## 1.可变形参

#include <stdarg.h>

void va\_start(va\_list ap, last);

type va\_arg(va\_list ap, type);

void va\_end(va\_list ap);

void va\_copy(va\_list dest, va\_list src);

用法:

va\_start(ap,fmt);

while(\*fmt)

{

switch(\*fmt++)

{

case 'd':

i = va\_arg(ap,long);

//或者

//i = va\_arg(ap,int);

printf("d:%d ",i);

break;

case 'f':

f = va\_arg(ap,double);

printf("f:%.2lf ",f);

break;

case 'c':

c = (char)va\_arg(ap,int);

printf("c:%c ",c);

break;

case 's':

s = va\_arg(ap,char\*);

printf("s:%s ",s);

break;

}

}

putchar('\n');

va\_end(ap);

## 2.目录操作

### 2.1基本函数

#include <sys/types.h>//目录操作头文件

#include <dirent.h>

struct dirent {

ino\_t d\_ino; /\* 节点号 \*/

off\_t d\_off; /\* 偏移量 \*/

unsigned short d\_reclen; /\* 文件的长度\*/

unsigned char d\_type; /\* 文件的类型，目录==4，普通文件==8\*/

char d\_name[256]; /\* 文件名称 \*/

};

/\*创建目录之前，需要先执行 umask（0） 然后再调用 mkdir 函数。\*/

umask(0);

mkdir(argv[1],S\_IRWXO);

DIR \*opendir(const char \*name);//

int closedir(DIR \*dirp)；

struct dirent \*readdir(DIR \*dirp);

int readdir\_r(DIR \*dirp, struct dirent \*entry, struct dirent \*\*result); //readdir 的可重入版本

void seekdir(DIR \*dirp, long offset);//设置 readfir 函数的读取指针位置。

off\_t telldir(DIR \*dir); //返回目录流当前的读取位置

int mkdir(const char \*pathname, mode\_t mode); //创建目录,mode 是目录权限。

int rmdir(const char \*pathname); //删除目录。只能删除空目录

int chmod(const char\* path, mode\_t mode); //mode 形如： 0777

### 2.2获取目录名

#include <unistd.h>

char \*getcwd(char \*buf, size\_t size); //获取当前目录，相当于 pwd 命令,输入的路径除了最后一层

char \*getwd(char \*buf);

char \*get\_current\_dir\_name(void);

int chdir(const char \*path); //修改当前目录，即切换目录，相当于 cd 命令

### 2.3剔除目录

#include <libgen.h>//剔除目录函数的头文件

char \*basename(char \*path); //只留下最后一层

### 2.4字符串拷贝

#include <string.h>//字符串拷贝

char \*strdup(const char \*s);

char \*strndup(const char \*s, size\_t n);

char \*strdupa(const char \*s);

char \*strndupa(const char \*s, size\_t n);

## 3.获取文件状态

#include <sys/stat.h>//获取文件状态函数头文件

#include <sys/types.h>

#include <unistd.h>

struct stat {

dev\_t st\_dev; /\* 文件所在设备的标识 \*/

ino\_t st\_ino; /\* 文件结点号 \*/

mode\_t st\_mode; /\* 文件保护模式 \*/

nlink\_t st\_nlink; /\* 硬连接数 \*/

uid\_t st\_uid; /\* 文件用户标识 \*/

gid\_t st\_gid; /\* 文件用户组标识 \*/

dev\_t st\_rdev; /\* 文件所表示的特殊设备文件的设备标识 \*/

off\_t st\_size; /\* 总大小，单位为字节\*/

blksize\_t st\_blksize; /\* 文件系统的块大小 \*/

blkcnt\_t st\_blocks; /\* 分配给文件的块的数量， 512 字节为单元 \*/

time\_t st\_atime; /\* 最后访问时间 time\_t 实际就一个长整型，保存的是秒数，可以使用 ctime 函数转为字符

串\*/

time\_t st\_mtime; /\* 最后修改时间 \*/

time\_t st\_ctime; /\* 最后状态改变时间 \*/

};

S\_ISLNK(st\_mode): 是否是一个连接.

S\_ISREG()是否是一个常规文件.

S\_ISDIR()是否是一个目录

S\_ISCHR()是否是一个字符设备.

S\_ISBLK()是否是一个块设备

S\_ISFIFO()是否是一个 FIFO 文件.

S\_ISSOCK()是否是一个 SOCKET 文件.

int stat(const char \*path, struct stat \*buf); //获取文件的状态保存在 buf 里。（可以直接传入文件路径）

int fstat(int fd, struct stat \*buf); //获取文件的状态保存在 buf 里（针对已经打开的文件进行操作）

int lstat(const char \*path, struct stat \*buf);//获取文件的状态保存在 buf 里。（不区分链接文件）

## 4.基于文件指针的文件操作函数

#include <stdio.h> //基于文件指针的函数

FILE \*fopen(const char \*path,const char \*mode);

FILE \*fdopen(int filds,const char \*mode);

FILE \*freopen(const char \*path,const char \*mode, FILE \*stream);

int fclose(FILE \*stream);

int fgetc(FILE \*fp)；//文件中获得一个字符

int fputc(int ch,FILE \*fp)；//向文件写入一个字符

char \*fgets(char \*str,int n,FILE \*fp);

int fputs(char \*str,FILE \*fp)；

int fprintf(FILE \*fp,char \*format,arg\_list)；//向文件写格式化数据

int fscanf (FILE \*fp,char \*format,arg\_list)；//格式化读数据

int fread(void \*buffer,unsigned sife,unsigned count,FILE \*fp)

int fwrite(void \*buffer,unsigned sife,unsigned count,FILE \*fp)

int getw(FILE \*fp) //以二进制形式到文件读取一个整数

int putw(int n,FILE \*fp) //以二进制形式向文件存一个整数

int feof(FILE \*fp)；//用来侦测读写文件指针是否读取到了文件尾。

long ftell(FILE \*fp)；//文件指针当前位置

void rewind(FILE \*fp)；//回到文件指针头

int fseek(FILE \*fp,long offset,int base)；//定位 参数3：SEEK\_SET SEEK\_CUR SEEK\_END

FILE \*tmpfile(void) //以二进制更新模式(wb+)创建临时文件。被创建的临时文件会在流关闭的时候或者在程序终止的时候自动删除。

int fsetpos(FILE \*stream, fpos\_t \*pos);

int fgetpos(FILE \*stream, fpos\_t \*pos);

int rename(const char \*oldpath, const char \*newpath);//重命名

## 5.基于文件描述符的文件操作函数

#include <fcntl.h>//文件描述符的头文件

#include <unistd.h>

int open(const char \*pathname, int flags); //文件名 打开方式

int open(const char \*pathname, int flags, mode\_t mode);//文件名 打开方式 权限

int close(int fd); //关闭文件， fd 表示文件描述符

size\_t read(int fd, void \*buf, size\_t count);//文件描述词 缓冲区 长度

size\_t write(int fd, const void \*buf, size\_t count);

#include <stdio.h>

FILE \* fdopen(int filedes, const char \*opentype); //文件指针与文件描述符的转换

int fileno(FILE \*stream);//把文件流指针转换成文件描述符用 fileno 函数

int remove(const char \*pathname);//删除文件或目录

int unlink(const char \*filename) //删除文件

## 6.形参解析函数

#include <unistd.h>//形参解析函数头文件

int getopt(int argc, char \* const argv[], const char \*optstring);//只支持短形参-r等等

extern char \*optarg; //头文件里面的全局变量，可直接调用

extern int optind, opterr, optopt;

第三个参数传入想要定义到参数类型，如-a，-b，-c等，填入形式"abc",或"a:b:c:",加:可以在这个参数后面传入字符串

#include <getopt.h>

int getopt\_long(int argc, char \* const argv[], const char \*optstring,const struct option \*longopts, int \*longindex);//兼容短形参-r和长形参--r

### 6.1使用

struct option ionion[]=

{

{"yjw",1,NULL,'y'},

{"sll",1,NULL,'s'},

{"lzj",0,NULL,'l'},//0代表不要参数带字符串

{0,0,0,0},

};

int res=0;

int longindex=0;

while(1)

{

res = getopt\_long(argc,argv,"a:b:c:d:",ionion,&longindex);

if(res==-1)break;

switch(res)

{

case 'a':

printf("-a:%s\n",optarg);

break;

case 'b':

printf("-b:%s\n",optarg);

break;

case 'c':

printf("-c:%s\n",optarg);

break;

case 'd':

printf("-d:%s\n",optarg);

break;

case 'y':

printf("-yjw:%s offset:%d\n",optarg,longindex);

break;

case 's':

printf("-sll:%s offset:%d\n",optarg,longindex);

break;

case 'l':

printf("-lzj:%s offset:%d\n",optarg,longindex);

//printf("-lzj\n");

break;

}

}

## 7.打开动态库的函数

#include <dlfcn.h>//打开动态库的函数头文件

void \*dlopen(const char \*filename, int flag);//动态库的路径加名字，

char \*dlerror(void);

void \*dlsym(void \*handle, const char \*symbol);//dlopen的返回值，函数名或者变量名

int dlclose(void \*handle);

编译时加-ldl

### 7.1使用:

/\*核心:打开动态库,\*/

void \*handle;

int (\*add)(int,int,char);//保存获取到的函数的函数指针

int \*data;

handle = dlopen("sum/libsum.so",RTLD\_LAZY);

if(handle==NULL)

{

printf("找库失败\n");

exit(0);

}

//调用,函数"sum"或者动态库中的全局变量"data"

add = dlsym(handle,"sum");

data = dlsym(handle,"ddd");

if(add==NULL)

{

printf("找函数失败\n");

exit(0);

}

int a = atoi(argv[1]);

int b = atoi(argv[2]);

char c = argv[3][0];

int z = add(a,b,c);

printf("ddd:%d\n",\*data);

printf("运算结果:%d\n",z);

//关闭

dlclose(handle);

## 8获取环境变量的函数

#include <stdlib.h> //获取环境变量的函数头文件

/\*全部只在当前进程里有效\*/

char \*getenv(const char \*name); //获取环境变量的值 常用

int clearenv(void); //清除环境变量的值

int setenv(const char \*name, const char \*value, int overwrite);// 用来改变或增加环境变量的内容。

int putenv(char \*string); //用来改变或者增加环境变量的内容

int unsetenv(const char\* name) //删除 name 环境变量的定义，即使不存在也不会出错

### 8.1使用：

char \*env\_p;

env\_p=getenv("PATH");

if(env\_p==NULL)exit(0);

printf("PATH=%s\n",env\_p);

## 9.automake自动生成Makefile工具简单使用:

1.写好代码后顶层目录下敲命令:autoscan

2.把生成的configure.scan改为.ac后缀

3.修改.ac后缀文件的内容

:AC\_INIT([yjw], [1.2.3], [2453013768@qq.com]) //三个方括号内改为自己想要的东西，分别代表：生成的可执行名、版本号、联系地址

AC\_CONFIG\_SRCDIR([main/main.c]) //自动识别包含main函数的主文件，但是又可能识别失败，需要自行修改

AC\_PROG\_CC //默认gcc编译器，修改可在变量后直接跟小括号，内填编译器名

AC\_PROG\_RANLIB //当需要生成静态库时，要加上此全局变量(动态库宏: AC\_PROG\_LIBTOO)

AM\_INIT\_AUTOMAKE(yjw,1.2.3) //都要加上这句话，两个参数和上面的要一致

AC\_OUTPUT //该变量指定在哪生成Makefile文件,如:

// AC\_OUTPUT(Makefile main/Makefile sub/Makefile sum/Makefile)

**PS:**（如果是动态库，除了改变.ac里面的静态库宏为动态宏以外，还要把所有.am中的.a后缀改为.la，并且所有宏也要改为对应的宏。然后在执行下面步骤之前先执行: libtoolize -f -c）

4.执行 aclocal 命令生成 aclocal.m4 文件

5.使用 autoconf 工具生成 configure 文件

6.使用 autoheader 生成 config.h.in 文件 //4、5、6是分别打上命令回车即可

7.在AC\_OUT指定的几个地方分别创建Makefile.am文件,打开分别编写各自的内容（用到的变量和方法参考万邦automake教程,可以放在第8步之前的任何位置）

