塞当前线程直到信号量sem的值大于0

互斥：一次只允许一个（进程或线程）访问共享资源 声明时在全局变量a，b都能访问

访问之前申请锁，访问之后释放锁

mutex：锁子（互斥量） 表示不可读阻塞 表示可写，写完后应该v操作

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);

锁子一定要在线程创立之前初始化，不然线程刚开始还是乱的

pthread\_mutex\_lock（&mutex）

(给共享资源枷锁)若：当前未锁，成功加锁，当前已锁，阻塞等待

pthread\_mutex\_trylock（&mutex) 若当前未锁成功枷锁，当前已锁则直接返回(所以这种需要轮询)

pthread\_mutex\_destory（mutex）

mutex不是强制使用的，但是内核建议你使用(怎么理解呢？)

就是其实A线程用锁访问共享数据var，B线程我就算不获得锁，也可以访问，并不是不能访问，只是内核

建议你B线程访问的时候先获得锁再去访问，防止数据乱掉。

restrict关键字：普通变量int \* m=xxx;如果不想用m去改变指向的内存的数据，我可以int \* p = m;然后用p去改

但是如果加了restrict关键词，就是不允许这样做，你要更改只能用m本身去更改，不能换个名字去改。

枷锁解锁的时机：(很重要)

1.在线程函数内部(如果是main中，在循环的内部枷锁解锁)---否则只能执行到枷锁，执行不到解锁了

2.在访问共享数据之前迅速枷锁，访问完成就迅速解锁--建议这么做。不要访问完了还干别的事情

3.stdout也是共享数据，在线程中printf也是访问共享资源(底层是write到2号fd，print的时候不枷锁，

有可能每个线程打印的话都搅和到一起，打印的话都不通顺了)

4.解锁后稍微睡一下usleep（），让别的线程可以抢占到锁，不然他们不好抢。容易易主

死锁产生的例子：

1.同一线程内部2次获得同一个锁，该锁子只有一个

2.p1有A锁请求B锁，p2有B锁请求A锁

怎么解决：p1用trylock去获取锁，一旦返回获取不到，则释放现在已lock的锁(A)，成全别人

产生死锁的四个必要条件：

互斥条件(Mutual exclusion)：资源不能被共享，只能由一个进程使用。

请求与保持条件(Hold and wait)：已经得到资源的进程可以再次申请新的资源。

非剥夺条件(No pre-emption)：已经分配的资源不能从相应的进程中被强制地剥夺。

循环等待条件(Circular wait)：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。

死锁的避免：

安全序列

银行家算法，保证拿到钱的客户项目能运行完，会还钱。

2.死锁的恢复

（1）最简单，最常用的方法就是进行系统的重新启动，不过这种方法代价很大，它意味着在这之前所有的进程已经完成

的计算工作都将付之东流，包括参与死锁的那些进程，以及未参与死锁的进程。

（2）撤消进程，剥夺资源。终止参与死锁的进程，收回它们占有的资源，从而解除死锁。这时又分两种情况：一次性撤消参

与死锁的全部进程，剥夺全部资源；或者逐步撤消参与死锁的进程，逐步收回死锁进程占有的资源。一般来说，选择逐步撤消的进程

时要按照一定的原则进行，目的是撤消那些代价最小的进程，比如按进程的优先级确定进程的代价；考虑进程运行时的代价和与此进

程相关的外部作业的代价等因素。

有两种方法创建互斥锁，静态方式和动态方式。POSIX定义了一个宏PTHREAD\_MUTEX\_INITIALIZER来静态初始化互斥锁，方法如下：

pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER;

在LinuxThreads实现中，pthread\_mutex\_t是一个结构，而PTHREAD\_MUTEX\_INITIALIZER则是一个结构常量。

动态方式是采用pthread\_mutex\_init()函数来初始化互斥锁，其中mutexattr用于指定互斥锁属性（见下），如果为NULL则使用缺省属性。

pthread\_mutex\_destroy()用于注销一个互斥锁，API定义如下：

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex)

销毁一个互斥锁即意味着释放它所占用的资源，且要求锁当前处于开放状态。由于在Linux中，互斥锁并不占用任何资源，因此LinuxThreads中

的 pthread\_mutex\_destroy()除了检查锁状态以外（锁定状态则返回EBUSY）没有其他动作。

读写锁：pthread\_rwlock\_xxx

口诀：写独占，读共享（各线程都是读的时候才共享\*\*）

写锁优先级高（如果别的线程有请求写锁的有读锁的，则读锁就不共享了，先让写线程获得写锁）

pthread\_rwlock\_init(); 只有一个锁

pthread\_rwlock\_destroy

pthread\_rwlock\_rdlock pthread\_rwlock\_rdtrylock

pthread\_rwlock\_wrlock pthread\_rwlock\_wrtrylock

pthread\_rwlock\_unlock

条件变量：与互斥锁配合使用

pthread\_cond\_xx 本身不是锁，但是可以造成线程阻塞，通常与互斥锁配合使用

pthread\_cond\_wait();难

pthread\_cond\_init

pthread\_cond\_destroy

pthread\_cond\_wait

1.使用前定义mutex 初始化mutex 锁住mutex 定义cond 初始化cond

2.阻塞等待条件变量满足+释放已掌握的mutex

3.被唤醒，函数返回时，接触阻塞，重新lock互斥锁

pthread\_cond\_signal

1.唤醒至少一个阻塞在条件变量上的线程

pthread\_cond\_broadcast

1.唤醒所有阻塞在cond上的线程

静态初始化cond方法：不需要init函数，只要：pthread\_cond\_t mycond=PTHREAD\_COND\_INITIALIZER

exp：

while(1) 子线程

{

//pthread\_mutex\_lock(&g\_mutex);

pthread\_cond\_wait(&g\_cond,&g\_mutex);//等待条件成熟，即g\_cond,g\_mutex都获得，程序向下运行，否则

//释放互斥量，继续阻塞(不消耗资源)等待

printf("rcv: %s\n",buf);

//pthread\_mutex\_unlock(&g\_mutex);

注意第一行与第四行可以屏蔽的，因为pthread\_cond\_wait本来就有获取不到就阻塞的功能，不需要在

线程中再去用mutex\_lock抢占锁了

}

主线程只负责给条件与资源，子线程能占到就往后执行，占不到就阻塞等待

while(1) 主线程

{

fgets(mybuf,100,stdin);

//pthread\_mutex\_lock(&g\_mutex);//也可以屏蔽

memcpy(buf,mybuf,strlen(mybuf));

pthread\_cond\_signal(&g\_cond);//唤醒g\_cond的等待

pthread\_mutex\_unlock(&g\_mutex);

}

信号量：互斥锁的升级版(可用于进程线程间)

互斥锁只能供1个线程使用，信号量初始化为N，就可以供N个线程使用，因为每次抢占只会-1，

互斥锁减1下就没了，信号量可以减N次9

同步：不同行业有不同的理解

设备间：同一时间参考--usart uart

编程中：按照先后预定的次序执行代码就是同步。彼此有依赖关系的调用不应该同时发生，同步就是阻止那些‘非要同时发生’的事件。

信号是同步的：1个线程写入数据，另1线程读取。如果我不写，你不能读。是对两个动作的同步（你走一下我走一下）

锁是保护共享资源的，我访问你就不能访问。你访问我就不能访问。

## 信号

void sig\_alarm(int sig)

{

printf("sig is %d, sig\_alarm is called\n", sig);

}

int main(int argc, char \*argv[])

{ 35 函数名

signal(SIGALRM, sig\_alarm); 注册给SIGALRM注册对应的处理函数

alarm(5); // 5秒后，内核向进程发出alarm信号， 执行对应的信号注册函数 否则它默认是会杀死进程的

sleep(20); //\*\*\*\*\*\*\*\*\*\*必须大于alarm的5s，如果没有这句话，程序瞬间就执行完了。

printf("end!\n");

return 0;

}

pause（） 暂停挂起进程，等待信号（信号有自己的处理函数， 目前学的信号打断默认都会直接杀死进程

有些信号让进城状态改变，有些例如-9会杀死进程））这种打断进程就不能执行到exit最后这句了

kill -l查看所有信号 注意所有信号对进程都有默认的处理方式

SIGKILL SIGSTOP不能被忽略 也不能被捕捉

SIGSTOP是暂停进程信号，不是终止

函数kill(pid,sig) pid可以指定getppid给父进程。。。。

raise(sig)只能给自己所在的进程发---

void abort();自己给自己发SIGABRT信号，终止进程并产生core文件。

SIGALRM：alarm（5）到时间就会发出的信号，发出信号如果有注册过这个信号的处理函数，就会跳过去执行

一个进程有且只有一个定时器，无论进程处于哪种状态，alarm都在计时\*\*\*\*

SIGINIT：在终端执行Ctrl+c会触发此信号，发出信号如果有注册过这个信号的处理函数，就会跳过去执行

ctrl+z：sigstop信号

ctrl+\：退出信号 3号信号

SIGCHILD：有子进程退出会发这个信号，可以提醒父进程去回收啦。

SIGSEGV 段错误

进程收到信号的三种处理方式：

默认：如果是系统默认的话，那就会终止这个进程

忽略 ：信号来了我们不处理，装作没看到 SIGKILL SIGSTOP 不能忽略

捕获并处理 ：当信号来了，执行我们自己写的代码（捕获信号这个动作是需要我们完成的） SIGKILL SIGSTOP 不能捕获

demo：

创建子进程代表售票员，父进程代表司机 ，同步过程如下：

售票员捕捉 SIGINT(代表开车)，发 SIGUSR1 给司机，司机打印(“let’s gogogo”)

售票员捕捉 SIGQUIT(代表停车)，发 SIGUSR2 给司机，司机打印(“stop the bus”)

司机捕捉 SIGTSTP(代表车到总站)，发 SIGUSR1 给售票员，售票员打印(“please get off the bus”)

int main(int argc, char \*argv[])

{

pid = fork();

if (pid > 0) /\* parent driver 父进程代表司机\*/

{

signal(SIGTSTP, driver\_handler); //父进程收到信号的处理

signal(SIGINT, SIG\_IGN); //收到此信号 ingore

signal(SIGQUIT, SIG\_IGN); //收到此信号 ingore

signal(SIGUSR1, driver\_handler);

signal(SIGUSR2, driver\_handler);

while (1) pause();

}

else /\* child saler 子进程代表售票员\*/

{

signal(SIGINT, saler\_handler);

signal(SIGQUIT, saler\_handler);

signal(SIGTSTP, SIG\_IGN); //收到此信号 ingore

signal(SIGUSR1, saler\_handler);

signal(SIGUSR2, SIG\_IGN); //收到此信号 ingore

while (1) pause();

}

return 0;

}

void driver\_handler(int signo)

{

if (signo == SIGUSR1)

printf("Let's gogogo!\n");

if (signo == SIGUSR2)

printf("Stop the bus!\n");

if (signo == SIGTSTP)

kill(pid, SIGUSR1); //在父进程中 fork返回的id就是子进程id号 可以直接通过pid给子进程发数据

}

void saler\_handler(int signo)

{

pid\_t ppid = getppid(); //子进程想给父进程发数据 需获取父进程id 才可以发

if (signo == SIGINT)

kill(ppid, SIGUSR1);

if (signo == SIGQUIT)

kill(ppid, SIGUSR2);

if (signo == SIGUSR1)

{

printf("please get off the bus\n");

kill(ppid, SIGKILL); //子进程去kill 父进程 自己也退出

exit(0);

}

}

## SYSTEM-V IPC通信：

每个IPC对象有唯一ID。

IPC一旦创建一支存在，除非主动删除（显式删除）

每个IPC对象有一个关联的key（key为0表示是私有的）。。1个key可以有很多id

linux命令：ipcs 查看当前ipc对象

ipcrm 删除ipc对象。

key\_t key; 0-127 ./a.c 'a'

key = ftok(".",100);//生成key。路径+文件名+工程名‘m’也可以

创建共享内存 只用ipc\_private创建出来的key默认为000

FLAG

创建新的，或者获取已有的共享内存。如果key没有任何共享内存，则创建（开辟），

如果有对应的，说明别的进程已经创建好了，可以直接获取。

返回映射后的地址=shmat(shmid,NULL,0) 映射 有了地址可以写memcopy。fgets。scanf

null系统分配 0表示可读可写SHM\_RDONLY只读 读：puts（地址）

shmdt（shmaddr）取消共享内存 传地址（用户空间地址删除）

int shmctl(shmid,int cmd,jiegouti \* buf)（内核空间地址删除）

IPC\_STAT IPC\_SET IPC\_RMID 这时共享内存没人用就会自动删除了候只是添加删除标记，等

获取属性存到buf 设置将buf设置到属性 删除共享内存ID用不着buf，传null

得到地址后可以调用含有地址的读写函数。往当前内核的地址空间写东西。

多个进程也可以映射到同一块实际物理地址空间，实现数据共享，只不过需要加保护措施，否则A进程写数据，B进程也写，就乱了。可以利用信号量保护。

多个进程使用共享内存通信时，创建者只需要1个，谁先运行谁创建，其他进程直接获取共享使用，实现通信。

读进程，会一直读，因为没有信号灯集去控制

while(1)

{

printf("rcv:%s\n",addr);

memset(addr,0,strlen(addr));

sleep(1);

}

写进程，等待stdin输入：每次有了输入则写到共享内存addr，读进程就能读出来数据了

while(1)

{

fgets(buf,100,stdin);

memcpy(addr,buf,strlen(buf));

}

### 信号灯集：---保护共享资源

使用信号灯集访问临界资源（共享内存）处理上面的一直循环打印的问题

https://blog.csdn.net/qq\_22207951/article/details/105793096 参考此博客

semctl(semid,num,IPC\_RMID);//删除信号灯num，如果semid灯集中有x个信号灯，需要删除x次，因为每次只能删除一个灯

semid = semget(SEM\_KEY, 2, IPC\_CREAT | 0600);//创建2个信号灯，中间入参就是要创建的信号灯个数

semctl(semid, 1, SETALL, sem\_args);//SETALL代表设置信号集中所有的信号量的值。

有了SETALL则The argument semnum is ignored.

semctl(semid,1,SETVAL,mynum); 设置第2个信号灯的值

semid = semget(key,3,IPC\_CREAT|0600);//给id为semid的信号灯集中创建3个信号灯，其实创建一个灯就可以了

arg.val = 1;

/\*\*\*对灯集合中的0号信号灯设置初始值\*\*\*/

ret =semctl(semid,0,SETVAL,arg);返回<0设置失败

ret =semctl(semid,0,GETVAL,arg);获取semid信号集中的灯0的值

shm\_read.c信号灯集与共享内存实现

union semun {

int val;写进信号灯中的数值，0或1就行--2值

struct semid\_ds \*buf;

unsigned short \*arrary;

}mynum; 不知道为什么不能直接union semun mynum去定义变量

/\*\*\*对信号量数组semnum编号的信号量做P操作\*\*\*/

int P(int semid, int semnum)semnum表示对哪一个信号灯去P

{

struct sembuf sops={semnum,-1, SEM\_UNDO};

return (semop(semid,&sops,1));

}

/\*\*\*对信号量数组semnum编号的信号量做V操作\*\*\*/

int V(int semid, int semnum)semnum表示对哪一个信号灯去V

{

struct sembuf sops={semnum,+1, SEM\_UNDO};

return (semop(semid,&sops,1));

}

int main()

{

char buf[100];

int pipdef;

key\_t key; // 0-127 ./a.c 'a'

key = ftok(".", 100); //鐢熸垚key銆傝矾寰?+鏂囦欢鍚?+宸ョ▼鍚嶁€榤鈥欎

char \*addr;

int shmid = shmget(key, 128, IPC\_CREAT | 0664);

addr = shmat(shmid, NULL, 0);

semid = semget(key, 1, 0666); //试图直接获取信号灯集ID

if(semid == -1)//获取信号灯集id失败

{

if(errno == 2)

{

if((semid=semget(key, 1, 0666|IPC\_CREAT))==-1)//创建信号灯集（只有1个灯），会返回新的ID

{

perror("init create sig fail\n");

return -1;

}

mynum.val = 1;

if(semctl(semid,1,SETVAL,mynum) < 0)//为信号灯集中的灯0赋值1

{

perror("init arg fail\n");

return -1;

}

}

else

{

perror("init 2 sig false\n");

}

}

while (1)

{

P(semid, 0);//-灯集中的0号灯的值

printf("rcv:%s\n", addr);

memset(addr, 0, strlen(addr));

V(semid, 0);

}

return 0;

}

shm\_write.c

int main()

{

semid = semget(key, 1, 0666); //试图直接获取信号灯集ID

if(semid == -1)

{

if(errno == 2)

{

printf("get sig fail\n");

//signal not exist ,so create it

if((semid=semget(key, 1, 0666|IPC\_CREAT))==-1)

{

perror("init create sig fail\n");

return -1;

}

mynum.val = 1;

if(semctl(semid,0,SETVAL,mynum) < 0)

{

perror("init arg fail\n");

return -1;

}

}

else

{

perror("init 2 sig false\n");

}

}

while (1)

{

P(semid, 0); //消耗信号灯集中的0号灯的数值

fgets(buf, 100, stdin);

memcpy(addr, buf, strlen(buf));//将键盘内容赋值给共享内存映射的地址

V(semid, 0); //v 0号灯的数值

}

}

消息队列几种ipc通信的第一步都是创建ftok--->key关键字

根据关键字创建本类型（内存、队列）获得句柄 id

操作消息队列就靠这个id

消息队列id = msgget(key,IPC\_CREAT|0666）

私有的key IPC\_PRIVATE或 ftok（多个进程都要访问用ftok）

msgflag：IPC\_CREAT|0666:如果消息队列不存在则创建，若存在返回id

向消息队列发送消息：消息的传送靠的是一个结构体（发送接收都通过这个MSG）

成功0失败-1 msgsnd（msgid，const void\*msgp，size，flag）0或IPC\_NOWAIT 堵塞不堵塞

消息缓冲区地址 消息正文长度

通信前先设置消息格式：一个结构体struct {

long mtype,//消息类型

char mtest[64]//消息正文

}MSG; //

#define LEN (sizeof(MSG)-sizeof(long)) 正文长度

从队列接受消息： 如果是0按照消息顺序接收 要收的mtype

msgrcv（msgid，const void\*msgp，size，long mtype，flag）标志位：0或IPC\_NOWAIT

size要稍微大，不然会丢失信息

控制消息队列：

0或-1 --msgctl（msgid,int cmd,jiegouti \* buf）

IPC\_STAT IPC\_SET IPC\_RMID

### uinx域socket

管道太老了，现在一般进程间通信用的都是socket与共享内存、消息队列。

共享内存效率最高，但是没有枷锁机制，所以一般与信号灯集一起使用。

client.c

#define UNIX\_DOMAIN "/home/zhangmiaoling/test/socket/UNIX.domain"

int main(){

int connect\_fd;

static struct sockaddr\_un srv\_addr;//服务端的地址是sockaddr\_un结构体

// creat unix socket--创建通信句柄

connect\_fd=socket(PF\_UNIX,SOCK\_STREAM,0);

srv\_addr.sun\_family=AF\_UNIX;

strcpy(srv\_addr.sun\_path,UNIX\_DOMAIN);//字符串赋值需要strcpy

//connect server 客户端要链接服务端的结构体地址

ret=connect(connect\_fd,(struct sockaddr\*)&srv\_addr,sizeof(srv\_addr));

while(1)

{

strcpy(send\_buff,"message from client");

//send info server将sendbuf中的内容发送过去

write(connect\_fd,send\_buff,sizeof(send\_buff));

}

}

server.c

#define UNIX\_DOMAIN "/home/zhangmiaoling/test/socket/UNIX.domain"

int main(){

static char rcv\_buff[1024];

struct sockaddr\_un clt\_addr;

struct sockaddr\_un srv\_addr;

listen\_fd=socket(AF\_UNIX,SOCK\_STREAM,0);

// set srv\_addr param

srv\_addr.sun\_family=AF\_UNIX;

strncpy(srv\_addr.sun\_path,UNIX\_DOMAIN,sizeof(srv\_addr.sun\_path)-1);

unlink(UNIX\_DOMAIN);

//bind sockfd&addr 将socketfd与srv\_addr绑定

ret=bind(listen\_fd,(struct sockaddr\*)&srv\_addr,sizeof(srv\_addr));

//listen sockfd

ret=listen(listen\_fd,1);//服务端建立监听

//have connect requst use accept

len=sizeof(clt\_addr);

//accept的时候会从clt\_addr中获取客户端信息

com\_fd=accept(listen\_fd,(struct sockaddr\*)&clt\_addr,&len);//阻塞等待与客户端链接成功，否则阻塞

//read and printf client send info

while(1)

{

read(com\_fd,rcv\_buff,sizeof(rcv\_buff));

printf("message from client %d : %s\n",num,rcv\_buff);

}

close(com\_fd);关闭与客户端链接的fd

close(listen\_fd);关闭服务端的fd

unlink(UNIX\_DOMAIN);删除通信的socket文件

return 0;

}

## 守护进程：

创建流程：

1.创建子进程，父进程退出

2.在子进程里面操作

（1）setsid 设置会话组 这样就脱离终端了

（2）改变子进程工作目录，chdir（“/”） changedir（改变当前进程的工作目录）

（3）设置文件权限 umask（0） 1是标准输出

（4）关闭所有文件描述符 close 循环，getdtablesize--获取当前进程占用的文件描述符数量

for (fd = 0, fdtablesize = getdtablesize(); fd < fdtablesize; fd++)

close(fd);

（5）进入需要的循环while（1） 避免浪费资源，0,1,2开着没啥用

编译阶段：

预处理pre 展开宏并且删除注释 -E

编译compile【生成.s】 -S

汇编asseble【生成.o】 -c

链接link【拼接多个.o生成可执行.elf此时包含调试信息（只能在linux中运行）】

objcopy：剥离无用的调试信息生成.bin可以在板子运行

## ARM

1. c执行函数fun（1,2）参数怎么传递过去的，利用栈，从右向左将2,1压栈，出栈1,2赋值给fun

c与汇编混合编程，怎么调用汇编函数传参。利用寄存器，寄存器比栈速度快。x86寄存器少所以利用栈传参。arm寄存器多，但是 参数传递只能用用寄存器r0~r3，多于4个参数，多出来的参数用栈。

1. r0用来接函数的返回值，函数的返回值只有1个。用r0就够了。把返回值封装成1个结构体返回也行

pc==r15.

lr==r14.

add r0,r0,r1===r0=r0+r1 汇编函数执行完最后一句mov pc,lr lr存的函数返回地址，函数执行完

告诉pc去lr的地址往后执行。

时钟周期：1s/频率 例如：1s/1Ghz == 1ns 1s大约能执行10^9个指令

thumb指令集：16位--arm指令集：32位

存储器访问速度：

寄存器->cache（sRAM随机静态存储器）->内存(DRAM动态随机存储器:DDRAM-SDRAM)->硬盘(Flash：norflash-bios，bootloader) ->网盘

DDR:(SDRAM)同步动态内存（sychonize）是随机访问存储器：访问那个点数据的事件都一样。怎么做到的呢

硬盘是顺序访问存储器：开头访问快，后面访问得等磁盘转过去才能访问。

芯片里面一般不放大电容电阻，体积太大

DRAM:动态内存（保存一位用1个电容）

SRAM:静态内存（保存一位用4-6个晶体管）

### 汇编指令

操作码（后缀s码）+目的寄存器+源寄存器+立即数 跟了s码才会影响cpsr的nzcv，不然不会影响：adds r0，r1，#ff

\*指令执行前判断cpsr条件码--指令执行后判断s码影响cpsr

相等跳转，检查cpsr beq aaa。。。先判断是否相等才跳转（条件码）

### Bootloader：

不属于操作系统，是引导操作系统的介质。用.c .s编写的

boot阶段：

关闭看门狗、中断 WTCON寄存器某位。。CPSR 的I/F位中断

初始化时钟---倍频到某一个主频，为外设分频

初始化串口

初始化内存，主要指ddram

初始化硬盘nandflash

loader阶段：

从Emmc/Flash把kernel搬运到到指定内存中，（自搬移）跳转到kernel所在地址。

汇编阶段与C阶段分析：

汇编阶段：

设为svc模式，关闭中断、看门狗、mmu、cache 中断向量表

基本硬件初始化（时钟、串口、flash、内存）

设置栈

跳转到c main

c阶段：

大部分硬件初始化

搬移内核到内存

运行内核

.lds 连接文件（查看每段代码的位置）

编译C代码不会create lds文件，因为libc库中包含链接file

编译汇编代码 需要自己写lds文件 才可以生成二进制

内核启动方式：

自启动模式，u-boot设置自启动参数。

bootcmd：setenv bootargs root=/dev/nfs nfsroot=192.168.1.101:/source/rootfs rw console=ttySAC2,115200 init=/linuxrc ip=192.168.1.250

交互模式：在u-boot中开发板与pc通过串口调试，网口tftp下载pc的内核，rootfs，然后自己用指令启动

### Cpu内部寄存器

r0-r12：保存数据临时变量

r13：sp-栈顶地址

r14：lr-函数返回地址，函数调转时存放函数运行完后的地址，调用完函数后，pc调到这里继续往下执行。

cpsr：当前程序状态寄存器

spsr：备份当前程序状态寄存器[中断时自动调用],不进中断没用

3级指令流水线：最高效

单片机中寄存器地址：

寻找一个寄存器地址的时候,先确定基地址--在 memory map章节查找 ,再找本寄存器偏移 即可找到对应的唯一地址

Gpio的使用：

1.使能gpio模块

2.确定工作模式

3.设置方向 读 写

4.若是写 ,写1/0 若是读 读管脚电平

### 串口裸机驱动：

串口为啥1byte1byte的发，因为可以减小收发双方时间基准不一样长时间通信有可能数据差出1bit，后面的所有数据就会出错，减小累计误差

arm-none-linux-gnueabi-ld start.o -Tmap.lds -o led.elf

arm-none-linux-gnueabi-objcopy -O binary -S led.elf led.bin 大写-O -S

·· 【1】只要把指定pin的con设为输出，data寄存器设为高电平就ok了

异步通信：数据传送是以字符为单位的，字符间的传送是完全异步的，字符中位于位之间同步（字符间异步 字符内部各位同步），不知道数据多会发，多会收，让你发就发，别人发就收

同步通信：传输以数据块为单位，字符之间，字符的位于位之间都是同步的。

\*\*uart2：为什么又要设置串口gpio的时钟又要设置uart外设的时钟。

固定用法，可以通过keil生成，pin想作为uart就要初始化pin为特定功能，然后再初始化uart控制器特性

两者都是使能 使能gpio引脚，引脚就能用了，使能uart，uart就能用了。

【1】配置第6章一对管脚pin的con为uart模式

【2】uart章节一共有好几个uart，配置对应的ulcon配置收发数据格式

【3】波特率

【4】数据收发方式-中断或轮训--一般都配置中断

【5】收发存储数据寄存器，UTxHn 0x13820020把要发送的数据放到这个寄存器,【就算发出去了，后面不用管了】

URxHn 0x13820024 【把数据放到这里就算读出来了，后面不用管了】

【6】状态：何时收何时发 UTRSTATn（0x13820010）：read这个寄存器

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*timer 定时器\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