8.执行automake --add-missing

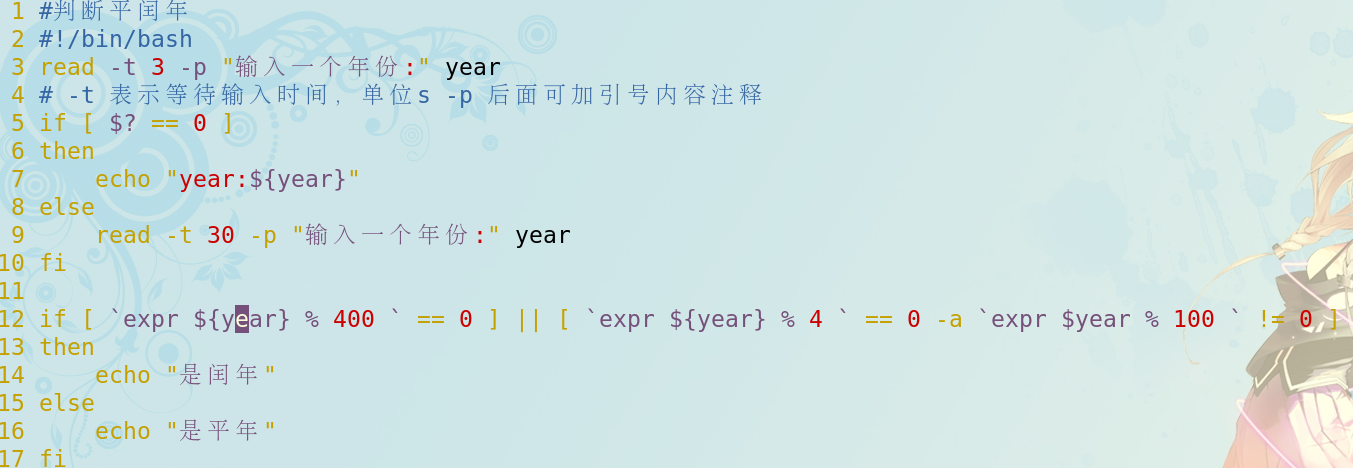
9.执行./configure --prefix=路径

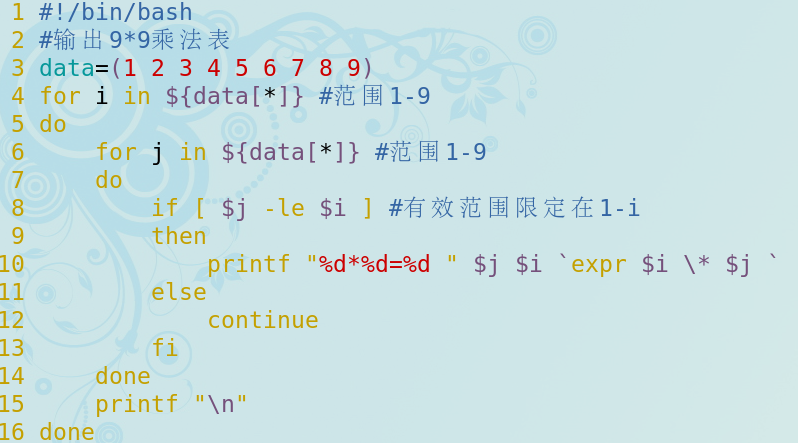
10.执行make 进行编译

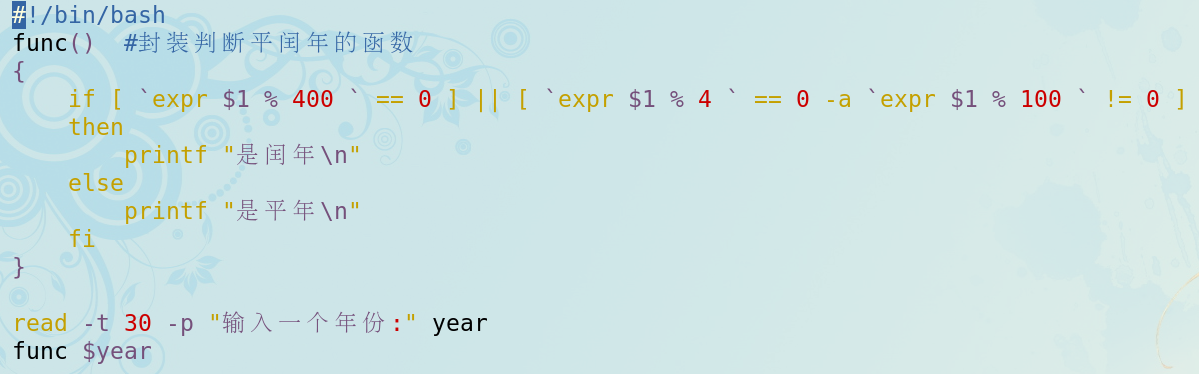
11.执行make install 进行安装

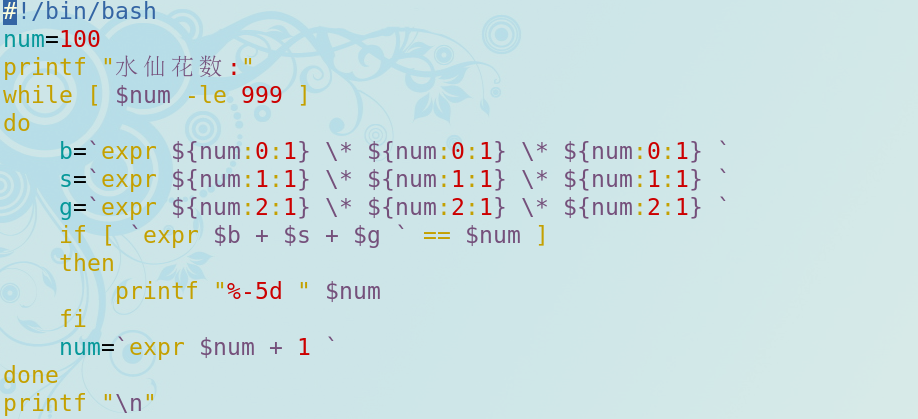
## 10.shell

### 10.1示例

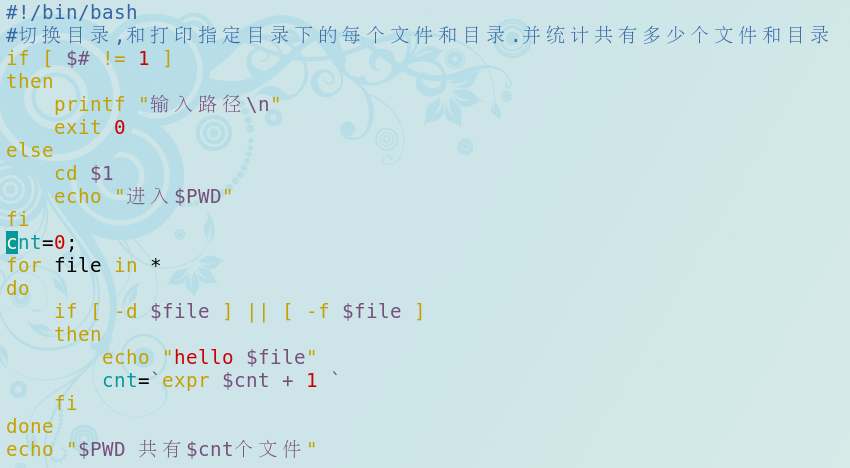


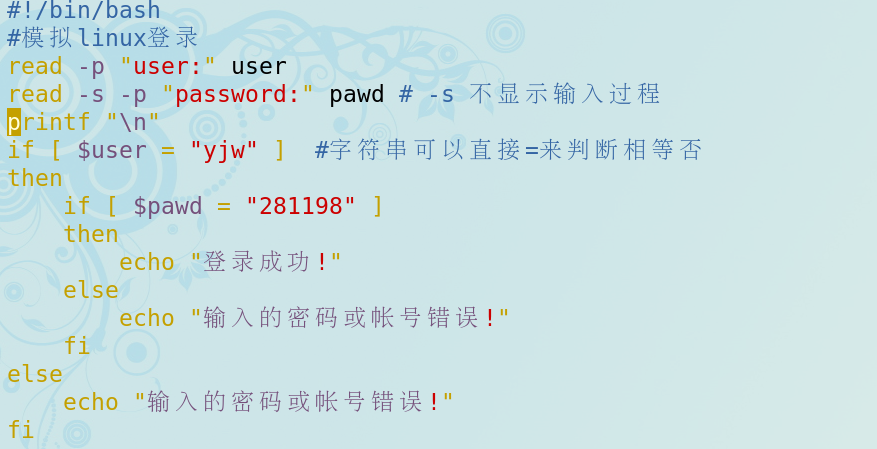


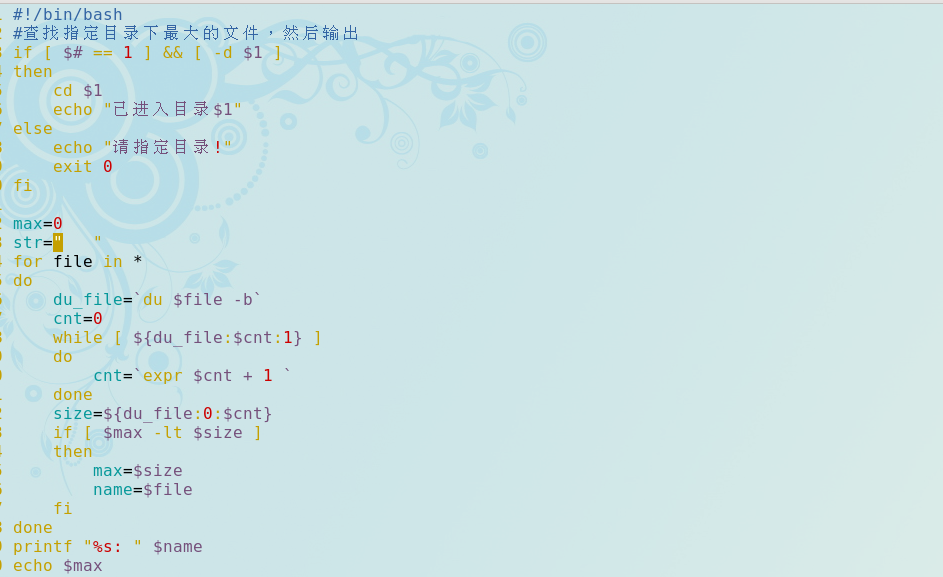












## 11.进程

### 11.1函数

**fg** :后台进程搬移到前台

**bg :**后台暂停进程开始

**Ctrl+z :**把正在运行的进程暂停并放在后台

**nice**

**renice**

**ps**

**jobs :**查看后台的进程

**top :**动态查看进程

**& :**加在命令后，把该命令放在后台执行

**kill**

**fork** :

**getpid :**得到PID号

exit() 和\_exit() 第一个退出时会清除IO缓冲区数据，输出到标准输出，再调用exit的系统调用退出程序。第二个直接调用系统调用，不会清除IO缓冲区。如：把printf(“111111”);打印函数后分别调用exit和\_exit结果不同，exit有数据打印在屏幕.(不能加\n,\n为行缓冲的标志)

int pause(void); //暂停程序, 暂停进程在收到信号后可以自行继续运行

#### 捕获信号函数

include <signal.h>

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);//形参：创建一个符合函数指针类型的函数，当作第二个参数，第一个参数填入linux系统信号。

Kill -l 可以查看系统全部信号及其编号

#### 发送信号的方法:

kill [-s signal|-p] [--] pid...

kill -l [signal]

kill -s 2 1234 //给PID为1234的进程发送2这个信号

kill -2 1234 //给PID为1234的进程发送2这个信号

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig); //通过C语言函数接口给指定的进程发送信号

#### 系统调用类：

System() 传入可执行函数绝对路径，不会对调用者产生影响。

捕获流管道：FILE \*popen(const char \*command, const char \*type); //流管道，参数：命令，打开方式“r”“w”

示例 popen("ls \*.mp3","r");

exec族:

系统调用，会覆盖调用者进程的函数空间，调用者代码失效

#include <unistd.h>

extern char \*\*environ;

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg,..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

使用方法:path表示传入绝对路径，file表示传入可执行文件名(必须存在在PATH的值下)

//char \*const argv\_cmd[] = {“ls”,”-l”,”\*.c”}

//execl("/bin/ls","ls","-l",NULL);

//execlp("ls","ls","-l",NULL);

//execv("/bin/ls",argv\_cmd);

//execl("/work/a.out","a.out","123","456",NULL);

//execvp("ls",argv\_cmd);

### 11.2进程间通信

#### 管道:

有阻塞功能，管道写端没数据发送过来时会一直等待(不分命名和无名管道),适合一对一通信

**命名管道：**任何进程都可以进行通信

命令行:mkfifo xxx:创建fifo文件(也就是命名管道,有实体，可以在不同进程间通信)

函数: int mkfifo(const char \*pathname,mode\_t mode);

**无名管道：**只能亲戚关系的进程间通信方式

int pipe(int pipefd[2]); #创建无名管道,无名管道就是没有名字也就是没有实体，只能在有亲戚关系的进程间通信 。pipefd[2];//0是读端，1是写端

**示例**：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int main(int argc,char \*\*argv)  {  pid\_t pid;  char buff[100];  int cnt;  int status;  int pipefd[2]; //[0]表示读 [1]写  //1.创建无名管道  pipe(pipefd);  //2.创建子进程  pid=fork();  if(pid==0)  {  printf("子进程的PID:%d\n",getpid());  //从管道的读端读取数据  cnt=read(pipefd[0],buff,100);  printf("读取的字节:%d,%s\n",cnt,buff);  }  else if(pid>0)  {  printf("父进程的PID:%d\n",getpid());  sleep(3);  //向管道的写端写数据  write(pipefd[1],"12345",6);  //等待子进程结束  wait(&status);  }  else  {  printf("进程创建失败!\n");  }  return 0;  } |

#### 消息队列:

适合一对多通信,可以一个用来发消息，多个用来读消息（ipcs -q来查看当前所有的消息队列情况）。

**写队列:**

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  //int msgget(key\_t key, int msgflg);创建和访问一个消息队列。创建标志的可选值为 IPC\_CREAT  //int msgsnd(int msqid, struct msgbuf \* msgp, size\_t msgsz, int msgflg);将消息添加到队列中  //ssize\_t msgrcv(int msqid, struct msgbuf \* msgp, size\_t msgsz, long msgtyp, int msgflg);  //从一个消息队列中获取信息。msgflg:有 3 种选项:0：当消息队列满时,msgsnd 将会阻塞，直到消息能写进消息队列  //int msgctl(int msqid, int cmd, struct msqid\_ds \* buf);控制函数,常用来删除消息队列。  //消息队列:生成键值的文件不能和共享内存生成键值的文件相同,会产生错误.和创建共享内存大致相同，msqid一般为msgget的返回值 ,msgsed的第二个参数格式是一个结构体类型，定义方式如下  #define BUFF\_SIZE 100  struct msgbuf  {  long mtype; //消息类型必须是 > 0,因为选择读取参数类型为0时，默认读取队列全部消息  char mtext[BUFF\_SIZE]; //数据大小  /\*.....\*/ //可以往后重复前两个格式  /\*.....\*/  };  int main(int argc,char \*\*argv)  {  if(argc!=4){ printf("格式:./a.out <filename> <消息内容> <消息类型-整数>\n");exit(0);}  key\_t key = ftok(argv[1],200);  if(key == -1) {printf("生成key失败!\n"); exit(0);}  int msq\_id = msgget(key,IPC\_CREAT|0666);  if(msq\_id == -1) {printf("消息队列创建失败!\n"); exit(0);}  struct msgbuf msg\_buff;  msg\_buff.mtype = atoi(argv[3]);  strncpy(msg\_buff.mtext,argv[2],BUFF\_SIZE);  printf("key:%d msqid:%d\n",key,msq\_id);  printf("send:%d\n",msgsnd(msq\_id,&msg\_buff,sizeof(struct msgbuf),0));  return 0;  } |

**读队列:**

|  |
| --- |
| //读消息队列  #define BUFF\_SIZE 100  struct msgbuf  {  long mtype; //消息类型必须是 > 0,因为选择读取参数类型为0时，默认读取队列全部消息  char mtext[BUFF\_SIZE]; //数据大小  /\*.....\*/ //可以往后重复前两个格式  /\*.....\*/  };  int main(int argc,char \*\*argv)  {  if(argc!=3){ printf("格式:./a.out <filename> <消息类型-整数>\n");exit(0);}  key\_t key = ftok(argv[1],200);  if(key == -1) {printf("生成key失败!\n"); exit(0);}  int msq\_id = msgget(key,IPC\_CREAT|0666);  if(msq\_id == -1) {printf("消息队列创建失败!\n"); exit(0);}  struct msgbuf msg\_buff;  printf("key:%d msqid:%d\n",key,msq\_id);  while(1)  {  printf("rcv:%d\n",msgrcv(msq\_id,&msg\_buff,sizeof(struct msgbuf),atoi(argv[2]),0));//0表示读取不到就阻塞  printf("type:%d data:%s\n",msg\_buff.mtype,msg\_buff.mtext);  }  //msgctl(msq\_id,IPC\_RMID,NULL);//删除消息队列,参数2通常为 IPC\_RMID 表示删除消息队列。  return 0;  } |

#### 内存映射:

|  |
| --- |
| #include <sys/mman.h>  …  //进行内存映射，将文件直接映射到内存条上，对数据进行地址操作。可选映射模式为共享，其他文件也可访问  //适合大文件复制  //void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset);  //addr:选择映射在内存上的位置，填NULL系统自行分配；lenth选择分配大小；prot表示映射区域的权限，四种:可执行、//读、写、不能存取(PROT\_EXEC、PROT\_READ、PROT\_WRITE、PROT\_NONE);flags表示映射区域的各种特性(一般填MAP\_SHARE//D-对映射区域的写入数>据会复制回文件内，而且允许其他映射该文件的进程共享。)；fd:文件描述符；offset:对于文件//的偏移量，一般0,返回值:分配地址的首地址.  //通过内存映射进行进程通信，多个进程可以同时映射同一个文件到内存空间，只要一个进程对文件进行了修改，其他进程都可以得到修改的数据。关闭对应的文件描述词时不会解除映射.分配的空间大小最多和原文件相同大小，可用int ftruncate(int fd, off\_t length);修改文件大小  int main(int argc,char \*\*argv)  {  if(argc!=2)  {  printf("格式:./a.out <filename>\n");//运行前必须有文件，因为打开方式不带创建  exit(0);  }  int fd;  char \*mem\_p;  struct stat st\_s;  fd=open(argv[1],O\_RDWR);  if(fd<0)  {  printf("打开文件失败!\n");  exit(0);  }  stat(argv[1],&st\_s);  mem\_p=mmap(NULL,st\_s.st\_size,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,0);  printf("size:%d\n",st\_s.st\_size);  if(mem\_p==NULL)  {  printf("内存映射失败!\n");  exit(0);  }  //直接对地址进行操作，也能改变文件内容  printf("原数据:%s\n",mem\_p);  memset(mem\_p,'\0',st\_s.st\_size);  memcpy(mem\_p,"0123488\n",8); //小于原文件内容时，剩下的空间变为'\0',打开为乱码  munmap(mem\_p,st\_s.st\_size);//取消映射,可不取消  close(fd);  return 0;  } |

#### 共享内存:

适合一对多来通信，多个进程可指向同个共享内存

**写内存：**

|  |
| --- |
| #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/types.h>  //key\_t ftok(const char \*pathname, int proj\_id);创建一个关键字,传入一个任意文件和数字，生成唯一键值  //int shmget(key\_t key, size\_t size, int shmflg);用于创建或打开一共享内存段,参数3:IPC\_CREAT,段不存在则创建  //void \*shmat(int shmid, const void \*shmaddr, int shmflg);共享内存段映射到进程空间的某一地址,地址通常填NULL  //int shmdt(const void \*shmaddr);将共享内存段与进程空间分离， 与 shmat 函数相反.并不会销毁空间  //int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);共享内存的控制函数，可以用来删除共享内存段。  //共享内存可以多个进程指向一个内存，只要设置其共享内存标识符一致即可。共享内存段的标识通常应该是 shmget 的  //成功返回值。这里传入文件只是用来生成键值，并没有对文件产生影响  //root权限运行！！！！！！！！！！！！！!!!!!!!!!!!!!!!!!!!!!!!!!!!!  #define MEM\_LEN 1024  int main(int argc,char \*\* argv)  {  if(argc!=3){printf("格式:./a.out <filename> <通信内容>\n");exit(0);}  key\_t key = ftok(argv[1],200);  if(key == -1 ) { printf("生成key失败!\n"); exit(0);}  int shm\_id = shmget(key,MEM\_LEN,IPC\_CREAT);  if(shm\_id < 0) { printf("生成内存标识符失败!\n"); exit(0); }  char \*mem\_addr = shmat(shm\_id,NULL,0);  if(mem\_addr==NULL) {printf("共享内存映射失败!\n"); exit(0); }  printf("key:%d id:%d addr:%x\n",key,shm\_id,mem\_addr);  memset(mem\_addr,0,MEM\_LEN);//共享内存映射成功，来写数据进行传输  strncpy(mem\_addr,"你是最胖的!",MEM\_LEN);  strcat(mem\_addr,argv[2]);  shmdt(mem\_addr);  return 0;  } |

**读内存:**

|  |
| --- |
| #define MEM\_LEN 1024  int main(int argc,char \*\* argv)  {  if(argc!=2)  {  printf("格式:./a.out <filename>\n");  exit(0);  }  key\_t key;  int shm\_id;  char \*mem\_addr;  key = ftok(argv[1],200);  if(key == -1) { printf("生成key失败!\n"); exit(0);}  shm\_id = shmget(key,MEM\_LEN,IPC\_CREAT);  if(shm\_id < 0) { printf("生成内存标识符失败!\n"); exit(0); }  mem\_addr = shmat(shm\_id,NULL,0);//进行内存映射，指向有数据的共享内存  if(mem\_addr==NULL) { printf("共享内存映射失败!\n"); exit(0); }  printf("key:%d id:%d addr:%x\n",key,shm\_id,mem\_addr);  printf("读取的数据:%s\n",mem\_addr);  // /\*5. 分离内存\*/  // shmdt(mem\_addr);  // /\*6. 删除内存段\*/  // shmctl(shm\_id,IPC\_RMID,NULL);  return 0;  } |

### 11.3:利用内存映射多进程拷贝文件

|  |
| --- |
| //fork多个进程来拷贝文件，将目标文件映射到内存中。加快拷贝速度  int main(int argc,char \*\*argv)  {  if(argc!=4)  {  printf("格式:./a.out <src\_file> <obj\_file> <进程个数> \n");  exit(0);  }  int s\_fd,o\_fd;  int course\_num,i;//进程个数  struct stat s\_info;  pid\_t pid;  char \*obj\_mem\_addr;//目标文件的内存映射首地址  //打开源文件和创建目标文件  s\_fd = open(argv[1],O\_RDWR);  if(s\_fd<0) {printf("打开源文件失败!\n");exit(0);}  o\_fd = open(argv[2],O\_RDWR | O\_CREAT,S\_IWUSR|S\_IRWXG|S\_IRWXO);  if(o\_fd<0) {printf("打开或者创建目标文件失败!\n"); exit(0);}  //因为映射目标文件到内存时，目标文件或许为空或者大小比源文件小而造成不能全部拷贝成功,所以要改变文件大小  stat(argv[1],&s\_info);//获取源文件大小  printf("s\_file\_size:%d\n",s\_info.st\_size);  ftruncate(o\_fd,s\_info.st\_size);//改变文件大小  obj\_mem\_addr = mmap(NULL,s\_info.st\_size,PROT\_READ|PROT\_WRITE,MAP\_SHARED,o\_fd,0);  if(obj\_mem\_addr == NULL) {printf("目标文件内存映射失败!\n");exit(0);}  //开始创建进程  course\_num = atoi(argv[3]);  int sun\_cpy\_byte = s\_info.st\_size / course\_num ;//计算每个子进程拷贝字节数  int far\_cpy\_byte = s\_info.st\_size % course\_num ;//计算父进程拷贝字节数  for(i=0;i<course\_num;i++)  {  pid = fork();  if(pid==0)break; //创建一个子进程，子进程就退出.保证父进程在创建进程  if(pid<0)  {  while(1)  if(wait()<0)break;  printf("fork失败!\n");  exit(0);  }  //sleep(1);  }  if(i<course\_num)//i小于进程个数时的进程是子进程  {  read(s\_fd,obj\_mem\_addr+i\*sun\_cpy\_byte,sun\_cpy\_byte);//往内存空间读取数据，目标文件内容也会改变  printf("子进程%d 拷贝成功%d Byte!\n",i,sun\_cpy\_byte);  }  else  {  read(s\_fd,obj\_mem\_addr+i\*sun\_cpy\_byte,far\_cpy\_byte);  printf("父进程 拷贝成功%d Byte!\n",i,far\_cpy\_byte);  while(1)  if(wait()<0)break;  printf("文件拷贝完成!\n");  }  } |