硬件时钟源：RTC->独立于CPU与其他硬件 时钟源可以来自内部低速时钟或外部低速时钟

只要单片机有电或后备寄存器有电，则rtc可以一直运行不间断。

linux开机时读取的时间就是rtc中的时间，关机时再将当前os的时间同步到rtc中。

为什么说看门狗与pwm都是定时器，因为他们配置的结果就是产生一个频率，实际就是产生一个固定的时间长度，

你配置输出频率的占空比还是基于这个时间，这个频率可以精准配置时间的。exynos配置的时候好几步都是弄频率的。

看门狗：WTCON = WTCON |1 |0x7<<3 |249<<8;

配置分频，产中断还是产复位信号。还是只当个定时器来用

WTCNT = 15625; //不要老想着位操作 ，赋值的时候直接赋值就好了。这个里面的数字会递减

看门狗的设置跟管脚没有关系，是芯片内存本来就有的功能。

现在只是为了学习看门狗才简单的配置这个寄存器。配置好寄存器，板子上电5s会自动reset，一般我们会在while1中喂狗，除非异常才会reset。

counter计数器是8-16bit的，总大小是是有限的

pwm：（其实是可以通过高低电平延时来控制频率的，但是延时函数会占用cpu工作时间，在控制舵机的时候别的事情都干不了。

共有5个pwm定时器，其中4个外接到了管脚有驱动能力Xpw0-3TOUT0，

外部输入的频率是100mhz，经过2级分频25----4----变成1m，设置2个寄存器

TCNTB0 = 2500; 2500的波形个数实际就设置了输出的波形是400hz，1000000/400=2500。2500个1m的波形才能产生1个400hz的波形

TCMPTB0 = 1250; //2500\*1/2---实际就设置了占空比为1/2

软中断：（swi）是用户命令，得先切换到用户状态才能使用这条命令

1板子上电会先运行b reset：我们需要在reset函数中设置好栈（给sp赋值栈底），svc模式栈就初始化好了

2现在改变状态为user态，直接执行swi（也可以随便给r0-r12寄存器赋值，等会中断完看能不能复原）就跳转到了swi的处理函数

3刚进入这个模式，板子状态会恢复为svc模式，lr保存刚才的pc下条指针，spsr保存刚才cpsr的值，

'-r12，lr压栈；分析是那号中断（swi有很多跳转号的），switch case对比一下往哪个子函数跳，然后bleq fun（这里才真正步入了中断子函数，因为要回来所以bl）；

4在中断fun中随便写几条命令改变r0-r12值，然后回去（mov pc lr）

5现在又回到了swihandler，我们可以出栈了，中断子函数都完事了。把刚才压进去的lr直接出栈给pc就行了

6观察现象，现在各个寄存器的值应该回到了跳转之前的状态。

### 硬件中断编程：

只有几个pin脚可以触发中断，以前的51单片机可以实现中断的腿很少的，本次的exynos4412有17个管脚可以实现，具体引出来几个腿看具体板子

第一阶段：1.设置pin脚寄存器，设为中断。看一下这个中断对应soc的那个中断号

2.配置中断的触发方式：（跳变沿还是高低电平）

3.取消中断掩码

第二阶段（都在GIC中断章节）：1.使能对应的中断号（在中断源章节看pin对应的那个）

2.把这个号交给cpun管理（一共有4个cpu）

3.设置中断优先级（只要大于cpu的中断处理阈值就好，要不人家不管）

4.使能cpu0中断

5.设cpu0的中断阈值0xff，（0优先级最好，255最小），这样设表示cpu0所有中断都要处理

6.使能GIC总开关

第三阶段（在irq\_handler中处理中断）：1.取对应的中断号对比，switch case：中处理

2.清除管脚中断标志位（在gpio章节EXT\_INTXX\_PENDD）

3.清除GIC的中断标志

4.结束中断

底层的寄存器。【防止编译器优化】（每次拿值不要从内存中拿，要从寄存器拿）

例如：两句话

i = 1;i = 2;如果不加volatile，编译器会将i=1过滤掉不执行。

但是对于reg来说，i=1会有动作，i=2也会有动作，编译器不可以优化的。

(\*(volatile unsigned int\*)0x11000c40)地址里的值是volatile的

提示编译器对象的值可能在编译器未监测到的情况下改

cpu的7种模式：user,sys,und(未定义指令)，svc（管理模式），abt（abort指令预取终止或数据访问终止），irq,fiq 。每种模式有专用的reg，不同模式可访问的reg不一样

## linux系统移植：

对于裸机程序可以直接go 0x20008000执行裸机代码

对于uimage（zimage+头4m），需要引导才能启动

### 裸机程序与嵌入式程序区别：

裸机：

1.运行速度快，直接操作硬件管脚（有可能损坏硬件）。映射到内存

2.一般裸机编程的板子都比较简单，设备成本低

3.处理问题单一，单任务执行，单进程—借助rtos也可以多线程

4.打印东西还得自己写函数

5.开发环境，下载器，串口，keil

系统：

1.多任务，开发效率高

2.不会损坏硬件，对硬件的操作都是基于驱动，（只要驱动写得好）linux会驳回你的非法操作

3.方便，可以使用linux的库（直接printf）io库

4.mmu内存管理单元负责地址映射，多进程不会内存越界

5.u-boot启动，开发环境，pc，串口线，网线，sd卡，emmc，tftp,nfs

### Tftp安装

tftpd-hpa 安装包

pc：tftpd（d表示damen守护进程）：

-->etc/default/tftpd-hpa

linux@farsight:~$ sudo vim /etc/default/tftpd-hpa

# /etc/default/tftpd-hpa

TFTP\_USERNAME="tftp"

TFTP\_DIRECTORY="/tftpboot" 设置tftp目录

TFTP\_ADDRESS="0.0.0.0:69"

TFTP\_OPTIONS="-l -c -s"

TFTP\_DIRECTORY 处可以改为你的 tftp-server 的根目录，参数-c 指定了可以创建文件

下载安装包后查看默认的存放tftp文件路径，之后可以在客户端通过tftp命令链接server

通过get可以下载server中tftp文件路径中的文件至clinet

### u-boot制作

bl1+bl2+uboot.bin = uboot-fs4412.bin

我们选用u-boot（这个是在源代码基础上改改再加上三星提供的bl0+bl1加密规则就可以移植到自己的系统中）

4 实验：制作SD卡--->u-boot-fs4412.bin 制作带uboot的sd卡 --sd卡插到笔记本去烧写

sudo dd iflag=dsync oflag=dsync if=/home/farsight/u-boot-fs4412.bin of=/dev/sdb seek=1

1.指定产品BOARD 编译为u-boot.bin

不能直接烧，要添加三星的加密方式，修改makefile

不能直接make，要用脚本./build这个是写好的带加密方式的编译

在start.s中 lowlevel设置临时栈，关闭看门狗

初始化串口、（不配置串口不能打印）

3.网卡

4.emmc、（\*\*\*\*\*\*\*【】）（movi write 命令都不能用）

先用sd卡启动（不然连串口都没有），将uboot下载到内存，然后write到emmc，这样直接从emmc启动u-boot可以直接用了

2.linux内核+设备树（这两在linux3.0之前是一体的）每次编译移动，两者都要做

解压的源码包：make mrproper--->make exynos\_defconfig(给自己板子弄合适的配置)

1.配置内核乱七八糟的

2.配置网卡，否则nfs不能用，则rootfs不能用

内核启动： 2汇编

3打印内核启动信息->初始化各模块->挂载根文件系统->启动第一个用户进程init

内核调试：printk()-->打印高于默认打印级别的信息

echo "7 4 1 7" > /proc/sys/kernel/printk 开启内核打印到屏幕

arm-none-linux-gnueabi-addr2line：很多2表示to的意思。寻找addr所在的文件行数

oops:pc:[<0xxxxxxxxx>].报错的时候回显示行号---写入NULL地址数据，访问NULL指针

解压内核code前用puts

解压内核时用putstr

内核解压后 uart初始化之前 printascii

uart初始化之后printk

uboot制作过程步骤： 1、确认第一条指令有运行到（点灯法）

每步都在初始化硬件 2、实现串口输出

加载引导内核到内存 3、网卡移植

4、FLASH移植（EMMC） 1.1初始化EMMC 1.2添加相关命令（movi write。。）

PC端：配置tftp服务（传电脑的-要移植的东西-到板子的内存，然后板子可以烧到emmc，以后就不用接电脑了）

配置nfs服务：实现象windows与linux的文件挂载功能的挂载，共享文件

基本没有soc自带网络功能，一般都没有网络功能。都有uart。spi。iic

用的DM9000，网络有专门的芯片，可以通过驱动网络芯片实现soc与dm9000通讯就好

块设备是有文件系统的，fd = open（块），读写操作进行访问、写入

将内核 文件系统 dtb存在emmc 以后就可以从emmc启动板子

- tftp 410000 uImage movi write kernel 410000

- tftp 410000 exynos.dtb movi write dtb 4100000

- tftp 410000 ramdisk.img movi write rootfs 4100000

- emmc分几块：kernel dtb rootfs 还有3个好像

搬动拨码开关为从emmc启动 设置从emmc启动的启动参数

- setenv bootcmd movi read kernel 410000;movi read dtb 420000;movi read rootfs 430000;bootm 41000 430000 42000;先

加载内核，接下来fs，（没fs你在哪里存放设备树呀），最后设备树

设置网络启动：

# setenv serverip 192.168.9.120

# setenv ipaddr 192.168.9.233

# setenv bootcmd tftp 41000000 uImage\;tftp 42000000 exynos4412-fs4412.dtb\;bootm

41000000 - 42000000

#setenv bootargs root=/dev/nfs nfsroot=192.168.9.120:/source/rootfs rw console=ttySAC2,115200 init=/linuxrc ip=192.168.9.233

u-boot 移植流程：

1.修改makefile

ifeq (arm,$(ARCH))

CROSS\_COMPILE ?= arm-none-linux-gnueabi- 既交叉编译工具链 开头部分

#endif

2.查看cpu是否支持 U-boot 已支持，见 arch/arm/cpu/armv7/exynos/

3.指定产品board--找一个最类似的board进行修改--这里参考的 board/samsung/origen/

$ cp -rf board/samsung/origen/ board/samsung/fs4412---拷贝一份源代码

$ mv board/samsung/fs4412/origen.c board/samsung/fs4412/fs4412.c--修改源程序中的origen.c为fs4412.c

$ vim board/samsung/fs4412/Makefile 修改新增的board的 makefile 修改 origen.o 为 fs4412.o

$ cp include/configs/origen.h include/configs/fs4412.h 修改.h文件

$ vim include/configs/fs4412.h

修改 origen 为fs4412

#define CONFIG\_SYS\_PROMPT "ORIGEN #"

为 #define CONFIG\_SYS\_PROMPT "fs4412 #"

修改 #define CONFIG\_IDENT\_STRING for ORIGEN

为 #define CONFIG\_IDENT\_STRING for fs4412

#vim boards.cfg

参考

origen arm armv7 origen samsung exynos

并在后面新增

fs4412 arm armv7 fs4412 samsung exynos

编译u-boot：$ make distclean $ make fs4412\_config $ make 能编译成功 但是该文件还不能在我们板子上运行，

我们需要对 u-boot 源代码进行相应的修改

怎么实现u-boot的串口输出：

修改board/samsung/fs4412/lowlevel\_init.S

添加临时栈：--设置栈

在lowlevel\_init: 后添加：ldr sp,=0x02060000 @use iRom stack in bl2、

关闭看门狗：

beq wakeup\_reset：添加

#if 1 /\*for close watchdog \*/

/\* PS-Hold high \*/

ldr r0, =0x1002330c

ldr r1, [r0]

orr r1, r1, #0x300

str r1, [r0]

ldr r0, =0x11000c08

ldr r1, =0x0

str r1, [r0]

/\* Clear MASK\_WDT\_RESET\_REQUEST \*/

ldr r0, =0x1002040c

ldr r1, =0x00

str r1, [r0]

#endif

添加串口初始化部分代码：

在 uart\_asm\_init: 的

str r1, [r0, #EXYNOS4\_GPIO\_A1\_CON\_OFFSET]

后添加

ldr r0, =0x10030000

ldr r1, =0x666666

ldr r2, =CLK\_SRC\_PERIL0\_OFFSET

str r1, [r0, r2]

ldr r1, =0x777777

ldr r2, =CLK\_DIV\_PERIL0\_OFFSET

str r1, [r0, r2]

注释掉 trustzone 初始化

注释掉

bl uart\_asm\_init

下的

bl tzpc\_init

重新编译 u-boot 后即可看到串口输出信息。

添加网卡初始化部分代码：

在 board/samsung/fs4412/fs4412.c 新增网卡初始化部分代码

emmc 初始化：

支持movi指令--cp 给的movi.c arch/arm/cpu/armv7/exynos/ 修改makefile支持编译movi

board/samsung/fs4412/fs4412.c 中新增 emmc 部分的初始化

### 内核编译制作：

•解压内核

•修改内核顶层目录下的Makefile

ARCH ?= $(SUBARCH)

CROSS\_COMPILE ?= $(CONFIG\_CROSS\_COMPILE:"%"=%)

为：

ARCH ?= arm

CROSS\_COMPILE ?= arm-none-linux-gnueabi-

• 导入 默认 配置

$ make exynos\_defconfig 只是配置不是编译

• 编译内核make uImage

arch/arm/boot目录下生成一个uImage文件（如果编译过程中提示缺少mkimage工具，uboot源码中的tools/mkimage拷贝

到ubuntu的/usr/bin目录下）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

弄设备树

2. 修改drivers/char/Kconfig drive目录中的每个文件夹都有Kconfig文件，写的代码最终会显示到对应的menuconfig中

在menu "Character devices"下面

添加如下内容：

config FS4412\_LED 头，到是也不能省略

tristate "FS4412LED Device Support" 有三个选项（bool是两个） + 标题

depends on ARCH\_EXYNOS4

help //选项说明，没有也可以

support leddevice on FS4412develop board

$ vim Makefile

修改：

ARCH ?= $(SUBARCH)

CROSS\_COMPILE ?= $(CONFIG\_CROSS\_COMPILE:"%"=%)