## 12.复制文件描述符状态(重定向的底层函数(dup))

#include <unistd.h>

int dup(int oldfd); //复制当前文件描述符，通过返回值返回

int dup2(int oldfd, int newfd);//将一个文件描述符状态复制给另一个描述符，

**如 dup2(pipefd[1],1); //将无名管道写端描述符状态给标准输出，那么该进程的所有的标准输出都输出到无名管道写端，可在读端读出来。(想要恢复标准输出，可用dup()提前备份)**

#define \_GNU\_SOURCE

#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);

### dup2的应用示例:

#### 示例1：

|  |
| --- |
| #include <stdio.h>  #include <sys/types.h>  #include <dirent.h>  #include <string.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/stat.h>  #include <sys/mman.h>  #include <fcntl.h>  int main(int argc,char \*\*argv)  {  if(argc!=2)  {  printf("参数格式:./app <新文件名称>\n");  return 0;  }  int fd;  fd=open(argv[1],O\_RDWR|O\_CREAT,S\_IWUSR|S\_IRUSR);  if(fd<0)  {  printf("创建文件失败.\n");  exit(0);  }  //int fd1=dup(1); //复制一个新的文件描述符  dup2(fd,1); //将fd的文件状态复制给1（标准输出）  printf("123\n");  printf("456\n");  close(fd);  return 0;  } |

#### 示例2：

|  |
| --- |
| #include <stdio.h>  #include <sys/types.h>  #include <dirent.h>  #include <string.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/stat.h>  #include <sys/mman.h>  #include <fcntl.h>  char buff[1024];  int main(int argc,char \*\*argv)  {  pid\_t pid;  int cnt;  int argfd[2];  pipe(argfd); //[0]读 [1]写  pid=fork();  if(pid==0)  {  dup2(argfd[1],1); //1表示输出文件描述符  //printf("12345-我是子进程打印的数据!\n");  execlp("ls","ls","-l",NULL);  }  else if(pid>0)  {  cnt=read(argfd[0],buff,1024);  buff[cnt]='\0';  printf("buff=%s\n",buff);  wait(NULL);  }  return 0;  } |

## 13.线程

### 13.1：线程基础

**#include <pthread.h>**

**int pthread\_create**

**(pthread\_t \*thread,const pthread\_attr\_t \*attr,void \*(\*start\_routine) (void \*),void \*arg);** //创建函数，参数1可得到线程id，类型为unsigned long; 2为设置属性，一般为NULL（有两种状态：默认结合状态，还有分离状态）；3为线程函数指针，填函数地址；4为3的形参，用不到填NULL

**Int pthread\_cancel(pthread\_t thread);** 杀死同一进程下的一个线程

**void pthread\_exit(void \*retval);** 线程退出函数，可填形参作为为返回值，如“abc”。可char \*p,用&p接收返回值

**int pthread\_join(pthread\_t thread, void \*\*retval);**  线程等待函数，带阻塞功能，等待不来线程结束信号，会一直卡死。参数2可接收形参返回值（唯一一个函数）。接收方法如上。

**int pthread\_detach(pthread\_t thread);**

//线程分离函数：该函数将默认的结合状态(joinable)变为分离状态(detached)，则该线程运行结束后会自动释放所有资源。用该函数想要还使线

//程无线循环,则主线程需要是死循环,和pthread\_join相同这两个函数都是处理正常结束的线程的资源。当线程异常结束时，则用到的是另一函数组(下面两个):

**void pthread\_cleanup\_push(void (\*routine)(void \*),void \*arg);** //注册清理函数,传入的函数用来处理一些善后工作

**void pthread\_cleanup\_pop(int execute);** //释放清理函数,形参填0和1两种,0不调用清理函数，1调用清理函数

//**触发调用清理函数的方法有三种:1--execute==1;2--调用pthread\_exit()函数;3--当被同进程下的另一线程杀死时:pthread\_cancel(线程id)**

#### 综合示例：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <pthread.h>  pthread\_t thread\_id1;//定义保存线程标识符的变量，类型为unsigned long 用lu打印数值  pthread\_t thread\_id2;  void routine\_1(void \*arg)//清理函数,无返回值  {  //free(),,close(),,,等等函数  printf("线程1已经清除完毕\n");  }  void \*start\_func1(void \*arg)//创建线程子函数实体1  {  pthread\_cleanup\_push(routine\_1,NULL);  printf("线程1!\n");  sleep(5);  //pthread\_exit("帅啊!");//线程退出函数,相当于进程的exit()函数,返回值形参可以填一个地址,通过pthread\_join接收  pthread\_exit(NULL);//如果下面调用自动回收资源的函数(detach和cleanup组)不支持接收返回值  pthread\_cleanup\_pop(0);//填0本身不会触发调用清理函数,但是中间有pthread\_exit()  }  void \*start\_func2(void \*arg)//创建线程子函数实体2  {  while(1)  {  printf("线程2!\n");  sleep(1);  //pthread\_cancel(thread\_id1);//也会触发线程1的清理函数  }  }  int main(int argc,char \*\*argv)  {  char \*p;  if(pthread\_create(&thread\_id1,NULL,start\_func1,NULL))  {  printf("创建线程1失败!\n");  exit(0);  }  //pthread\_detach(thread\_id1);  //线程分离函数该函数将默认的结合状态(joinable)变为分离状态(detached)，则该线程运行结束后会自动释放所有资源。用该函数想要还使线  //程无线循环,则主线程需要是死循环,和pthread\_join相同这两个函数都是处理正常结束的线程的资源。当线程异常结束时，则用到的是另一函  //数组:  //void pthread\_cleanup\_push(void (\*routine)(void \*),void \*arg); //注册清理函数,传入的函数用来处理一些善后工作  //void pthread\_cleanup\_pop(int execute); //释放清理函数,形参填0和1两种,0不调用清理函数，1调用清理函数  //触发调用清理函数的方法有三种:1--execute==1;2--调用pthread\_exit()函数;3--当被同进程下的另一线程杀死时:pthread\_cancel(线程id)  if(pthread\_create(&thread\_id2,NULL,start\_func2,NULL))  {  printf("创建线程2失败!\n");  exit(0);  }  //pthread\_detach(thread\_id2);//比较常用  //pthread\_join(thread\_id1,(void \*\*)&p);//不常用，因为在线程多时，手动处理麻烦，一般设置为自动收回线程占用的资源  //printf("线程1返回值:%s\n",p);  //pthread\_join(thread\_id2,NULL);//等待线程结束，等待不到结果会阻塞,第二个参数用来接收返回地址，是一个二维指针,不接收为NULL  //除了调用join函数,这个阻塞对于那些线程的实现是必不可少的  while(1)  {  }  return 0; |

### 13.2:线程通信机制(同步和互斥)

#### 13.2.1:互斥锁

互斥锁：适用在多个线程之间中只能同时运行一个能被或者能影响别的线程的情况

i**nt pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr);** 初始化锁,参数2为状态,填NULL默认快速锁

使用初始化函数为动态初始化互斥锁，静态初始化方式：**普通锁(快速锁)** **pthread\_mutex\_t fastmutex = PTHREAD\_MUTEX\_INITIALIZER; 嵌套锁（递归锁，可以多次上锁）: pthread\_mutex\_t recmutex = PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP(**需要在编译选项中指明**-D\_GNU\_SOURCE); 检错锁: pthread\_mutex\_t errchkmutex = PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP;**

**int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);** 销毁锁

**int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);** 上锁 上锁失败会阻塞，一直等待获取到锁再去上锁

**int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);** 尝试上锁，上锁失败返回错误值然后继续往下执行

**int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);**  解锁

**示例：**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <pthread.h>  //int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr); 初始化锁,参数2为状态,填NULL默认快速锁  //使用初始化函数为动态初始化互斥锁，静态初始化方式:快速静态锁 pthread\_mutex\_t fastmutex = PTHREAD\_MUTEX\_INITIALIZER;  //int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex); 销毁锁  //int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex); 上锁 上锁失败会阻塞，一直等待获取到锁再去上锁  //int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex); 尝试上锁，上锁失败返回错误值然后继续往下执行  //int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex); 解锁  pthread\_mutex\_t mutex; //定义锁变量  pthread\_barrier\_t barrier;//屏障机制变量，等待所有抢票线程创建完毕在进行统一抢票  pthread\_t thread\_id[100];//储存线程ID  int p\_num=20;//火车票数量  void \*create\_thread(void \*arg)  {  int i = \*(int \*)arg;  printf("线程%d--%lu准备就绪\n",i,pthread\_self());  pthread\_barrier\_wait(&barrier);  while(p\_num>0)  {  pthread\_mutex\_lock(&mutex);  printf("线程%d-%lu 抢到火车票!",i,pthread\_self());  p\_num--;  printf(" 剩余票数:%d\n",p\_num);  pthread\_mutex\_unlock(&mutex);  sleep(1);  }  }  int main(int argc,char \*\*argv)  {  if(argc!=2)  {  printf("格式:./a.out <线程个数> \n");  exit(0);  }  pthread\_mutex\_init(&mutex,NULL);  int i=0;  int data[100]={0};  int cnt=atoi(argv[1]);  pthread\_barrier\_init(&barrier,NULL,cnt);  for(i=0;i<cnt;i++)  {  data[i]=i;  pthread\_create(&thread\_id[i],NULL,create\_thread,&data[i]);  //sleep(1);  }  for(i=0;i<cnt;i++)  pthread\_join(thread\_id[i],NULL);  pthread\_mutex\_destroy(&mutex);  pthread\_barrier\_destroy(&barrier);  return 0;  } |

#### 13.2.2:读写锁

读写锁：适合在一个生产者和多个消费者关系的线程之间，如一个去写，多个线程需要去同时读数据.

读写锁有三个特性:当写上锁(获取锁—也就是请求运行线程)时，不解锁(释放锁)去读上锁会失败；当读上锁或者写上锁时，不解锁，去写上锁会失败；当读上锁时，不解锁再去读上锁不会失败；**也就是读上锁可以同时进行，读、写上锁，写、写上锁不可以同时进行**

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock, const pthread\_rwlockattr\_t \*restrict attr); //读写锁初始化----读写锁变量、读写锁状态

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock); // 读模式加锁

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock); //不阻塞

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock); //写模式加锁

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock); //解锁

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock); //销毁读写锁

#### 13.2.3:自旋锁

自旋锁:单纯的阻塞，不会引起任务调度，适合在小线程等待阻塞时，会大大减小CPU资源浪费。(保护消耗时间**很短的代码**)

int pthread\_spin\_destroy(pthread\_spinlock\_t \*); //自旋锁销毁

int pthread\_spin\_init(pthread\_spinlock\_t \*, int); //自旋锁初始化

int pthread\_spin\_lock(pthread\_spinlock\_t \*); //上锁

int pthread\_spin\_trylock(pthread\_spinlock\_t \*); //尝试上锁

int pthread\_spin\_unlock(pthread\_spinlock\_t \*); //解锁

#### 13.2.4:条件变量

条件变量:适合在控制线程运行时。使用条件变量时需要互斥锁的配合，即互斥锁要锁住条件变量等待函数

//初始化与销毁函数

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,const pthread\_condattr\_t \*restrict attr);

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

//条件变量唤醒函数

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond); //给全部线程运行的条件变量，当给出这个信号时，当前所有被wait()阻塞的线程全部开始运行

int pthread\_cond\_signal (pthread\_cond\_t \*cond); //给一个线程运行的条件变量信号，当给出这个信号时，按线程被阻塞顺序开始运行，调用一次此函数线程运行一个

//条件变量等待函数

int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex); //等待条件变量信号

**示例**:

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <string.h>  #include <signal.h>  #include <stdlib.h>  #define MAX\_CNT 10  pthread\_cond\_t cond;  pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER;  void \*thread\_func(void \*arg)  {  int i=\*(int\*)arg;  printf("第%d个线程启动.\n",i);  pthread\_mutex\_lock(&mutex);  //等待条件变量  pthread\_cond\_wait(&cond,&mutex);  //公共资源  printf("ID=%lu,i=%d\n",pthread\_self(),i);  pthread\_mutex\_unlock(&mutex);  }  //红绿灯类似  void sighandler(int sig)  {  pthread\_cond\_broadcast(&cond); //唤醒阻塞队列里的所有线程  //pthread\_cond\_signal(&cond); //唤醒一个线程  }  int main(int argc,char \*\*argv)  {  int i;  pthread\_t id[MAX\_CNT];  pthread\_cond\_init(&cond,NULL);  signal(2,sighandler);  for(i=0;i<MAX\_CNT;i++)  {  pthread\_create(&id[i],NULL,thread\_func,&i);  sleep(1); //先让上一个子线程先运行起来  }  for(i=0;i<MAX\_CNT;i++)  {  pthread\_join(id[i],NULL);  }  pthread\_cond\_destroy(&cond);  pthread\_mutex\_destroy(&mutex);  return 0;  } |

#### 13.2.5:屏障机制(围栏机制)

围栏机制:适用在当创建完全部线程，也需要同时启用线程时。和条件变量pthread\_cond\_broadcast()释放全部线程的效果类似，不过这个是当全部线程到达时，就**直接开始运行**，并且**运行先后无序**。**条件变量需要手动释放，且运行顺序按照线程阻塞顺序来.**

int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);

int pthread\_barrier\_init(pthread\_barrier\_t \*restrict barrier,const pthread\_barrierattr\_t \*restrict attr, unsigned count);

int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier); //等待函数，一般调用在线程一开始。

**示例见互斥锁示例!**

#### 13.2.6:信号量

信号量：信号量不同于锁机制，它的获取信号量相当于上锁(实质函数内定义的**变量在获取到后进行--1**)，释放信号量相当于解锁（在**释放后变量进行++1**）。当初始化定义变量值为1时，实现效果和互斥锁相同。（在不叠加的情况下—因为**获取和释放可以叠加**）

头文件不同，前五种机制使用<pthread.h>

#include <semaphore.h>

**int sem\_init(sem\_t \*sem, int pshared, unsigned int value);** //如果 pshared 的值为 0，那么信号量在进程的线程之间共享， 并且应位于所有线程可见的某个地址（例如，全局变量） 能够，或在堆上动态分配的变量），如果 pshared 不为零，那么信号量在进程之间共享，信号量的值就位于共享内存区域。Value为信号量的初始值

**int sem\_destroy(sem\_t \* sem);** //注销信号量

int sem\_post(sem\_t \* sem); //释放信号量，相当于解锁

int sem\_wait(sem\_t \* sem); //获取信号量，相当于加锁

int sem\_trywait(sem\_t \* sem); //不阻塞

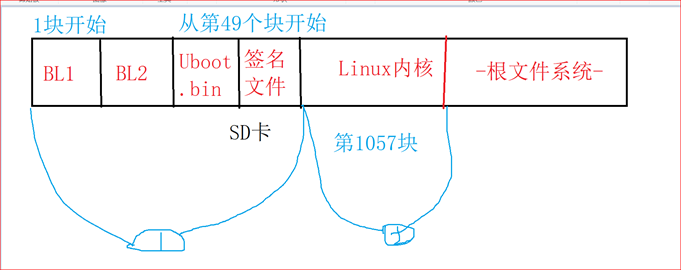
int sem\_getvalue(sem\_t \* sem, int \* sval); //获取信号量值

**示例**:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <pthread.h>  #include <semaphore.h>  //int sem\_init(sem\_t \*sem, int pshared, unsigned int value);  //int sem\_destroy(sem\_t \* sem);  //int sem\_post(sem\_t \* sem); //释放信号量，相当于解锁，可以叠加，++1  //int sem\_wait(sem\_t \* sem); //等待信号量,相当于加锁,可以叠加，相当于--1  //int sem\_trywait(sem\_t \* sem); //不阻塞  //int sem\_getvalue(sem\_t \* sem, int \* sval);//获取信号量的值  pthread\_mutex\_t mutex; //定义锁变量  pthread\_t thread\_id[100];//储存线程ID  int p\_num=20;//火车票数量  sem\_t sem;//设置信号量变量  void \*create\_thread(void \*arg)  {  int i = \*(int \*)arg;  printf("线程%d--%lu准备就绪\n",i,pthread\_self());  while(p\_num>0)  {  sem\_wait(&sem);//获取信号量，只获取不释放，那么全部阻塞，等待创建完毕后再全部一起释放  printf("线程%d-%lu 抢到火车票!",i,pthread\_self());  p\_num--;  printf(" 剩余票数:%d\n",p\_num);  sleep(1);  }  }  int main(int argc,char \*\*argv)  {  if(argc!=2)  {  printf("格式:./a.out <线程个数> \n");  exit(0);  }  int i=0;  int data[100]={0};  int cnt=atoi(argv[1]);  sem\_init(&sem,0,0);//信号量初始化，0，使得开始第一个线程就阻塞  for(i=0;i<cnt;i++)  {  data[i]=i;  pthread\_create(&thread\_id[i],NULL,create\_thread,&data[i]);  }  for(i=0;i<20;i++)//释放火车票一样多的次数  sem\_post(&sem);  for(i=0;i<cnt;i++)  pthread\_join(thread\_id[i],NULL);  return 0;  } |

## 14.驱动学习

**启动程序可以分为6部分**

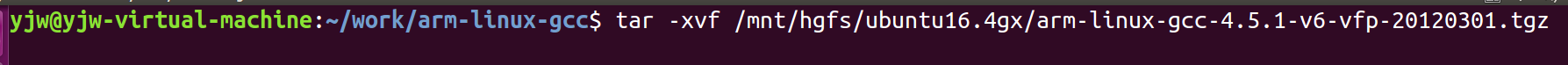


BL1为三星不开源的引导程序(8k)，BL2最大为14k，所有该部分为uboot引导程序的一部分，来做emmc的初始化、sd卡的初始化等。Uboot.bin为完整引导程序，BL2将比较大的引导程序拷贝在内存中允许，签名文件为验证信息。内核和根文件系统在引导程序运行时检测是否存在.

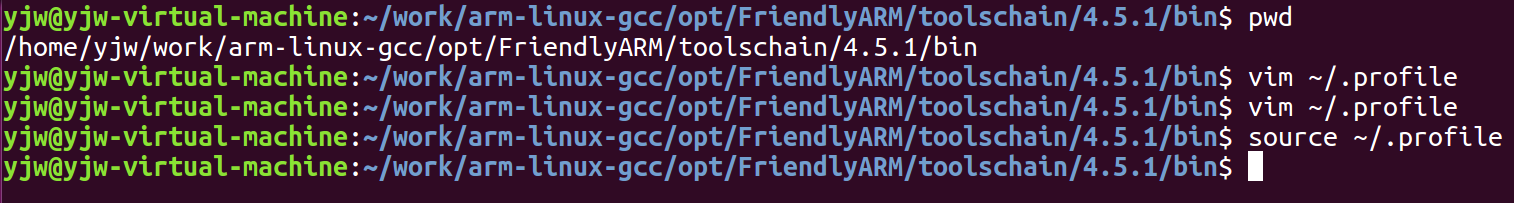
### 14.1 交叉编译器安装

**安装交叉编译器**

1. 解压编译器到工作目录(Ubuntu安装编译器会失败，因为编译器为32位的，Ubuntu为64为，所有要安装 sudo apt-get install lib32ncurses5)。



1. 将可执行文件目录添加到环境变量



文件.profile里面添加 （RedHat为~/.bash\_profile）

PATH="/home/yjw/work/arm-linux-gcc/opt/FriendlyARM/toolschain/4.5.1/bin:$PATH"

3.重启虚拟机

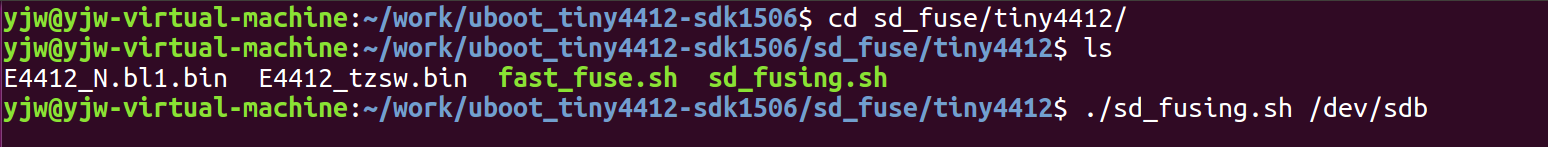
### 14.2 u-boot引导程序烧录

1. 解压uboot源码到工作目录

2. 执行4个指令: make distclean 、make clean 、make tiny4412\_config 、make

(boards.cfg文件存放当前Uboot支持的芯片型号)

1. 运行脚本向SD卡写uboot引导程序(一般插入的SD卡为sdb文件,自己查询)