为：

ARCH ?= arm

CROSS\_COMPILE ?= arm-none-linux-gnueabi-

make exynos\_defconfig + make menuconfig

make uImage

通过上述操作我们能够在 arch/arm/boot 目录下生成一个 uImage 文件，这就是经过

压缩的内核镜像。

如果编译过程中提示缺少 mkimage 工具，需将第二天编译的 uboot 源码中的

tools/mkimage 拷贝到 ubuntu 的/usr/bin 目录下

### 修改设备树文件：

cp arch/arm/boot/dts/exynos4412-origen.dts arch/arm/boot/dts/exynos4412-fs4412.dts

添加新文件后需修改 Makefile 才能编译

$ vim arch/arm/boot/dts/Makefile

在

exynos4412-origen.dtb \

下添加如下内容 exynos4412-fs4412.dtb \ 在boot所在目录直接编译设备树文件即可

### USB卡/SD卡/网卡的移植：

1.需要修改设备树文件 新增网卡节点

2.menuconfig中配置-支持网卡

### ramdisk制作

制作ramdisk文件：

1.dd if=/dev/zero of=ramdisk bs=1k count=8192 ( ramdisk为8M) 制作8M的镜像

2.mkfs.ext2 -F ramdisk 格式化

3.sudo mkdir /mnt/initrd ; sudo mount -t ext2 ramdisk /mnt/initrd 挂载镜像到 /mnt/initrd

4.将我们的文件系统复制到 ramdisk 中 ，将测试好的文件系统里的内容全部拷贝到 /mnt/initrd 目录下面

cp /source/rootfs/\* /mnt/initrd –a

5. 卸载ramdisk umount /mnt/initrd

6.压缩 ramdisk 为 ramdisk.gz gzip --best -c ramdisk > ramdisk.gz

7.格式化为 uboot 识别的格式，完成后可以直接使用

mkimage -n "ramdisk" -A arm -O linux -T ramdisk -C gzip -d ramdisk.gz ramdisk.img

8.需要配置内核支持 RAMDISK 作为启动文件系统

make menuconfig

File systems --->

<\*> Second extended fs support

Device Drivers

SCSI device support --->

<\*> SCSI disk support

Block devices --->

<\*>RAM block device support

(16)Default number of RAM disks

(8192) Default RAM disk size (kbytes) (修改为8M )

General setup --->

[\*] Initial RAM filesystem and RAM disk (initramfs/initrd) support

### 文件系统制作

进入Busybox解压后的源码目录

配置->编译->导入库（其中就有arm-gcc）->

自制的文件系统中的etc下添加文件inittab

在etc下添加文件fstab（proc下、 sys下的fs 这是2个文件系统 ）

用文件系统工具生成ramdisk文件系统映象文件

组织管理计算机存储设备上的大量文件，并提供用户交互接口

分为3类：

磁盘文件系统

磁盘-cdrom-dvd-u盘-磁盘阵列 常见格式：ext（扩展文件系统） vfat iso9600(cd-rom) ufs（unix fs） fat ntfs

网络文件系统

在服务器端仍是本地的磁盘文件系统，客户端通过网络远程访问主机数据 nfs服务 samba

专有/虚拟文件系统：

TempFs(临时文件系统) --不住留在磁盘

ext是linux 最常用的文件系统格式

fat ntfs 是win上最常用的文件系统格式

SCSI 磁盘上的设备名称：/dev/sda ---高性能 稳定

IDE 硬盘的设备名称 /dev/hda

hd表示IDE磁盘 sd表示SCSI与SATA磁盘

将内存中的数据写入磁盘或从磁盘中读出，称为内存交换(swapping)

流程

1.下载busybox，进入目录

2.make menuconfig

Busybox Settings --->

Build Options --->

[\*] Build BusyBox as a static binary (no shared libs)

[ ] Force NOMMU build

[ ] Build with Large File Support (for accessing files > 2 GB)

(arm-cortex\_a8-linux-gnueabi-) Cross Compiler prefix

() Additional CFLAGS

3.make 编译

4.make install 默认安装位置为：源码目录下的\_install目录

5.在 \_install 目录中创建别的目录 dev etc mnt proc var tmp sys root

6.拷贝工具链中的目录至 \_install 目录

cp /home/linux/toolchain/gcc-4.6.4/arm-arm1176jzfssf-linux-gnueabi/lib/ . -a

7.为了确保lib目录不超过8M

rm lib/\*.a ；arm-none-linux-gnueabi-strip lib/\* ；du -mh lib/

8.在etc 下添加启动文件 inittab

#this is run first except when booting in single-user mode.

::sysinit:/etc/init.d/rcS

# /bin/sh invocations on selected ttys

# start an "askfirst" shell on the console (whatever that may be)

::askfirst:-/bin/sh

# stuff to do when restarting the init process

::restart:/sbin/init

# stuff to do before rebooting

::ctrlaltdel:/sbin/reboot

9.添加文件 fstab

#device mount-point type options dump fsck order

proc /proc proc defaults 0 0

tmpfs /tmp tmpfs defaults 0 0

sysfs /sys sysfs defaults 0 0

tmpfs /dev tmpfs defaults 0 0

这里我们挂载的文件系统有三个 proc、sysfs 和 tmpfs。在内核中 proc 和 sysfs 默认都支持，

而 tmpfs 是没有支持的，我们需要添加 tmpfs 的支持--

修改内核的menuconfig

make menuconfig

File systems --->

Pseudo filesystems --->

[\*] Virtual memory file system support (former shm fs)

[\*] Tmpfs POSIX Access Control Lists

10.在 etc 下创建 init.d 目录，并在 init.d 下创建 rcS 文件，rcS 文件内容为：

#!/bin/sh

# This is the first script called by init process

/bin/mount -a

echo /sbin/mdev > /proc/sys/kernel/hotplug

/sbin/mdev -s

为 rcS 添加可执行权限： $ chmod +x init.d/rcS

11.在 etc 下添加 profile 文件，文件内容为：

#!/bin/sh

export HOSTNAME=farsight

export USER=root

export HOME=root

export PS1="[$USER@$HOSTNAME \W]\# "

PATH=/bin:/sbin:/usr/bin:/usr/sbin

LD\_LIBRARY\_PATH=/lib:/usr/lib:$LD\_LIBRARY\_PATH

export PATH LD\_LIBRARY\_PATH

12.新制作的文件系统尺寸若超出 8M，删除不需要的库文件

### 其他相关知识

交叉编译工具集：arm-linux-

size a.out：显示可执行文件的数据段、代码段、bss段大小 注意：没有堆栈，堆栈在内存中的

nm xxx：符号标签 显示代码与地址的对应关系表

nm -D a.out:显示符号表

strip xxx：剔除xxx中 代码与地址的对应关系表，剔除了nm 命令就查不到了

strings xxx：查看此二进制文件中 可打印 的字符串 就是printf对应的打印信息

objdump -d xxx：反汇编 显示目标文件的汇编

readelf -s a.out 显示符号表，上层的printf在底层被转换成什么函数执行的

readelf -d 查看当前二进制所需的库，ldd也可以看

readelf -h xx.a/xx.so/a.out 查看此二进制的运行架构

file a.out 也可以查看二进制架构

安全编译选项：

-D\_FORTIFY\_SOURCE:缓冲区保护，防止缓冲区溢出，对于很多函数，编译时都会变成xx\_chk。可通过objdump -d看到

调试：

选择将linux的某个文件夹挂载到demo。这样就会同步demo与linux中的数据，可以通过操作linux来操作demo

tftp客户端：tftp 20008000 a.txt //先将a.txt数据搬移到内存空间

md 20008000 显示这里的数据存的什么 u-boot 中才有这个命令

md.b 30008000 40 看内存中的数据 只查看40字节

memory display

tftp服务端：32位：sudo tftpd-hpa tftpd表示是守护进程 netstst -ua 查看目前的网络服务

nand：格式--nand +earse/write/read +内存地址源/目标 +nandflash内部地址 +搬移大小

写之前要先擦除：

nand earse 0x6000000 2048//擦除6m地址空间2k字节大小空间（其实是扇区擦除的）

nand write 20008000 6000000 2048//将20008000内存中2k的数据搬到6m nand所在位置。这样断电也不会消失

nand read 20008000 6000000 2048//将nand中的6m位置的数据搬到内存中

emmc的分区： config--库--ramdisk--dtb--uImage--bootloader

内核启动：

1，bootagrs:root=rfs在哪里 init=启动的第一个init在哪里 console=使用哪个设备做控制台

2，fs烧写：rfs用户与内核之间的中介

1>>nfs普遍用的 TCP/IP

先安装nfs-kernel-server 自己做/copy

配置/etc/exports 设置一个自己想挂载的任意目录（这个目录这会还是空的）

/test \*（rw，sync，no\_subtree\_check）

sudo etc/init.d/nfs-kernel-server restart

$ sudo mount -t nfs localhost:/source/rootfs /mnt

macos的mount： sudo mount -o resvport 192.168.1.104:/media /mnt

有时候挂载的时候会卡死，需要

mount -t nfs -o nolock,vers=3 不锁定

/etc/rc.d/init.d/xxx 各种服务的启动脚本都在这里放着

/etc/rc.local 开机后自动执行的脚本，可以自己写个啥进去，每次开机自动就执行了(例如mount)

2>>ramdisk掉电消失

root=/dev/ram initrd=02100000,8M init=/linuxrc console=ttySac..

完整代码：

tftp 20008000 uImagine

tftp 21000000 initrd.img.gz rfs压缩包

setenv bootagrs root=/dev/ram initrd=2100000,8M init=/linuxrc console=ttySac..