烧录成功后，烧录的内容为**前四块内容**,运行fast\_fuse.sh是在烧录过BL1的情况下运行。

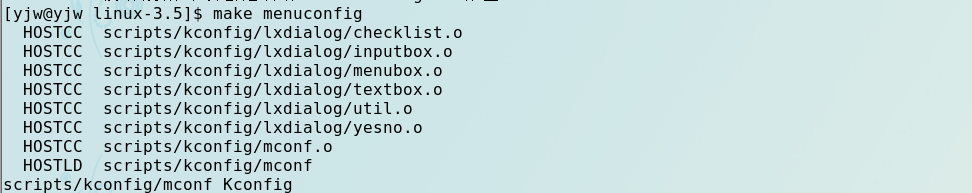
### 14.3 linux内核烧录

1. 解压linux系统内核到工作目录

2. 在内核目录下执行 cp tiny4412\_linux\_defconfig .config

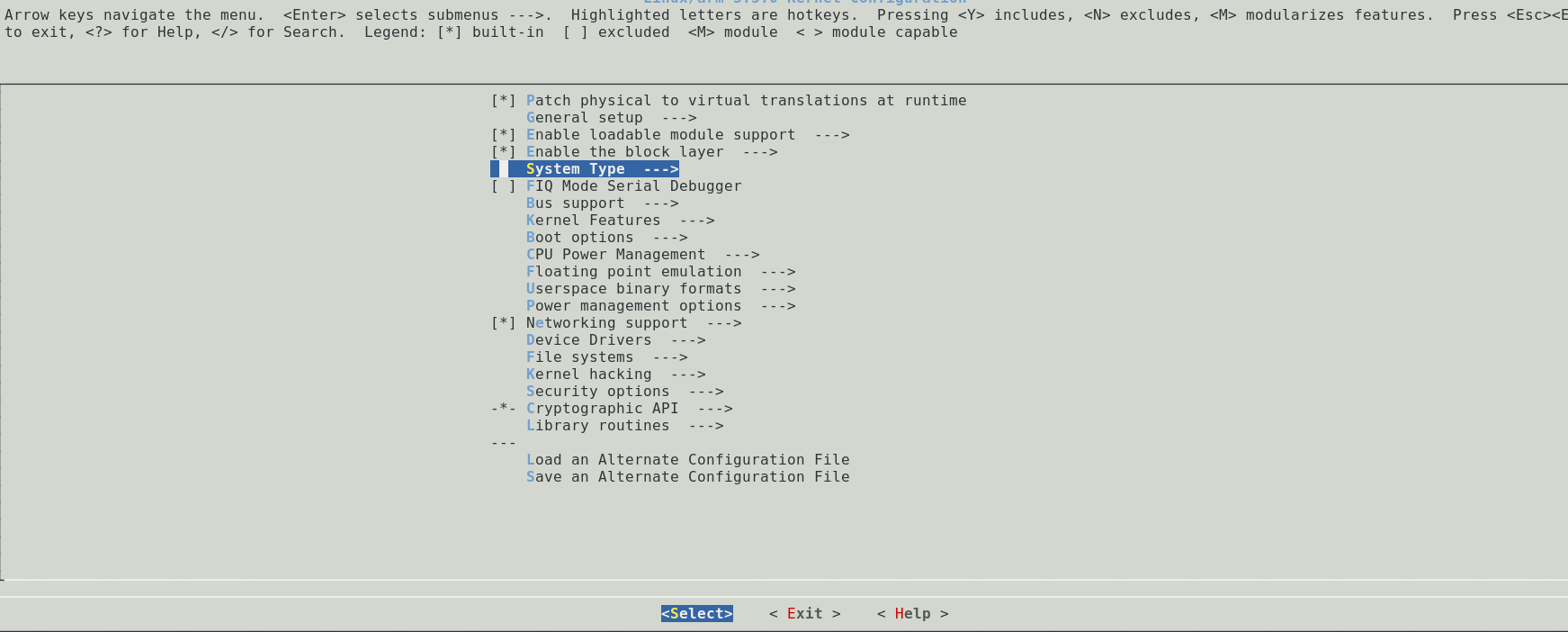


3. make menuconfig（执行指令前先调小字体）

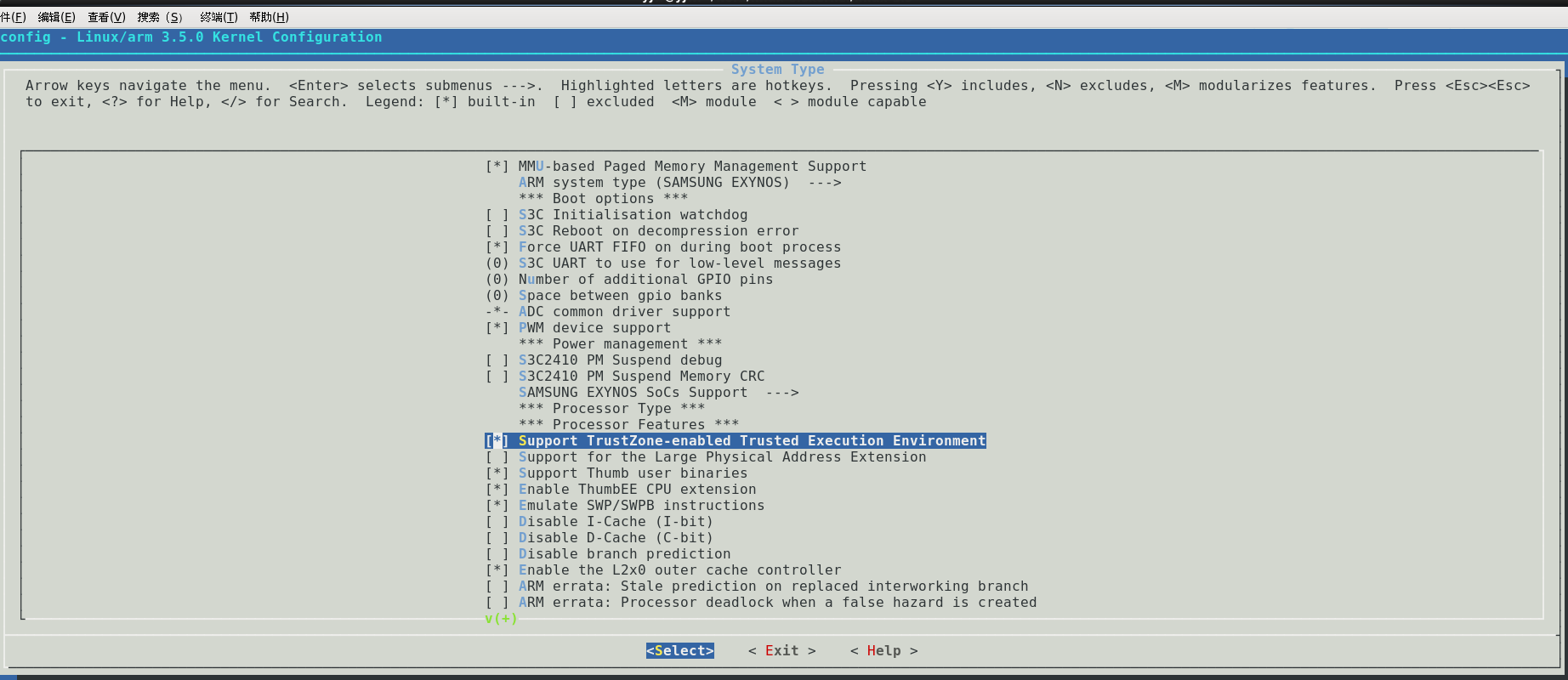


(ubuntu下make menuconfig错误解决办法: ddddd.安装缺的库 )

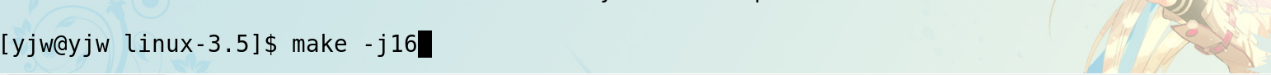
1. 选中系统类型



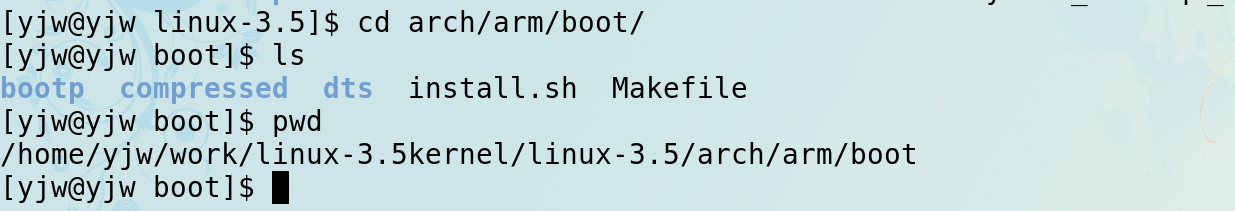
1. 选中按空格把\*去掉，退出并保存



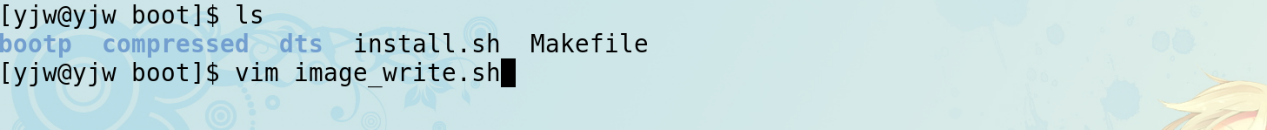
1. 编译



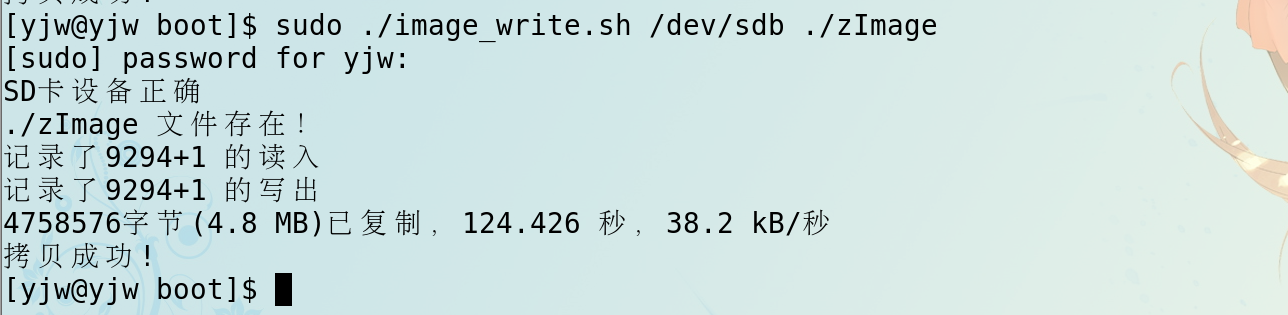
1. 启动目录



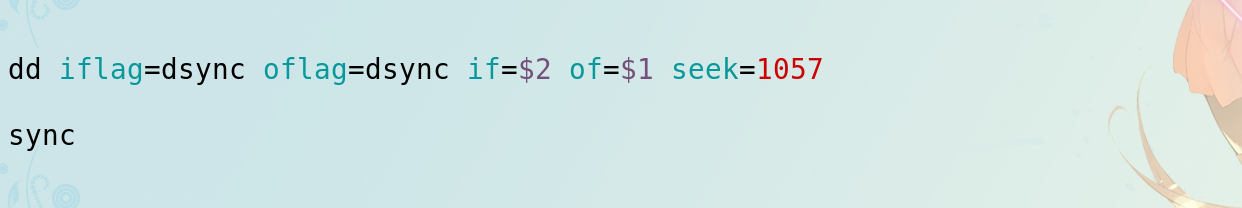
8. 创建并编写向SD写内核脚本



9. 向SD卡写入内核程序



10. 补充-脚本代码主要是这两句起作用，写这两行就绪



### 14.4 文件系统烧录

### 14.5 (uboot启动的情况下)将电脑的文件拷贝开发板DDR内存中的方法

1. 先在PClinux下将xxx.bin文件通过dd命令烧写到SD卡的指定扇区

再通过uboot的mmc read命令读取到DDR里。mmc 命令中的参数都是 16 进制表示，不是 10 进制表示…EMMC里存放BL1和BL2、签名、uboot的位置和SD卡里差一个扇区,其余的内核和文件系统还是一样.（SD卡）u-boot--49 ，0x290块；bl1—1 ，8k；bl2—17，14k ；签名文件---705，0x140块； 内核---1057，0x3000块 ，文件系统 环境变量单独放在1025，占32块

先清除EMMC内的程序。

emmc open 1 //从SD卡启动

mmc write 1 40000000 0 A000 //把DDR内存中的垃圾数据写到EMMC，让识别不出来是代码。清除大小A000为20M //可以不写

从DDR内存0x48000000的位置向’1’（EMMC）xxx的位置写xxx个块(512字节)。‘1‘是相对于从SD卡启动来说。从哪个设备启动哪个设备编号(device-num)就是0…在uboot内指令都是以16进制传参。

emmc open 1

mmc read 0 48000000 1 10 //读取10块数据到内存，每块占用地址512个，一个地址16位写入一个字节数据。

mmc write 1 48000000 0 10

mmc read 0 48000000 11 20

mmc write 1 48000000 10 20

mmc read 0 48000000 31 290

mmc write 1 48000000 30 290

mmc read 0 48000000 2c1 140

mmc write 1 48000000 2c0 140

mmc read 0 48000000 421 3000

mmc write 1 48000000 421 3000

2. 通过FAT文件系统直接拷贝。Uboot可以识别FAT文件系统。

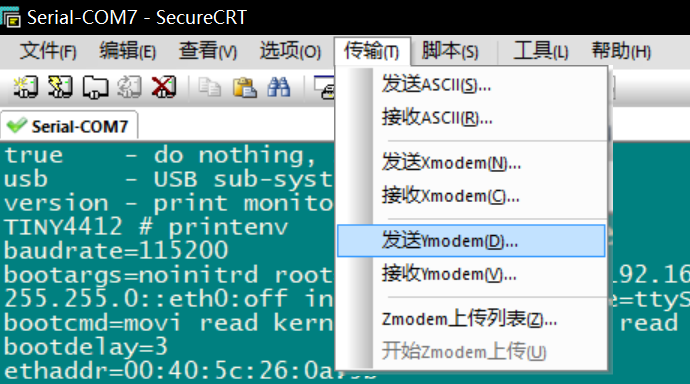
通过fatload命令加载。

**fatload <interface> <dev[:part]> <addr> <filename> [bytes] 尖括号必填项，方括号选填**

如：fatload mmc 0 40000000 zImage不指定大小就是整个文件传入。0就是0:1(**暂时不可用**)

3. 先执行loady <addr> 再通过串口软件(SecureCRT)的’传输’功能直接传输到DDR指定位置。

Loady 40000000



4. 通过ftp服务器网络下载。----没有网卡驱动—无法实现。

5. 通过uboot内部指令movi进行拷贝

**对设备读写操作，需要先打开，读写完毕，再关闭。**

如：emmc close 1 打开设备 1

emmc open 1 关闭设备 1

movi read fwbl1 0 40000000; //从 SD(设备编号为)拷贝 bl1 到 DDR 内存地址

emmc open 1; //打开 EMMC 设备

movi write zero fwbl1 1 40000000; //将 DDR 地址处数据写入到 EMMC 对应位置

emmc close 1; //关闭 EMMC 设备

movi read bl2 0 40000000; //从 SD(设备编号为 0)拷贝 bl2 到 DDR 内存地址

emmc open 1; //打开 EMMC 设备

movi write zero bl2 1 40000000; //将 DDR 地址处数据写入到 EMMC 对应位置

emmc close 1; //关闭 EMMC 设备

总共6部分，其余的方式以此类推

或者精简程序:

movi r f 0 40008000;emmc open 1;movi w z f 1 40008000;emmc close 1;  
movi r b 0 40008000;emmc open 1;movi w z b 1 40008000;emmc close 1;  
movi r u 0 40008000;emmc open 1;movi w z u 1 40008000;emmc close 1;  
movi r t 0 40008000;emmc open 1;movi w z t 1 40008000;emmc close 1;  
movi r k 0 40008000;movi w k 1 40008000;

### 14.6 Linux内核及驱动

#### 1．寻找某个命令的实现方法

1---先打开source insight工程，搜索有无和命令相同的文件.比如cmd\_xxxxx.c

2---找不到就换个思路根据函数的提示语句，来进行搜索整个项目里面的内容。

最好将提示语句分开搜索，以免有的语句是用%S打印出来.

3---根据找到的地方，再看相关调用的函数，再进行跳转，一步一步深入。

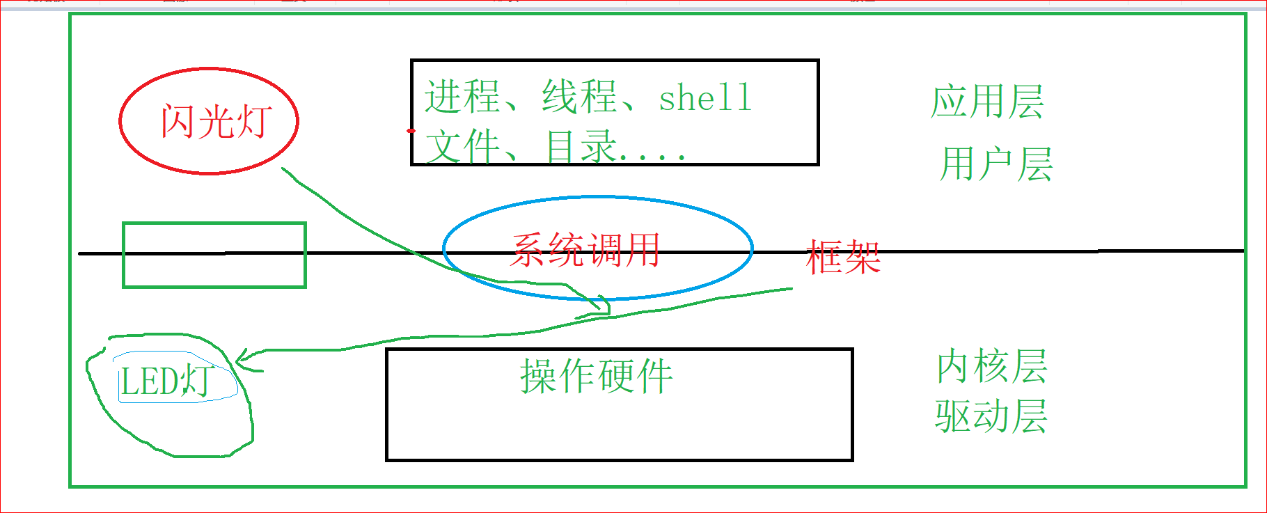
4---根据找到的最底层函数来分析

#### 2. /dev下设备介绍

在Linux下3种设备**框架**:

1. 字符设备 : 键盘、鼠标、触摸屏、…….
2. 块设备: U盘、SD卡、…
3. 网络设备: 有线网卡、无线网卡…

USB驱动框架-----通用框架



内核管理驱动: 设备号。

设备号: 主设备号+次设备号组成。共32位 宏:MKDEV(ma,mi)实现设备号的合成

主设备号: 表示设备类型。(鼠标、LCD、)，高12位，MAJOR(dev) 得到主设备号

次设备号: 表示同类设备的具体设备编号。低20位, MINOR(dev) 得到次设备号

**一个创建设备节点的函数：**

|  |
| --- |
| mknod console c 5 1 |

mknod <设备节点名称> <设备类型> <主设备号> <次设备号>

设备类型: c 表示字符设备 b表示块设备

#### 3．字符设备

**有3种字符设备注册方式：**

1. 杂项设备注册方式

杂项设备注册的特点:

1. 注册函数和注销函数就2个

2. 主设备号固定为10

3. 次设备号范围: 0~255 255时表示自动分配次设备号

4. 自动在/dev下创建设备节点

1. 早期设备注册方式—linux2.6内核之前使用的注册方式,现在使用的是linux3.5内核。

早期设备注册特点:

1. 注册注销函数2个

2. 主设备号范围: 0~255

3. 当注册一个主设备号之后，就占用了所有的次设备号

4. 不会自动创建设备节点

1. 标准设备注册方式

杂项设备是使用早期设备注册

早期设备是使用**标准设备**注册

(**杂项设备就是一个早期设备注册方式使用的一个典范**)

**杂项设备调用实现原理**:

注册过程:

misc\_register杂项设备注册函数(如果选择次设备255,那么注册前内核先自动分配一个没有被占用的次设备号)根据次设备号调用device\_create设备创建函数在/dev目录下创建一个设备节点，再调用list\_add向杂项设备链表添加该设备信息。

注销过程：

删除/dev下的设备节点，再删除杂项设备链表中的设备信息。(这些注册、销毁函数，创建、删除节点函数，添加、删除链表信息等等的函数都往往成对出现)

调用过程：

应用层函数(如open打开设备文件)----》调用早期设备文件方式的注册函数中创建杂项设备的操作函数集合(中的open函数)-----》改变杂项设备的文件操作集合指向，指向新添加的次设备号的文件操作集合再调用对应设备文件的操作函数集合(的open函数)。因为改变了文件集合指向，所有再调用open、write等就是调用次设备的文件集合

(所以说杂项设备类是在早期设备注册方式上注册，而杂项设备某个设备又是基于杂项设备注册方式)

编写的驱动如果要随时在内核中运行如红外解码，那么就不用写设备注册函数，直接设置IO口，安装驱动即可。如果需要在代码中打开并调用驱动节点，那么只需加上注册和注销函数和文件操作集合即可，但要保证两个驱动都安装时，IO口不会冲突

### 14.7 开发板挂载nfs文件系统服务器

**如果开发板有完整的环境，包括内核和文件系统，那么可以之间在命令行设置ip**

**Ifconfig eth0 192.168.1.201**

**再进行挂载.要进行解锁，否则会卡死,还要使处于同一网段下**

**Mount -t nfs -o nolock <ip>:<共享目录> <挂载目录>**

**如mount -t nfs -o nolock 192.168.1.200:/home/yjw/rootfs /mnt/**

1. **搭建NFS文件共享服务器步骤**

**编辑/etc/exports 文件，示例: /work/ \*(rw,no\_root\_squash,sync)**

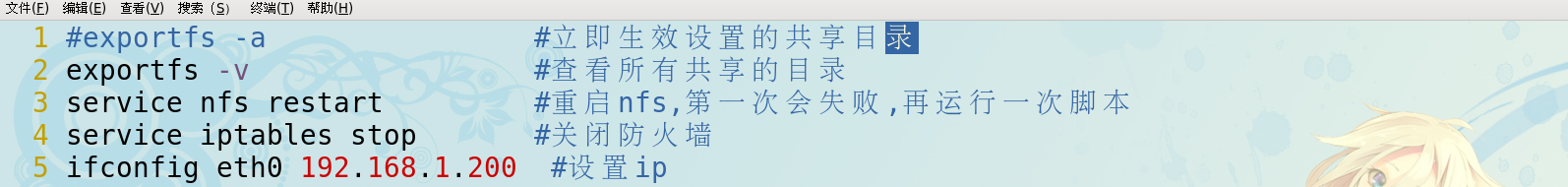
|  |
| --- |
| /home/yjw/rootfs \*(rw,no\_root\_squash,sync)  路径格式：  实际路径 \*(rw,no\_root\_squash,sync)  实际路径：要共享的目录  \* ：允许所有的网段访问  rw ：读写权限  sync：资料同步写入硬盘  no\_root\_squash： nfs 客户端共享目录使用者权限 |

1. **NFS服务器相关命令**

|  |
| --- |
| [wbyq@wbyq project]$sudo exportfs -a //目录修改立即生效  [wbyq@wbyq project]$sudo exportfs -v //输出当前系统共享的目录  [wbyq@wbyq project]$ sudo service nfs start //开启NFS服务器  [wbyq@wbyq project]$ sudo service nfs stop //停止NFS服务器 [确定]  [wbyq@wbyq project]$ sudo service nfs restart //重启NFS服务器 |

1. **关闭防火墙(sudo setup)，并手动去左上角设置添加一个ip**

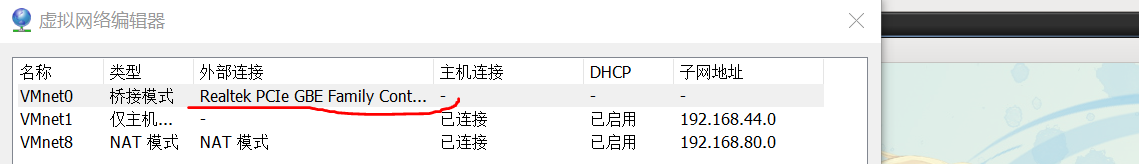
可以编写一个**脚本启动**



启动后可**自测试**是否设置成功,执行：sodu mount -t nfs 192.168.1.200:/home/yjw/rootfs ./mnt

成功后，取消挂载sudo umount ./mnt

1. **网线连接开发板和电脑，挂载文件夹.要先设置虚拟机桥接到电脑以太网卡**.



**4.1 编译制作根文件系统**

工具箱下载地址： <https://busybox.net/>

下载BusyBox 1.23.2 稳定版本.