bootm 20008000 这样就可以启动内核了 不能用go，go命令是直接执行的，内核的头不能执行

## gdb

```c

l -n:从第n行开始查看代码 l func：查看具体函数

l:默认显示当前运行位置上下5行的代码

bt：查看调用栈与层级关系

c : 继续运行程序，直到遇到下一个断点，若没有断点，程序就会运行完。

set var 变量名=‘xxx’ 或者数字 set var i=10

./a.out arg1 arg2 程序如果运行时这样传参

set args arg1 arg2 用gdb调试的时候就这样传 run 之前 对应./a.out arg1 arg2

show args 查看传入的参数

gdb里面的数字代表的是：行数 + 代码内容

info b:查看断点信息

b 15 if i==15 按条件设置断点，不是一到15行就停止了，还得i=15，才会断住。

b filename:15/function 跳到文件名为filename的文件的15行，或者函数名字处

display num1:每次next都会主动将num1变量的值打印出来，看他是否变化

undisplay 变量i的编号。编号的获取：i d

finish：从函数中跳出来，回到上一层函数。并不是一下就跳回到main。跳到上一层继续调试

ptype a：打印变量a的类型

del 4：删除编号为4的断点

watch:被设置观察变量发生改变时才打印显示

i watch：显示观察变量

enable breakpoints 启用断点

disable breakpoints 禁用断点

---进程：

set follow-fork-mode parent 默认就是父进程，这个不用set

set follow-fork-mode child

设置调试模式：set detach-on-fork [on/off]默认是on

on：调试一个进程另一个进程继续运行完

off:调试一个进程另一个进程被gdb挂起 阻塞

info inferiors 查看当前调试的进程信息

inferior 进程id ：切换当前调试的进程 1/2

---线程：

info threads 查看线程

thread 线程id:切换线程

set scheduler-locking on只运行当前进程

set scheduler-locking off 运行全部的线程

thread apply 线程id cmd 指定某线程执行某gdb命令，id是1、2、3不是系统的线程id

thread apply all cmd 指定全部的线程执行某gdb命令

---gcc

-I ：如果不指定，默认从/usr/include 中寻找.h

-L：如果不指定，默认从/usr/lib 中寻找.so

-O：优化选项，不能与-g 一同使用

-Wall：将warning视为error，有warning就退出，不会继续运行

time ./a.out 计算a.out的运行时间

## git

以往对于共享资源，1个人操作的时候其他人不能打开此文件。

git config -global user.name 'yonghuming'

git config -global user.email 'youxiang'

git commit -a -m "//adas" -a参数要求hello.c已经在暂存区了（之前必须已经git add过了）更新暂存区的hello.c

这个操作直接将工作区的代码添加到代码库，不再需要先加到暂存区

比较版本的差异：git log 查看提交日志

git log --oneline 少量信息显示（\*\*用的更多）

git log + filename 显示此文件的log

git log -p filename 查看此文件所有历史提交所作的修改

git show + commitid 显示此commitid的更改文件

git reflog：显示信息中多了Head版本的HEAD{X},HEAD{Y}

git diff 315f9fe 666eb6c比较版本1与版本2

git diff 315f9fe 比较当前版本与版本315f9fe

git diff HEAD~ HEAD~~ 比较 上2个版本差异

HEAD表示当前版本

版本前进回退：git reset --hard 9a9ebe0:就回到了过去的版本，工作区文件中某些内容没了

重置 --hard：本地库移动HEAD指针，并且重置暂存区，并且重置工作区

--mixed：本地库移动HEAD指针，并且重置暂存区

--soft：仅仅在本地库移动HEAD指针

如果rm \*所有文件丢了怎么办？

git checkout HEAD . “·”表示返回最新版本所有 “.”类似于通配符“\*”

git checkout HEAD~ . / hello.c 捡1个 “·”表示返回上个版本所有

git checkout 666eb6c hello.c弄回来最旧的版本的hello.c

tag：git tag v0.9 315f9fe相当于给315f9fe重命名

版本删除：rm readme.txt + git commit -a 从仓库删除readme.txt 不然git status显示的删除只是工作区的删除，

不是本地仓库的删除

git rm hello.c 从仓库删除hello.c

打包：git archive HEAD~ -o pro.tar.gz

添加前缀：git archive HEAD~ --prefix=work\_ -o pro.tar.gz

$ : tar -ztf pro.tar.gz 只查看不是真正的解压

tar --touch -xvf xx.tar忽视时间解压

tar -cf --exclude=.git `ls -a` 打包除了.git外其他文件

work\_hello.c

work\_readme.txt

git branch dev：创建分支 dev-------git branch feature\_A创建featur\_A分支

一般主分支用来打补丁，改bug的。

合并分支：合并之前要切换回对应要合并的主分支（不一定是master）。

git checkout master

git merge dev（没有冲突会自动合并）

如果主分支与别的分支都修改了hello.c

merge时会：

git merge hot\_fix：将hot\_fix分支合并过来

git merge feature\_A：将feature\_A合并到主分支

Auto-merging hello.c

CONFLICT (content): Merge conflict in hello.c

Automatic merge failed; fix conflicts and then commit the result.

这次重新修改hello.c再合并：git commit -a这样就合并完了。

传统开发模型：

需求分析，概要分析，详细设计，编码，单元测试，集成测试，验收测试（1环套1环，时间出问题周期太长）

敏捷开发流程： （迭代）

做一段先上线，有问题改问题，一个一个阶段来。每个阶段不会出大问题。根据客户需求改动

建立远程连接git remote add origin https://gitee.com/jl\_git/intelligent\_warehouse.git

删除远程git remote rm origin https://gitee.com/jl\_git/intelligent\_warehouse.git 相当于给网站取别名

push到这里git push

git push -u origin master -u：以后默认往这里推送（git push就好了）

工作区->缓存区->本地代码库->远程代码库

\* [new branch] master -> master表示将本地的master推送到了远程代码库master

git remote add origin https://github.com/jialei-github/jialei.git

git push -u origin master

git fetch orgin master + git merge orgin/master

git fetch + git merge显示当前冲突 fetch只是把远程下载到本地，并不会更改工作区内容，merge才会更改本地工作区内容

git log -p master.. orgin/master

git merge master.. orgin/master

\*\*如果有冲突push不上去的时候要先更新本地代码库，然后merge解决冲突（重新编辑一下git add+git commit）才能push

如果不是基于远程库最新版所做的修改，直接push不可以。必须先拉取解决冲突

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

git reflog 取代git log --oneline

git log 多屏显示时space 向下翻页 b向上翻页

git push origin master/fenzhi1/fenzhi2/xx

如果想直接往别人的url提交代码，需要加入别人的团队，让别人set collaborations + 自己的账号

git fetch origin master+git merge origin/master

git checkout origin/master 切换到远程分支

合并两次提交：

（1）：若远程都已经提交完事了，现在想合并其中的某几个commit，可以通过

git rebase -i commitid 更改除了第一个pick之外的其他pick为squash ：wq保存

再更改合并之后的commit描述 ：wq

git push -f 得强推

（2）：若刚刚提交了一个commit，现在更改了文件再提，想跟刚才的commit合并为同一个commit

git add 1.c 后 直接git commit --amend 直接写合并之后的commit描述就好了

最后也得git push -f

git add --all后想从缓冲区删除部分文件

git rm --cached GJC4824\_01\_145\_V101\_V100.\* 删除新add的所有GJC4824\_01\_145\_V101\_V100开头的文件

git rm --cached dir/ -r 递归删除add的dir目录

## vim

- vim 1.c +18直接打开就是第18行

vim中左括号跳右括号：%

vim 查看二进制文件：：%!xxd + enter

^ + e:光标位置不变，文件向上移动

ctrl+x + ctrl+n 自动补齐单词

home/end 跳到行首行尾

:saveas ./path/filebak 另存为

:e ./filename 打开新的文件 此时vim会同时打开其他文件

x: cmd 删除当前光标所在字符

O：行上插入

cw： cmd 替换当前单词

0： cmd 跳到行开头

^: cmd 跳到行开头第一个单词位置

$: 跳行尾

P：赋值当前粘贴版内容到当前行之前

:bn bp 切换当前打开的vim文件

.： cmd 重复上次的cmd，底行模式输入的不算cmd

w: cmd 跳下个单词开头

e: cmd 跳下个单词结尾

%： cmd 匹配括号 左跳右 右跳左

\* + enter 高亮显示当前光标处的单词 n寻找下一个 N寻找上一个

fx： cmd 跳到当前行中x字符所在位置

tx: cmd 跳荡当前行中的x字符所在位置之前

批量屏蔽c代码+//

批量行前插入：ctrl+v批量选中+I 插入字符后 +esc 完成批量的行前插入

批量字符后插入：ctrl+v批量选中+A 插入字符后 +esc 完成批量的行后插入

批量行末尾插入字符：ctrl+v选中行+shift+$选中整行尾+A行后插入+字符+esc

同一个文件分屏操作：底行模式 两个屏幕操作同一个文件

:split 上下分屏

:vsplite 左右分屏 切换ctrl+w +方向键

两个屏幕操作不同文件怎么做

:split ./otherfile 打开其他文件

所有的vim配置都在.vimrc里面

## makefile

$@:目标文件

$<:第一个依赖文件

$^:所有依赖文件

%.c：所有.c文件

.PHONY:clean假想目标，make clean的时候不会真的生成clean文件

普通的目标：make目标的时候会生成目标文件的。

变量：

A := abc 及时变量，现在A的值就确定下来

B = 123 用到此变量时值才会缺定

all:

@echo $A:加@则命令行本身不显示，只显示打印结果

@echo $B

A := $(C) 及时变量，现在A的值就确定下来，是空，因为此时C都还没赋值呢.相当于A=(C!=null)?C:"";

B = $(C) 用到此变量时值才会缺定，则B的值是abc

C = abc

C += 123 ：此时C的值为--abc 123

A ?= 延时变量，只有第一次定义才会起作用，后面再使用?=就不再起作用了，忽略此句

---

A = a b c

B = $(foreach f,$(A),$(f).o)

则此时的B=a.o b.o c.o

C = a b c d/

D = $(filter %/,$(C)) 从C中匹配目录字符，则D = d/

E = $(filter-out %/,$(C)) 从C中剔除d/， 则E = a b c

files = $(wildcard \*.c):匹配此makefile所在的文件夹中的所有存在的.c文件（真实文件）。

filesx = $(patsubst %.c,%.d,$(files))：将files变量的结果中所以有的.c变成.d赋值给filesx，

obj = $(patsubst ./%.c,./%.o, $(src)) 有没有 ./ 效果一样

对于不符合.c结尾的字符串，照搬下来不做修改

实例：---------------------------------------------

objs = a.o b.o c.o

dep\_files := $(patsubst %, .%.d,$(objs))----dep\_files=.a.o.d .b.o.d .c.o.d

dep\_files := $(wildcard $(dep\_files))

CFLAGS = -Werror -I. （-Iinclude表示在include目录查找） 表示头文件在当前目录查找

test:$(objs)

gcc -o test $^ 利用所有的.o生成目标文件

ifneq ($(dep\_files),) 如果此变量不为空

include $(dep\_files)

endif

%.o:%.c

gcc $(CFLAGS) -c -o $@ $< -MD -MF .$@.d 防止因为.h文件修改而不编译

distclean:

rm $(dep\_files)

顶层目录的CFLAGS：编译所有.c文件都会用到的编译选项

顶层目录的LDFLAG：链接最后程序时用到的链接选项

子目录中的EXTRA\_CFLAGS:只给当前目录下的文件（不包含子目录）设置的额外的编译选项，可以不设置

子目录中的CFLAGS\_xxx.o：只给当前目录中具体的某个xxx.c文件设置的编译选项，编译其他.c时不加此选项

obj-y += xxx.o 表示要编译当前目录中的xxx.c文件

obj-y += yyy/ 表示要编译当前目录下的yyy子目录

makefile的生成方式：

1.利用cmake命令与 Cmakelists.txt（txt文件比较简单）

2.automake工具，即-./configure 生成makefile

%.o:%.c:

$(cc) -c %.c -o %.o 自动将所有.c生成对应的.o

obj = 1.o 2.o 3.o ...太多了怎么破？函数参数，参数直接写到函数名后面即可之间用‘，’隔开

1.src = $(wildcard ./\*c)获取本目录下的所有的.c赋值给src

2.obj = $(patsubst ./%.c,./%.o, $(src)) 将src中的任意1个.c转变为.o赋值给obj

依赖生成目标

无依赖执行直接执行伪目标

直接make生成的是终极目标 target

其实make是个软件，makefile只是个配置文件，make 命令没有makefile也能执行的

例如在有xx.c的目录中直接执行make xx，就会自动根据xx的名字找对应的.c然后生成对应的xx可执行文件

CC xx.c -o xx 隐士规则

目标：依赖

sh xx.sh 这里其实可以执当前目录的的shell脚本（用sh去运行此脚本）

默认执行make命令其实是make的makefile中的第一个目标

顶层make -C xxx -DMACRON 定义的宏会传到底层的任意makefile中去。

顶层makefile中定义的变量若底层不做修改，保持原值，底层可以增加例如：

CFLAGS+=-s -ftrapv -g等等。

#program comment 编译期间可以打印调试信息。

驱动模块的makefile：

#CFLAGS+=$(DEBFLAGS)----驱动里编译调试信息

ifeq ($(KERNELRELEASE), )//注意有个“ ，”这句话是看KERNELRELEASE是否为空

KERNELDIR ?= /lib/modules/$(shell uname -r)/build //选择制作ubuntu所在的makefile所在的路径

#KERNELDIR ?= /home/farsight/rootfs/lib/modules/3.14.0/build //选择板子上的linux内核makefile路径

PWD := $(shell pwd) //当前驱动.c的路径

modules:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

else

obj-m := hello.o

endif

clean:

rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c .tmp\_versions Module\* modules\*

## 驱动开发：

#### 模块传参：

在源码中写好可以传参，然后模块安装的时候就可以传参进去，类似函数传参，确定唯一的模块

代码.c中也要有配置才行

int x；

module\_param(x,int,0664); 变量名、类型、权限不能是0666

char y;

module\_param(y,charp,0664) char型指针，传字符串的

sudo insmod ./hello.ko x=1 y=“字符类的这么传” 安装时才传，安装后以后这里面的变量就固定下来了

多个.c的模块怎么写：

更改makefile：在obj-m:=hello.o 下一行+

还只能是objs：hello-objs := hello.o hehe.o

注意不能这么写，两个hello.o产生了递归了。生成的hello.ko卸载不了了

可以写成 hehe1.o hehe2.o

hello-objs 就是要生成的ko的名字 hehe1 hehe2是依赖的中间文件

#### 模块导出：

产生依赖A---->B b中使用了a的函数与变量\*\*\*（删除的时候先卸载b.ko才能卸载a.ko）

1.编写a.c时在变量后面加一行EXPORT\_SYMBOL(变量名)；

函数名后面加一行EXPORT\_SYMBOL(函数名)；

导出a.ko中的变量与函数。

2.编写b.c时先要声明extern int x;

extern int add(int a,int b);

先声明后才能使用。

3.编译a.c----->生成a.ko------>insmod a.ko

方法一：4.将a.c所在目录下的Module.symvers拷贝☞b.c所在目录，然后编译b.c----->生成b.ko---->insmod b.ko

modprobe：

modprobe ： 模块探测

把相关的Module.symvers模块 copy 到/lib/modules/3.123-gerneric 下

把ko拷贝到/lib/modules/3.123-gerneric的update目录下就能直接通过modprobe探测到了

sudo depmod -a 建立模块之间的关系

sudo modprobe hehe 这个随便在哪里执行都可以安装模块2了 解决依赖关系

### 驱动代码:

ioctrl：

long(\*unlocked\_ioctl)

命令生成： \_IO \_IOR \_IOW 这几个命令写法、参数不一样，功能基本差不多

#define \_IOC(dir,type,nr,size)

type----随机数magic

dir-----命令的方向，读，写

nr------命令的序号

size----命令的参数类型（int char）

inode：存储文件的元信息(+各个数据块的地址)--真正访问的时候会根据元信息将这些数据块拼接到一起

创建设备文件：mknod /dev/hello c 249 123 观察inode使用情况

普通文件：touch 1 2 3 观察inode数量

硬盘容量很大只占用20%但是写不了文件？？？因为inode用满了

作驱动的时候open的是设备文件，不是普通文件，操作还是读写

struct file\_oprations hello\_ops={

.release = hello\_close, //并不会直接关闭，如果2个程序同时打开同一个dev，其中1个并不会关闭另一个的打开。 直接关闭的话2个程序都不能用了

}

make modules 编译板子中需要用到的modules。

//如果直接make modules\_install会将编译的模块放到当前所在的系统中，这是不对的。

我们需要将编译的模块放到板子的/root/lib/modules == ubuntu/~/rootfs

make modules\_install INSTALL\_MOD\_PATH= ~/rootfs 将模块安装到rootfs中

执行完 ~/rootfs/lib/ 下多了1个/lib/modules文件夹

直接写模块入口函数：int init\_module(void) 这么写就不需要写module\_init了

出口函数：void cleanup\_module(void) 这么写就不需要写module\_exit了

module\_init与module\_exit 相当于给入口函数出口函数重命名的

安装：insmod +demo.ko---------对应module\_init(demo\_init);入口函数【安装】

卸载：rmmod+demo---------对应module\_exit(demo\_exit);出口函数【卸载】

modinfo ./hello.ko查看模块的相关信息

最后一行的versionmod:就是编译mod时的内核版本。

mod安装不上时可能是当前内核是A，而编译此mod的内核是B，所以不能insmod。

1.写驱动程序

2.修改Kconfig 添加1个菜单项，这项可以在make menuconfig中进行配置

3.make menuconfig 配置刚才的那一菜单项，要不要编译

4.查看.config 看一下是否配置了

5.在drive/char 下修改Makefile（a.c所在目录的）

设备文件：

设备号： 设备号是一个纽带， 链接驱动程序 和 用户空间的设备文件，

写设备驱动程序， 首先申请设备号：

cat /proc/devices 查看可用的设备号（主设备）

设备：虚拟设备/dev/xxx 可以mknod创建虚拟的设备

静态申请设备号：自己指定要注册的主设备号，次设备号

int register\_chrdev\_region(from, count, char \*name)

注册的第一个设备号（注意这个不是主设备号，是总设备号） 可以传MKDEV(ma,mi)，count=1就行，表示只注册一个设备号

void unregister\_chrdev\_region(dev\_t from, unsigned count)

cat /proc/device 查看当前os中的设备号。

动态申请设备号：若申请多个，通过dev\_t\*变量传出来

int alloc\_chrdev\_region(dev\_t \* dev,从设备号，count，char\*name)；动态根据/proc/device看那个号没用就可以用

注销：注销方式一样的，就申请方式不一样

void unregister\_chrdev\_region(dev\_t from, unsigned count)

在ubuntu里面的驱动makefile中：KERNELDIR ?= /lib/modules/3.14.0/build ubuntu中linux内核的makefile

在demo板中的makefile：KERNELDIR ?= /source/rootfs/lib/modules/3.14.0/build arm中的makefile

$ dmesg 查看内核的日志信息 ，一般查看之前先清一下历史记录 sudo dmesg -c：清除历史日志信息

sudo cat /proc/kmsg enter另一窗口运行主程序，看内核打印信息

内核模块与设备驱动之间的关系：

模块：linux内核组件管理的方式 驱动：都是基于模块进行注册和注销的

驱动的种类：

字符设备：IO传输以字符为单位-----应用层read/write，硬件就紧接发生进行读写操作---eg：鼠标、键盘、触摸屏

块设备：有缓存的 应用层read/write，硬件不会紧接发生---eg：磁盘，闪存

网络设备：eg：网卡【网络设备没有设备文件】

描述所有字符设备有个结构体：cdev【内核要管理就要有统一的机制】就是这个结构体

struct cdev{

struct module \*owner;//THIS\_MODULE

const struct file\_operation \* ops; //里面全是函数指针，操作方法集

dev\_t dev; //设备号

int count; //一般一个驱动对应一个设备，有时候一个驱动也会对应多个设备

struct list\_head list; //链表  
}

MKDEV(major，minor)；根据主次设备号生成设备号

MAJOR（设备号）；提取主设备号 --高12bit 区分哪类驱动（串口）

MINOR（设备号），提取次设备号 --低20bit 区分具体哪个驱动设备（串口1，2，

写设备驱动程序首先要申请设备号，----这是注册设备号呢-------

设备号注册之后才可以制作设备文件（根据设备号） ---对应的命令 mknode

，之后操作设备都依靠的是设备文件

内核内的模块添加：

1.写驱动程序

2.Kconfig 就是配置内核选项的文件-- .config文件中是配置过能看到的文件怎么配置的

3.make menuconfig 根据Kconfig文件配置要编译进内核或者要编译成mod的模块

4.记住要添加内核选项还要修改makefile（修改a.c所在目录的makefile）

如何编写字符设备驱动：5步 操作集刚上来就要写

0.注册设备号【2种方法】

[1]自动分配

[2]指定设备号注册

1.struct cdev \* cdev\_alloc(); 就是创建字符设备。 cdev就是字符设备,申请一段空间存字符设备--申请失败cdev\_del(pdev);

2.void cdev\_init(&cdev,&ops操作集); 初始化cdev结构体ops可以操作cdev

3.int cdev\_add(&cdev，设备号，设备个数)； 将设备号与设备关联起来的字符设备注册到内核，由内核统一管理，某个设备号有哪些operations，就定下来了。

此时用户怎么使用这个驱动呢？ 得通过一个设备文件，可以sudo mknode /dev/haha0 c 250 0来创建字符设备。

自动创建设备文件：对应命令mknode

设备其实不是驱动程序（内核空间的）创建的，只是向sys注册，设备是守护进程创建（守护进程看到这里sys有信息，就根据信息去创建）

（驱动本身不能在用户空间创建dev）

通过创建类设备（设备信息），创建设备device（设备信息），只是向sysfs注册设备信息，sysfs是内核里内存型的fs，主要管理设备、驱动。sysfs在开机时候就会挂载到/sys

pcls = class\_create(THIS\_MODULE,"hello");//类 --->对应/sys/class/classname

for(index = 0;index < count;index++)

{

device\_create(pcls,NULL,MKDEV(ma,mi+index),NULL,"hello%d",index);创建设备文件 >对应/sys/class/classname/devname

}

DEVICE\_CRE\_ERROR:

for(index = 0;index < count;index++)

device\_destroy(pcls,MKDEV(ma,mi+index));//这里就是销毁pcls

CLASS\_CRE\_ERROR:

class\_destroy(pcls);//销毁类

ALLOC\_ERROR://顺序执行，跳到哪里，一直往后执行

cdev\_del(pdev);

REGISTER\_ERROR://只是个标号，不会打断操作

unregister\_chrdev\_region(devno,count);

return -1; \*\*\*&&&&注意这里必须有return -1；不然会出问题，本来出问题就应该return 的，只把资源释放，还是没有退出，相当于函数没有返回值的

}

用户空间：（在内存中占用0-3）

运行用户程序

内核空间：（3-4）

运行：驱动，内核代码

printK:打印优先级，可以设置打印出来那种信息：错误、调试、、、、

驱动的地址映射：

1.直接将硬件地址写到驱动中，将这些硬件地址ioremap为虚拟地址然后实行硬件操作---不便扩展但是很方便简单

2.将硬件资源放到一个struct中去维护，操作的时候还是同样的方法

3.每个硬件对应一个platform\_driver驱动（多个硬件与多个驱动怎么联系呢）

4.引入总线，硬件的修改不会影响驱动，驱动的修改不会影响硬件

5.引入设备树，平台设备中扫描到资源，通过总线与驱动连接起来

platform\_bus\_type：一边是dev列表，一边是dro列表，中间通过总线自动联系起来

#### 分离-设备，驱动

当注册一个设备时，会通过虚拟总线找匹配的平台驱动。（驱使硬件设备动起来read/write/ioctl/open/close/mmap）

平台设备框架把驱动与设备分离起来，要改那块改那块就好。改动起来不影响。

设备是硬件--驱动是软件driver

注册设备就是把设备挂到平台总线 会通过虚拟总线找对应的驱动

注册驱动就把驱动挂到总线上 平台驱动结构体中有个探测函数，寻找设备

虚拟总线成为platform

挂在虚拟总线上的设备称为 platform——device

挂在虚拟总线上的驱动称为 platform——driver

platform\_device\_register（1参数platform\_device）自己写一个设备然后注册进去，注册平台设备--挂在设备到平台总线

platform\_drive\_register（1参数platform\_driver）注册平台设备--挂载驱动到平台总线

struct platform\_device platform\_device定 义设备的信息

{

。name = “asdas”；

。dev = {

。release = 注销时用

}

.num\_resources = 3, 资源个数，资源数组，存的寄存器

.resource = myresource,

}平台设备添加资源--平台驱动获取资源

static struct platform\_driver g\_stPlantFormDriver =

{

.driver = {

.name = "hehe1\_platform",// 该平台驱动支持的设备

.owner = THIS\_MODULE,

},

.probe = HelloDriverProbe,//注册了该平台驱动时， 如果该驱动支持的设备存在，当前的probe函数会被执行

.remove = HelloDriverRemove,//注销该平台驱动时， 当前的remove函数会被执行到

};

探测函数在驱动中：找对应的设备，直接就能使用了

int HelloDriverProbe(struct platform\_device \*pDevice)

{//平台驱动的探测函数, 同时带过来平台设备的结构体pDevice

//字符设备的完整的流程

hello\_init(pDevice);

printk("Driver,HelloDriverProbe\r\n");

return 0;

}

#### 设备树

有了设备树，就不用写平台设备的代码了。&&&&&内核加启动时载的时候就将平台设备注册好了。省去了自己写平台设备代码

将dtb文件放进/boot目录重启即可加载 ，形成platform\_dev结构体，drv中可以直接通过方法获取这些结构体。

linux-3.14/arch/arm/boot/dts 目录中的exynos4412-fs4412.dts就是设备树源代码

更改目录下的exynos4412-fs4412.dts，添加设备节点

在顶层目录linux-3.14/下执行 make dtbs生成exynos4412-fs4412.dtb 放入/var/lib/tftpd-pha/目录中（可以给这个dtb换个名字）

【】用device+drive写驱动时，match通过platform\_driver.name与platform\_device.name匹配

【】用设备树时，驱动的匹配通过，platform\_driver.of\_match\_table 描述驱动支持的设备树里的某一个设备

\*\*\*一个设备节点可以包含整个板子的所有寄存器，不用建立很多节点

dts语法：

常用属性：

#address-cells 表示address要用几个32bit的数值去表示（32-64bit的CPU此值不同）

#size-cells 要用几个32bit的数值去表示大小

label:uart@0x1234{}; node的lable若存在，需要更改此node的属性，可以直接：

address-cells与 size-cells影响子节点中的reg属性

{

#address-cells=<1>

#size-cells=<1>

memory{

reg=<0x8000000 0x2000000> 因为address-cell是1，

则0x8000000就表示地址 因为size-cell 为1，则0x200000就表示大小

}

#address-cells=<2> 64位系统地址就需要2字节表示

#size-cells=<2>

memory{

//对于不同的设备 reg表示不同的意思

reg=<0x8000000 0x2000000 0x1234 0x5678> 因为address-cell是2，

则0x8000000 0x20000000就表示地址 因为size-cell 为2，则0x12345678就表示大小

}

}

led{

compatible="koA","koB","koC" 表示当前设备树支持驱动 A B C

}

//公版dtsi 被板子AB 都include ，但是A中支持uart B不支持 则可以在后面更改对应的status

status="disable" 或 "ok" 或 ""

可以在根节点之外修改此节点属性

1. &lable{----可以新增节点属性，也可以修改之前的节点属性

var='value';

status='enable'/'disable';可以使能或禁掉某个节点

compatible='A','B','C';此设备兼容A,B,C3种驱动，优先级A,B,C

};

'

/{

uart0:uart@fe00100{

compatile="ms1665"

reg=<0xfe00100 0x100>

};

}

//因为上面有label，所以可以直接修改uart@fe00100 这个节点是虚拟的

&uart0{

status="disable"

}

'

2. 若lable不存在，需要更改此node的属性，可以直接：

&[/uart@0x1234]{-----------/uart表示根节点下的uart节点，

var='value';

};

/ {

100ask\_led@0 {

name=val; val有3种取值

1."string" 双引号内随便写

2.<u32 u32 u32> 有几个写几个u32的数据

3.[hex hex hex] 有几个写几个hex的数据

4.组合起来:<0x12345678>,"mystring",[12 34]

compatible = "100as,leddrv";

pin = <GROUP\_PIN(3, 1)>;

};

100ask\_led@1 {

compatible = "100as,leddrv";

pin = <GROUP\_PIN(5, 8)>;

status = "disabled"; 定义为disabled后初始化的时候就不初始化此节点了

};

};

dts文件格式：

/det-v1/; dts的版本

/{

[资源定义]

[子节点]

};

node的格式：label有没有都行，可以方便的引用某个节点--[]中有没有都可以

[label:]node-names[@unit-address]{

[资源定义]

[子节点]

};

dts必须有根节点：根节点必须有如下属性

/{

#address-cells=<1>; 在子节点的reg属性中，用几个u32来描述地址

#size-cells=<1>; 在子节点的reg属性中，用几个u32来描述大小

compatible="samsung,smdk2440"; 此板子兼容哪些平台，值是字符串列表

model="smdk2440"; 如果两款板子compatible配置一致则通过model区分，表示这个板子时什么

cpus{ cpu节点

#address-cells=<1>; 在子节点的reg属性中，用几个u32来描述地址

#size-cells=<1>; 在子节点的reg属性中，用几个u32来描述大小

cpu0:cpu@0{

......

}

};

memory{

reg=<0x8000000 0x2000000> 前者表示内存起始地址，后者表示内存大小

};

chosen{//通过设备树给内核传入参数，例如bootargs,这个节点是虚拟的

bootagrs="initrd root=/dev/nfs nfsroot=192.168.1.101:/source/rootfs rw console=ttySAC2,115200 init=/linuxrc ip=192.168.1.250"

};

};

dtsi:被别的dts包含的头文件，里面放的是公有的内容

/sys/firmware/devicetree/base 里面就可以查看设备树中的各个节点，boot的时候用哪个dtb启动，

则此目录中的结构就是哪个dtb文件的内容

--------------------------------------------------

怎么加载dtb中的数据呢？

将dtb文件放到/boot目录中，将之前开机用的dtb文件重命名为bak，重启单板即可

查看启动板子的设备树文件：/sys/firmware/devicetree/base 里面放的就是dtb文件中的每个节点以及节点中的每个属性

/sys/devices/platform 内核根据设备树解析出来的部分device\_node结构体转化而来的platform\_dev结构体

内核对设备树文件的处理过程：

DTS--(编译)-->DTB--(uboot将dtb文件按传给内核，内核解析)-->device\_node--(某部分会被转换成platform\_dev)-->platform\_device

我们可以每次从根节点遍历所有的子孙节点。

具体哪些device\_node会被转换成platform\_dev结构体呢？

A. 根节点下含有compatile属性的子节点

B. 含有特定compatile属性的节点的子节点

如果一个节点的compatile属性，它的值是这4者之一："simple-bus","simple-mfd","isa","arm,amba-bus",

那么它的子结点(需含compatile属性)也可以转换为platform\_device。

C. 总线I2C、SPI节点下的子节点：不转换为platform\_device

某个总线下到子节点，应该交给对应的总线驱动程序来处理, 它们不应该被转换为platform\_device。

比如以下的节点中：

/mytest会被转换为platform\_device, 因为它兼容"simple-bus";

它的子节点/mytest/mytest@0 也会被转换为platform\_device

/i2c节点一般表示i2c控制器, 它会被转换为platform\_device, 在内核中有对应的platform\_driver;

/i2c/at24c02节点不会被转换为platform\_device, 它被如何处理完全由父节点的platform\_driver决定, 一般是被创建为一个i2c\_client。

类似的也有/spi节点, 它一般也是用来表示SPI控制器, 它会被转换为platform\_device, 在内核中有对应的platform\_driver;

/spi/flash@0节点不会被转换为platform\_device, 它被如何处理完全由父节点的platform\_driver决定, 一般是被创建为一个spi\_device。

exp:

/ {

mytest { --------------ok

compatile = "mytest", "simple-bus";

mytest@0 { --------------ok

compatile = "mytest\_0";

};

};

i2c { --------------ok

compatile = "samsung,i2c";

at24c02 { --------------not ok

compatile = "at24c02";

};

};

spi { --------------ok

compatile = "samsung,spi";

flash@0 { --------------not ok

compatible = "winbond,w25q32dw";

spi-max-frequency = <25000000>;

reg = <0>;

};

};

};

怎么跟drv匹配呢？

参考图片platdev+platdrv.png

int HelloDriverProbe(struct platform\_device \*pDevice)

{

hello\_init(pDevice);平台驱动的探测函数里面会把平台设备的结构体带过来struct platform\_device \*pdevic

映射ioremap的时候ledcon = ioremap(pDevice->resource[0].start, 4);

完成寄存器的映射 pDevice->num\_resources是带过来的资源总数

iounmap(ledcon);iounmap接触映射

dts 被编译成dtb 后内核利用此dtb启动后，会将dtb中的节点转化成device\_node节点，部分device\_node

又会转化成platform\_device节点 供驱动绑定识别

那些会被转化呢？ 查看docs文档设备树章节

根节点 含有compatible属性的子节点

#### 中断：

以下是三种切换的区别:

a. 函数调用：

在函数A里调用函数B，实际就是中断函数A的执行。

那么需要把函数A调用B之前瞬间的CPU寄存器的值，保存到栈里；

再去执行函数B；

函数B返回之后，就从栈中恢复函数A对应的CPU寄存器值，继续执行。

b. 中断处理

进程A正在执行，这时候发生了中断。

CPU强制跳到中断异常向量地址去执行，

这时就需要保存进程A被中断瞬间的CPU寄存器值，

可以保存在进程A的内核态栈，也可以保存在进程A的内核结构体中。

中断处理完毕，要继续运行进程A之前，恢复这些值。

c. 进程切换

在所谓的多任务操作系统中，我们以为多个程序是同时运行的。

如果我们能感知微秒、纳秒级的事件，可以发现操作系统时让这些程序依次执行一小段时间，进程A的时间用完了，就切换到进程B。

切换过程是发生在内核态里的，跟中断的处理类似。

进程A的被切换瞬间的CPU寄存器值保存在某个地方；

恢复进程B之前保存的CPU寄存器值，这样就可以运行进程B了。

中断上下文切换：

将用户态的15个reg 与 cpsr存起来---然后进入内核态 pc指向内核态代码

恢复现场：从栈中把reg值全吐出来（一个进程只有一个栈，切换的时候数据全存到栈里面了）

进程上下文切换，进程间切换，每个进程都有自己的栈数据网哪里存呀？

内核中的定时器：

jiffies：自系统开机就开始计数+1+1+1+1，可以根据os自启动以来的jiffies数计算开机以来运行时间jiffies/HZ（单位秒）

HZ：决定每s产生多少次定时器中断

\*\*\*jiffies每隔 1/HZ 时间 +1 即1/(1s/100) 是个时间段数（这个HZ不是Hz单位）

HZ = 1s/100

内核的HZ决定jiffies计数，一般1s增加250次

定时器作用：延后执行某个动作，定时查询reg状态。

HZ通过config中的CONFIG\_HZ去配置。

驱动中断：在dtb中指明那个脚要中断，中断触发方式

fs4412-key {

compatible = "fs4412,key";

interrupt-parent = <&gpx1>; gpx1的2脚，第二个2是触发方式：下降沿

interrupts = <2 2>;

};

plat\_drive.c只改of\_table\_id.

hello.c中：

在初始化函数外部实现中断处理函数：

int keypara = 100;

irqreturn\_t keyhandler(int no, void \*para（这里可以给它传参）)

{

printk("no = %d, para = %d\n",no, \*(int\*)para);

return IRQ\_HANDLED;

}

1.pkres = platform\_get\_resource(pDevice, IORESOURCE\_IRQ, 0);（获取的是中断号）

printk("pkres->start = %d, pkres->end = %d\n",pkres->start, pkres->end);

2·初始化注册中断

ret = request\_irq(pDevice->resource[0].start, keyhandler, pDevice->resource[0].flags, "key-int", (void\*)&keypara);

中断函数默认就会禁止掉其他中断，不允许中断嵌套。例如你在中断函数中{HAL\_delay(5)},这个delay本身就是中断，所以他会卡死到这里。

3.中断的使用很简单，只需要在ko的init中注册中断函数就可以；也可以放到open函数中注册，这样的话必须调用open函数，

中断才算真正被注册，具体看业务需求。中断处理的释放函数可以放到cleanup中，也可以放到close中去释放中断，看业务需求。

request\_irq(中断号，真正的处理函数，xx,flag)；flag为中断触发方式，包括：

高/低电平触发；上升/下降沿触发；

一旦中断驱动被调用注册，则随时可以触发。

#### 多路复用

应用层的select，epoll，select--->驱动层的helo\_poll（）

select检测fd对应的io通道上是否有数据传过来，若有数据把相应的读集合位置1.然后处理此事件

select--->需要定义file\_operations里面的poll（file\*结构体pf，轮询表ptable）

{

poll\_wait（pf，等待队列，ptable）；

}

异步通知：设置支持异步（信号驱动的异步io）

底层检测的 异步链（队列）有没有 这种信号发出来 然后调用相应得中断处理函数

等待队列：队列+进程调度机制 实现 异步事件通知机制+信号量

### 驱动移植

led驱动 移植进内核

1.拷贝fs4412\_led\_drv.c 拷贝到 drivers/char目录

2.修改drivers/char/Kconfig

在 menu "Character devices"下面

添加如下内容：

config FS4412\_LED

tristate "FS4412 LED Device Support"

depends on ARCH\_EXYNOS4

help

support led device on FS4412 develop board

3.修改 drivers/char/Makefile

在文件最后添加如下代码

obj-$(CONFIG\_FS4412\_LED) += fs4412\_led\_drv.o

4.make menuconfig时在Device Drivers->Character devices->FS4412 LED Device Support 前面按y 编译进内核

编译内核后 ，重启板子 用写好的测试程序去测试 看灯是否会开关

编译 LED 驱动为模块

make menuconfig时在Device Drivers->Character devices->FS4412 LED Device Support 前面按m编译 为模块

make uImage 编译无led的内核

make modules 编译config设为M的模块

cp drivers/char/fs4412\_led\_drv.ko 到单板，测试驱动能否适配

### 阻塞底层实现-并发竟态：

阻塞：想要对设备操作时不能操作，则阻塞，进程睡眠。

满足可操作条件后，内核唤醒，进程继续执行。

非阻塞：不能操作时，不会睡眠立刻返回结果。

read write阻塞的底层靠

先在hello\_init中初始化等待队列。init\_waitqueue\_head(&q);

wait\_event\_interuptable（q，con）q等待队列头 con唤醒条件（1结束等待0继续等待）

把请求读数据的进程放到休眠等待队列中，睡会等有人写入可读资源

当进程正常运行时，进程放在运行队列等待被运行

当进程休眠时，进程在等待队列等待被唤醒

wake\_up\_interuptable（&q）唤醒进程

在hello\_read让程序加入等待队列头，在hello\_write中让程序唤醒读进程

1个cpu就可以实现并发无法实现并行 多个cpu可以实现并行

通过1个全局变量的值解决竟态可以吗？？（看着没啥毛病，但是由于时间片的分配会影响的）程序的执行可能会被打断

hello\_open() -----这个函数其实不实现也可以，平时打开文件打开的是设备文件，open并不是只通过这个函数打开的文件

{ hello\_opne的作用本质是实现设备的初始化可以在此函数中添加 pwm\_init();

if(flag==0) 这才是这个open函数的作用，如果只是函数中prink一下的话，这个函数没啥用，可以不定义的

{ hello\_open更像一个初始化函数

现在没人打开，你可以用

flag=1；把它置1，防止另一个进程也调用---但是如果cpu给他的时间片到了，同时另一个进程在它置1之前也进行了调用。就出错了

}else

{ 现在有人已经打开了，不能使用}

}

problem：1个程序运行时flag准备推出显示0，在推出时，时间片刚好完了，此时还未正常推出，另一个程序不能继续open的。

原子操作：（每个函数就是一条命令，一定会执行完成，不管时间片还有没有都能正常 执行完运行）

之前的程序不会被切断

自旋锁：在操作临界资源的过程前后加锁解锁\*\*\*\*\*保证操作全局变量的过程即使被打断，别人也没办法操作这个全局变量，

对共享资源枷锁，保护共享资源（注意只是在全局变量操作前后加解锁，不能中间隔得太远，不然别的程序获取不了锁，会卡死）

运用于内核进程间，只能被一个内核任务（进程或中断）持有，抢锁抢不到CPU会忙轮询，旋转，所以一般尽量少用。

自旋锁怎么保证临界区资源，有可能持有锁的时候被中断掉，还是可能破坏临界区，所以衍生出别的函数：

spin\_lock\_irqsave(获得锁之前禁止cpu中断，将中断状态保存在flag参数中)

spin\_lock\_irq（获得锁之前禁止cpu中断，但不保存中断状态）

spin\_lock\_bh（禁止硬件中断，允许软中断）

spin\_lock();自旋锁

spin\_unlock();解锁

down(&sem);内核的信号量

up(&sem);

内核的进程与进程之间用信号量去通信（信号量可以有多个），若初始化为1个就成了互斥信号量

进程与其他内核代码或者中断中保护临界区时使用：自旋锁

\*\*\*\*如果只用1个全局变量，不用锁，open一个文件后，还来不及修改全局变量，另一个进程也可以打开这个文件，这样就会有问题了

$:gnome-system-monitor---任务管理器

信号量要点：在驱动中全局变量定义：struct semaphore mysem;

在驱动初始化函数中初始化信号量：sema\_init(&mysem, 1);

在hello\_open函数中：down\_interruptible(&mysem); //获取信号量，sem-1

在hello\_close函数:up(&mysem); // 释放信号量 sem+1

信号量的值只有1--0 二值信号

如果有2个打印机（2个资源），可以信号量初始为2，第一次操作可以用其中1个打印机，第二次用第二个

互斥体使用与信号量类似

阻塞非阻塞：（看drive目录中应用程序的对比）

针对非阻塞，驱动怎么实现

初始化时init\_waitqueue\_head(&myqueue);初始化队列头

睡眠与while1：程序阻塞的时候是睡眠，cpu不会升高，只会卡住程序

while1cpu升高，卡死程序

sleep是延时运行（程序也在运行，滴答定时器，中断弄得）

unistd中的usleep（n）---us级的延时

### 内存映射机制

1. io内存空间映射

映射物理寄存器的地址到内核空间，得到其在内核空间的虚地址

p = ioremap（0x2432423423,4）地址长度 将物理地址（soc外部外设的真实的地址）映射为内核中的虚拟地址（3-4g）

最后可以通过往虚拟地址写值，写到物理地址中

实际上ioremap是按页进行映射的，整页整页的去映射。假设物理地址是0x1002,size=4

a.物理地址按页取整，得到地址0x10000

b.size按页取整，size=4096

c.把起始地址为0x10000，size4096的空间映射到虚拟地址空间，假设虚拟地址的起始地址为

0xf0010000,则需要映射的物理地址0x1002对应的虚拟地址为：0xf0010002

2.内核空间操作该虚地址，操作它就是操作真正的物理寄存器

val = ioread(p)

iowrite(val,p)

3.iounmap

ioreadl/iowritel-> 对io内存的op

iol/iob/iow->对硬件地址读

outb/outl/outw->对硬件地址写数据

linux内核内存分配

按页申请内存

\_\_get\_free\_pages(int gfp\_mask,ulong order);

mask----属性

order---0----2^0页 ----n-----2^n页

分配的内存物理上是连续的

按页申请内存：

unsigned long \_\_get\_free\_pages(int gfp\_mask, unsiged long order)

mask ----申请内存的属性

order -------------0 ----------- 2^0 1页

1 ---------- 2^1 2 页

1页 ==== 4096 字节

大页：2M ---适用于GB级别的内存管理

大页：1GB ---适用于TB级别内存管理

配置大页内存后：OS开机后预留连续的大小为hugesize\*hugepages 大小的内存

若内存不足虚拟机启动会报错out of memory

使用大页内存可以减少OS的管理与访问页面的时间，内核中的swap进程不会占用此部分内存的。大页合理运用提升OS性能

分配的内存物理上是连续的 ， 申请到的内存的大小一定是2 的order 次幂

使用该函数一次可以申请的最大的内存 1024页 \* 4096字节 ===== 4m 字节

申请的内存 内核空间使用

void free\_pages(unsigned long addr, unsigned long order)

按精确的字节数申请：

static \_\_always\_inline void \*kmalloc(size\_t size, gfp\_t flags)

size ---- 具体的字节数

flags ---- 申请内存的属性

按具体的字节数申请内存， 物理地址是连续的 ，

申请的内存 内核空间使用

flags :

GFP\_KERNEL - 申请内核空间的一段内存时， 可以睡眠， 正常情况下用它

GFP\_ATOMIC----- 申请内存时不能睡眠， 主要用在中断的处理函数里，

void kfree(const void \*objp)

vmalloc ：

void \*vmalloc(unsigned long size)

申请精确字节的内存， 内核空间使用，

物理地址不一定连续

void \*p; 碎片如果很多但是不连续

p = vmalloc(30); 可以申请到， p[0] , p[1],p..., p[29],

p =kmalloc(30) ; 申请不到

void vfree(const void \*addr)

内存对象池：

内核为了满足任意数量的内存使用请求，创建了由各种不同固定大小的内存块组成的内存对象池 当kmalloc申请内存空间时，内核就会将

一个刚好足够大的空闲内存块分配出来。比如要申请100字节，kmalloc就会返回一个128字节的内存块

在4k字节大小页面的系统上，分配的最小内存块是32字节，最大128k字节

设备驱动程序如果常常反复使用同样大小的内存块，可以自己创建一个内存池

内核空间内存：

线性映射区域：3g开始的虚地址 和 物理内存0地址开始的 一段内存 一一对应的关系 。

高端内存（内核空间的一段虚地址）：很少的高端内存的虚地址 可以访问到 更大的物理地址空间 ，

mmap 方法：之前操作文件用open的fd，现在用指针操作就可以，效果一样可以read，write

应用层里：

void \*mmap(void \*addr, size\_t length, int prot, int flags,

int fd, off\_t offset);

把内核的一段内存， 映射到用户空间(对应到打开的某个文件中，所以有个fd)

addr ------------ 映射到用户空间的地址， NULL---- 映射到用户空间的地址由内核决定， 可以返回映射到用户空间的地址，这里填NULL就好

length ------------ 把内核空间的一段内存， 映射到用户空间， 映射多大？(文件有多大就映射多大)

prot ------------ 内核空间的一段内存，映射到用户空间， 映射属性，映射之后是可读，可写？PROT\_READ | PROT\_WRITE

flags ------------- 内核空间的内存的映射方式， 共享的映射， 私有的映射？MAP\_SHARED（对用户空间修改的时候内核

空间的对应地址也同步修改，MAP\_PRIVATE，对用户空间的地址的修改不会影响内核空间的地址的数据）

fd ---------------- 从哪里进行映射？（只要mmap返回了，fd就没用了，可以直接close就好）

offset ----------- 映射时的偏差地址，

返回值： 内核空间的内存， 映射到用户空间的地址

int munmap(void \*addr, size\_t length);

解除映射

使用mmap必须注意以下：

1.创建映射区的过程中，隐含1次对映射文件的读操作-所以fd的打开应可读可写，mmap中也应该可读可写

2.当MAP\_SHARED时映射区的权限应<=文件打开的权限

当MAP\_PRIVATE无所谓，因为mmap中的权限是对内存的限制

3.映射区的释放与文件变比无关，只要映射建立成功，文件可以立即关闭都可以

4.当文件的映射大小为0时，不能创建映射区，用于建立映射区的文件必须有实际大小(一般新建的文件大小就是0，得写入东西才可以映射)

5.munmap传入的地址一定是mmap的返回地址，坚决杜绝指针的++ --操作。你可以char \* p = ret.去操作p，而不是操作ret

6.文件的偏移量必须是4K的整数倍(mmu映射的内存就是4K 4K的映射的)

7.一定要检查返回值(mmap成功返回void\*指针，失败返回MAP\_FAILED宏)

用户空间的页面允许中断，内核空间的不允许中断。

驱动里：

实现mmap 方法,

/\*\*

映射内核空间的一段内存到用户空间

\* @vma: 用户映射内核内存的要求

\* @addr: 映射到用户空间的地址

\* @pfn: 内核空间的一段内存的物理地址（以页为单位）

\* @size: 映射多大

\* @prot: 映射属性

\*/

int remap\_pfn\_range(struct vm\_area\_struct \*vma, unsigned long addr,

unsigned long pfn, unsigned long size, pgprot\_t prot)

virt\_to\_phys（pkbuf） >> 12 虚地址转换成物理地址, 物理地址/ 4096 得到pkbuf 的内存的页码

## Mcu内部通讯机制

### I2c学习：

IIC发送时先发最高位，串口发送时先发最低位。

发送数据的过程中：

scl为低时，发送器发送信号，sda可以改变高低

sdl为高时，接收器接收信号，sda必须保持，不可改变

收发双方共用SCL，所以双方时间基准完全一致，是同步通信方式。

可见：SCL是低时发送器发，SCL为高时（接收器读数据或者发ack）

如果从机一直不应答ack，主机自动stop。

直接touch + 已存在的文件名相当于把文件重新写一下，不会让之前的内容消失.目的是更改时间戳

iic的芯片（8脚的为例）：7根线表示地址（7中的4个已固定为1010 ），其他3个(a2 a1 a0)接线确定h/l

1 0 1 0 a2 a1 a0 r/w

iic速度较低，用它传视频不可能，传输距离只有几米，不是一对一的，是多对多的。

主机不一定是发送器，只有第一个byte是主机发送给从机的，这个byte的最后一个bit决定谁是发送器，谁是接收器，

数据传输的过程中就会一直保持为发送器或接收器，直到重新发送start信号更换读写的状态。通信的开始结束信号都是主机发起的，start与

stop信号之间，发送器可以一直发，接收器可以一直读。

通讯开始之前scl与sda都是一直是H电平。

start信号：scl为H时，sda由高变低，start信号

stop信号：scl为H时，sda由低变高。

ack信号：scl为低时，从机拉低sda，作为响应。

通讯开始：sda由高变低，start信号

接下来的7个sck为h的时候，sda发1 0 1 0 0 0 0即从机地址

第8个sck为h时，主机发送r/w要求。

第9个sck为h的时候，从机给主机发ack信号。-------------建立好握手可以实现数据发送

10--17个sck中，sck上发送8位数据....算是地址..第一次发的数据就是地址

18sck中从机回ack表示接受完成

19sck中stop信号触发通讯结束

iic寄存器说明：

con：sck速度，状态位（正在传输、已传完），使能ack，使能tx/rx

stat状态：主机作为发送/接收，设置开始/停止条件，数据Tx/rx使能，ack能否被接收（\*\*主方提供时钟\*\*）

dat：可以装设备地址也可以收发8bit的数据

传输时1次传输8bit，高位在前低位在后

总线空闲状态：sda为H

陀螺仪：mpu6050 三轴传感器

fs4412硬件资源：

gpio接口驱动：pwm接口驱动：按键中断驱动：adc接口驱动：spi接口驱动：iic接口驱动：

mpu6050.c ：物理总线iic下的驱动，真实的总线的驱动

驱动中还是探测函数--->探测函数会带过来iic设备pdevice

平台驱动：虚拟总线下的驱动

stm32杜洋的sendbyte(u8 slaveaddr器件地址,u8 slave内部寄存器地址，u8 pdatabuf)函数，主机发数据

每个iic器件内部有很多寄存器（每个寄存器存放对应的数据），你要往哪个寄存器写数据/或者从哪里读数据，

时序图，发一下要while（）等一下，是在干嘛啊？？是接收ack信号？？还是为了凑时序啊

发送start信号--等待

发送从机地址--等待

发送器件内部寄存器地址--等待

发送要发的数据--等待

发送结束信号（）--------结束信号是我主机发送的，告诉你我发完了

主机接受数据：

recvbyte(slaveaddr,从机寄存器地址（数据寄存器）)

while（）从机忙的时候，等待阻塞

不忙了发送start信号--等待

发送从机地址--等待

发送器件内部寄存器地址--等待

设置单片机允许其他器件产生开始信号-开启iic的接收（其他器件就可以向iic发送数据了）

将接收buffer的数据返回

读写eeprom例子：没有crc校验，一般除了存储芯片外的其他iic都会有crc校验

根据扇区进行操作

针对板子的驱动

步骤：初始化iic相关的gpio（给1个结构体赋值，然后调用iic\_init）gpio作为输入时速度没啥用，只有做输出时才有用

工作模式：iic相关的cr寄存器结构体 赋值，然后调函数初始化

iic\_byte\_write（）

{

产生start信号——>检测ev5事件，看start是否成功。while检测ev5，给个延时

senaddr发子设备地址->检测ev6事件，等待检测成功while-ev6 给个延时

senddata（）第一次发的数据会被认为子设备内部reg地址 ，while->ev8

sendata（）发数据 while检测ev8标志位

产生stop信号

}

read\_random(n，\*data)

{

1.start+写地址 要从哪里读

2.start+读地址

while（n--）

{

若接收buf满，则\*data = recvdata

if（n==1）

{

是能nack 收到最后1byte

\*data = recvdata；

}

data++;

}

}

linxu下的iic驱动：有几个结构体（）内核给我们准备了iic自己的发送与接受方法

iic\_client：每个clie对应1个drivernt

iic\_driver：

iic\_adapter：总线适配器。就是控制器 （在设备树中把你的iic设备天加到对应的iic适配器下就好了）

iic{

iic1{

地址

}

iic2{

}

}

iic\_algorithm：描述适配器与从设备的算法

linux下不用关心怎么控制寄存器进行iic的操作。因为内核有写好的函数。iic的厂家也写好了总线驱动

i2c\_transfer(adapter,struct iic\_msg \*msg,int num);

内核就有，这个函数。

流程：

1.struct i2c\_driver mpu6050\_driver = {.of\_match\_table = of\_match\_ptr(mpu6050\_dt\_match),}定义平台驱动，从设备树找

2.入口函数i2c\_register\_driver(THIS\_MODULE,&mpu6050\_driver);挂自己定义的struct i2c\_driver 若存在会跳到探测函数

3.int mpu6050\_probe(struct i2c\_client \*pclient, const struct i2c\_device\_id \*id)

探测函数会拿到iic的资源pclient。

{

正常的字符设备号申请...设备文件创建..

}

mpu6050\_read\_byte(struct i2c\_client \*pclient, unsigned char reg)

这个函数用linux内核提供的i2c\_transfer(adapter,struct iic\_msg \*msg,int num)完成的。

struct iic\_msg \*msg{

从芯片的地址（探测的资源的成员），读还是写（0/1），2（几个数据），数据char buf【2】

}

int mpu6050\_write\_byte(struct i2c\_client \*pclient, unsigned char reg, unsigned char val)

具体实现：

static int mpu6050\_read\_byte(struct i2c\_client \*pclient, unsigned char reg)

{

char txbuf[1] = { reg };

char rxbuf[1];

struct i2c\_msg msg[2] = {

{client->addr, 0, 1, txbuf},

{client->addr, 1, 1, rxbuf}

};

先发写地址我要reg读，再发读地址，用rxbuf去保存

ret = i2c\_transfer(pclient->adapter, msg, ARRAY\_SIZE(msg));

if (ret < 0) {

printk("ret = %d\n", ret);

return ret;

}

return rxbuf[0];

}

static int mpu6050\_write\_byte(struct i2c\_client \*pclient, unsigned char reg, unsigned char val) iic寄存器地址

{

char txbuf[2] = {reg, val};

struct i2c\_msg msg[1] = {

{client->addr, 0, 2, txbuf},

};

i2c\_transfer(pclient->adapter, msg, ARRAY\_SIZE(msg));

return 0;

}

### Spi学习：

时钟是主机产生的，每个spi周期传输1bit数据

主机主动拉低cs，表示我要与某个spi设备进行通信，然后sck就开始跳动了。（刚开始sck是0或1极性设置）

LCD模块的通信（芯片之间的通讯）

高速、串行（只有2根线是收发）、全双工、同步总线（总线就是多对多，usart就不是总线1对1）

极性：在未开始通信前（或者说空闲状态下）的sck的电平0/1 可以设置

相位：0--->前沿采样后沿输出 1--->前沿输出（发送）后沿采样（获取）（通过这个双方同步）（沿：上升沿或下降沿）根据sck的沿

速度：目前最快3.4M/s 输入口速度48M，所以要预分频。确定波特率（sck）

First bit：LSB/MSB

（lsb不是低位先传，而是低字节先传，是以字节为单位的）

同理msb：不是高位先传，应该是高字节先传，一般的机器的存储都是lsb

主机驱动mosi信号----采样miso信号

spi的通讯双方的极性与相位要一致（配置）

编程时每个spi模块都有自己的一套寄存器，就地址不一样，名字功能都基本一样

配置conr：开启中断|使能spi|发完中断？|设主机还是从机|极性|相位|先lsb|msb

baud：bus总线过来的时钟 / 分频因子 = baud

状态reg：接收完成flag（接收缓冲区满）| 可以发送（发送buf空）

spi+flash： w25q64bv

spi最高速度80m，stm32这个总线上72m，二分频才36m

spi的send与recv是同一个函数：发送的同时也在接收，

{

while（txbuf如果不为空）；则等待

txbuf为空了，则sendata（data）；放到发送寄存器就不用管的，他会自己1bit1bit的移位出去

while（rxbuf不满，则等待）

return 接收buf满了直接返回数

}

### Adc

一般soc内部用的逐次逼近型。猜数字量转换成模拟量与实际模拟量对比，多猜几次结果就对了

采样有单端采样，差分采样模式

采样精度不高的adc基准源直接接ACC就可以。

采用不同采样分辨率，时钟设置不一样（分辨率高的采样慢）

编程配置：

1。打开adc模块时钟源（等会用的时候自己再分频）+pin脚对应的port组的时钟

2.pin脚功能设置为adc用（不是设普通gpio）

3.

1）adc时钟分频设置

2）采样时间（long time /short time）

3）用多少精度

mux：选择哪个通道呢

trigger方式：软件/硬件

连续采样/1次采样（连续采样比较耗电，费cpu）

数据从adc0的14mux读

这条不是初始化设置的：观察采样完成的flag（采样1次是否完成），读数据后把这1bit擦除了

adccon： 分辨率、FLAG转换是否结束、时钟使能、分频、工作模式（省电standby、正常）、启动方式（置1转换开始，转换完自动为0）

exynos4412两种启动方式0bit 1bit都可以启动，我们选择0bit启动（启动后这bit自动清0）

adcdata：数据reg

adcmux：通道选择reg

End of conversion flag(Read only) 当它置1，值才可读