然后解压到工作目录，再执行 make menuconfig 配置编译器和安装位置

|  |
| --- |
| [wbyq@wbyq busybox-1.23.2]$ make menuconfig  Busybox Settings --->  Build Options --->  (**arm-linux-**) Cross Compiler prefix // **该条信息下回车，填arm-linux-**  Busybox Settings --->  Installation Options ("make install" behavior) --->  (/home/yjw/rootfs) BusyBox installation prefix //该条信息下回车，退格删除默认路径，填自定义路径 |

设置完成保存退出再进行编译安装

**4.2完善根目录系统**

1. **创建文件夹**：mkdir lib dev etc/init.d home proc sys root opt tmp var -p

2. **拷贝动态库**

|  |
| --- |
| [wbyq@wbyq lib]$ pwd  /home/wbyq/rootfs/lib  [wbyq@wbyq lib]$  cp /home/wbyq/work/arm-linux-gcc/opt/FriendlyARM/toolschain/4.5.1/arm-none-linux-gnueabi/lib/\*.so\* ./  [wbyq@wbyq lib]$  cp /home/wbyq/work/arm-linux-gcc/opt/FriendlyARM/toolschain/4.5.1/arm-none-linux-gnueabi/sys-root/usr/lib/\*.so\* ./ |

拷贝完成用tree命令看，总共146个文件

3. **创建设备文件**

|  |
| --- |
| [wbyq@wbyq rootfs]$ cd dev/  [wbyq@wbyq dev]$ sudo mknod console c 5 1  [wbyq@wbyq dev]$ sudo mknod null c 1 3  [wbyq@wbyq dev]$ pwd  /home/wbyq/rootfs/dev |

4. **拷贝fstab文件，完善ETC**

|  |
| --- |
| sudo cp /etc/fstab etc/ |

5. **拷贝inittab文件**

cp busybox解压目录/examples/inittab rootfs/etc/

|  |
| --- |
| [wbyq@wbyq etc]$ pwd  /home/wbyq/rootfs/etc  [wbyq@wbyq etc]$ cp /home/wbyq/work/busybox/busybox-1.23.2/examples/inittab ./  [wbyq@wbyq etc]$ ls  fstab init.d inittab |

6. **修改inittab文件**。删除原有全部内容，粘贴下面的内容

|  |
| --- |
| ::sysinit:/etc/init.d/rcS //系统初始化阶段**执行**的文件: /etc/init.d/**rcS (必须要可执行权限)**  console::askfirst:-/bin/sh //指定控制台的脚本解释器  #askfirst表示需要按下回车才可以进入系统 respawn表示开机直接进入系统  ::ctrlaltdel:/sbin/reboot //系统重启命令  ::shutdown:/bin/umount -a -r //关机时执行的操作: 取消挂载的操作 |

7. **创建etc/init.d/rcS文件,写入配置代码**

|  |
| --- |
| [wbyq@wbyq rootfs]$ touch etc/init.d/rcS  [wbyq@wbyq rootfs]$ cat etc/init.d/rcS  mount -a  mkdir /dev/pts  mount -t devpts devpts /dev/pts  echo /sbin/mdev > /proc/sys/kernel/hotplug  mdev -s #根据已经安装的驱动自动生成设备节点文件命令。  /bin/hostname wbyq #终端的名称--可以不加 |

1. **修改etc/init.d/rcS权限为可执行。**

|  |
| --- |
| [wbyq@wbyq rootfs]$ chmod 777 etc/init.d/rcS |

9. **创建 etc/profile 文件**

|  |
| --- |
| [wbyq@wbyq rootfs]$ touch etc/profile  [wbyq@wbyq rootfs]$ cat etc/profile  USER="id-un"  LOGNAME=$USER  PS1='[\u@\h \W]\#'  PATH=$PATH  HOSTNAME='/bin/hostname'  export USER LOGNAME PS1 PATH |

10. **拷贝分组信息和密码文件到etc目录下**

|  |
| --- |
| [wbyq@wbyq rootfs]$ sudo cp /etc/passwd etc/  [wbyq@wbyq rootfs]$ sudo cp /etc/group etc/ |

11. **设置开发板环境变量**

开发板选择从EMMC启动，重启开发板，3-2-1之前按下回车。

(1). **设置环境变量 如:**

|  |
| --- |
| setenv bootargs root=/dev/nfs nfsroot=192.168.1.200:/home/yjw/rootfs ip=192.168.1.201:192.168.1.200:192.168.1.1:255.255.255.0::eth0:off init=/linuxrc console=ttySAC0 lcd=S702 |

(2). **保存环境变量**

(3). **重启开发板**

|  |
| --- |
| setenv bootargs  root=/dev/nfs nfsroot=<NFS服务器IP地址>  :<NFS共享的目录>  ip=<开发板的IP地址>:<网关>:<子网掩码>::eth0:off init=/linuxrc console=ttySAC0 lcd=S702 |

(4) **等待读秒完成**，引导内核后进入linux系统内

### 14.8 挂载UbuntuNFS文件服务器

参照 ubuntu系统下根文件系统制作与挂载.Pdf (系统编程文件夹下)

### 14.9 开发板挂载本地文件系统

#### 本地挂载方法一:

**如果开发板有完整的环境，包括内核和文件系统，那么可以之间在命令行设置ip**

**Ifconfig eth0 192.168.1.201**

**再进行挂载.要进行解锁，否则会卡死,还要使处于同一网段下**

**Mount -t nfs -o nolock <ip>:<共享目录> <挂载目录>**

**如mount -t nfs -o nolock 192.168.1.200:/home/yjw/rootfs /mnt/**

**将文件系统放入SD/EMMC某个分盘，然后设置bootargs环境变量来启动文件系统**

1. **使用busybox生成文件系统然后完善操作方法和14.7前面一致.**

2. 将文件系统打包并放入文件系统下(rootfs)

3. (如果EMMC有完整的环境除了文件系统，请跳至第5步.) 将BL1、BL2、Uboot、内核烧至SD卡，然后设置环境变量：

setenv bootargs root=/dev/nfs nfsroot=192.168.1.200:/home/yjw/rootfs ip=192.168.1.201:192.168.1.200:192.168.1.1:255.255.255.0::eth0:off init=/linuxrc console=ttySAC0 lcd=S702

4. 保存再使用命令将所有包括环境变量复制到EMMC

1. 从EMMC启动开发板。进入Uboot命令行，对EMMC用fdisk命令进行分区。分区后对所有盘初始化为ext3 或者ext2 格式，
2. 现在bootargs环境变量的值应该和上面的一样，那么就重启，进入Linux命令行，此时挂载在NFS服务器。然后使用mount命令将EMMC任意分盘挂载在 mnt/目录下。
3. 解压rootfs.tar解压到挂载目录,并将解压后的mnt/下的rootfs/下所有内容搬移到mnt/下。然后取消挂载重启进入uboot命令行.
4. 设置环境变量bootargs改变启动方式，如: set bootargs root=/dev/mmcblk0p2 rootfstype=ext3 init=/linuxrc console=ttySAC0,115200 mmcblk0p2为刚才挂载的那个盘，ext3为该盘的文件格式

#### 本地挂载方法二:

**使用厂家工具进行把所有信息从SD烧写到EMMC**

1. 将友善之臂光盘资料B盘中images的FriendlyARM.ini文件和Superboot4412.bin和linux目录下的ramdisk-u.img文件复制到SD卡下创建的images下。
2. 解压F:\系统编程\光驱资料\A\tools\Linux\_tools.tgz到linux的工作目录下.
3. 删除要打包镜像的文件系统中dev/下的所有文件 :sudo rm /home/yjw/rootfs/dev\* -rf
4. 使用解压出来的bin目录下的make\_ext4fs创建文件系统镜像.使用方法如 ：

sudo ./work/linux-tools/usr/local/bin/make\_ext4fs -s -l 1024M rootfs.img rootfs 倒数第三项为指定生成的镜像空间，rootfs.img为生成的镜像名，rootfs为使用的文件系统

1. 将生成的文件移动到SD卡的images下.
2. 重新编译内核，将14.3节内核烧写中步骤重复执行，不执行去掉的星号一步，为了将签名文件也生成在内核中。然后将重新生成的zImage移动到SD卡的images下.
3. 修改FriendlyARM.ini文件。将启动安卓内核改为启动Linux内核。并将Linux栏下的内容改为如下格式:

Linux-BootLoader = Superboot4412.bin #引导程序

Linux-Kernel = zImage #修改路径

Linux-CommandLine = root=/dev/mmcblk0p1 rootfstype=ext4 console=ttySAC0,115200 init=/linuxrc ctp=2 skipcali=y ethmac=1C:6F:65:34:51:7E coherent\_pool=2M lcd=s702 #不修改

Linux-RamDisk = ramdisk-u.img #修改路径

Linux-RootFs-InstallImage = rootfs.img #修改路径及文件系统镜像名

1. 将SD卡插入开发板，设置从SD卡启动，等待出现OS “Linux” XXXX一行后，改为从EMMC启动，然后重启。完成。

### 14.10 外部中断、定时器、工作队列

#### 1.外部中断:

**请求中断号**函数：Irq = gpio\_to\_irq(gpio); //请求中断号,填入gpio返回分配的irq外部中断号

**请求中断**函数：int request\_irq(unsigned int irq,irq\_handler\_t handler, unsigned long flags,const char \*name,void \*dev\_id); //中断号,中断服务函数，中断触发方式、中断名、往中断服务传入数据的参数

**关闭中断**函数：disable\_irqirq);

**释放中断**函数：free\_irq(key[i].irq,&key[i]);

#### 2.定时器:

struct timer\_list timer; //定时器结构体需要关心的几个参数

**定时器初始化**:

**timer.expires**=jiffies+msecs\_to\_jiffies(3000); //装载值

**timer.function** = timer\_function; //定时器中断服务函数

**timer.data** = 200; //向中断服务函数传参

**init\_timer**(&timer); //初始化

**装载值，表示定时多久,赋值格式 jiffies+HZ\*5 (HZ当前内核默认为200，而HZ可以表示为1秒,\*5意思是定时5s)或者: jiffies+msecs\_to\_jiffies(5000) :毫秒转换为节拍数函数 jiffies为时间基准**

void (\*function)(unsigned long); //定时器中断服务函数

unsigned long data;//给服务函数传参\*/

**/\*启动定时器:添加定时器到定时器队列\*/**

**add\_timer**(&timer);

/\*关闭、注销定时器\*/

**del\_timer**(&timer);

//重新设置装载值 当达到装载值进入中断后，如果还要继续产生中断，那么用修改函数重新设定定时

**mod\_timer**(&timer,jiffies+msecs\_to\_jiffies(3000));

#### 3.工作队列

在内核中，中断服务内代码不宜过于复杂，而且完全**不能写延时函数**，因为**中断由内核接管处理**，如果使用延时函数那么会导致进程休眠，那么此时处理中断的进程正是内核，就**会造成内核崩溃**

那么处理中断就延申出了**中断顶半部和中断底半部**。顶半部处理时间短的事情，底半部处理时间长，还可以加延时函数。而底半部就是由**工作队列**来实现。

struct work\_struct { //工作队列结构体

atomic\_long\_t data;

struct list\_head entry;

**work\_func\_t func; /\*** 工作函数指针 **\*/**

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif  
}

**/\*工作初始化\*/**

**INIT\_WORK**(&work,work\_func);

**/\*添加一个工作到工作队列 主要放在服中断务函数内\*/**

**schedule\_work**(&work);

**/\*销毁工作\*/**

**cancel\_work\_sync**(&work);

### 等待队列、poll、select、epoll、异步IO(signal)

#### 等待队列

**等待队列适合处理在单输入时**，（一个文件设备节点的读写操作）

在没有等待队列之前，当应用层在向内核读取数据或者其他操作时，有时不会接收到数据。例如在等待读取按键键值时，

while(1)

{

Ioctl(fd,READ,&data);

Printf(“%d\n”,data);

}

当没有按键按下时，应用层也会向内核发起读取数据的请求，内核也会给应用层发送一些没有用的数据，那么就会一直浪费CPU的资源。所有引用等待队列可以大大节省CPU的资源 在 Linux 中, 一个等待队列由一个"等待队列头"来管理, 一个 wait\_queue\_head\_t 类型的结构, 定义在<linux/wait.h>中. 一个等待队列头可被定义和初始化。

静态定义:

DECLARE\_WAIT\_QUEUE\_HEAD(name); (预编译时宏被替代为：wait\_queue\_head\_t name = \_\_WAIT\_QUEUE\_HEAD\_INITIALIZER(name))

动态定义:

wait\_queue\_head\_t my\_queue;  
init\_waitqueue\_head(&my\_queue);

休眠函数:

wait\_event(wq, condition) //休眠不可中断，在没唤醒之前，所有信号都会阻塞，等在休眠被唤醒之后信号才会被执行。如在休眠时Ctrl+c不会结束进程，在执行唤醒函数之后，进程会立马结束

wait\_event\_interruptible(wq, condition) //休眠可中断，

wait\_event\_timeout(wq, condition, timeout) //（可以指定时间）（不可中断）

wait\_event\_interruptible\_timeout(wq, condition, timeout) //（可以设置超时时间）（可中断）

**唤醒函数:**

wake\_up(x)//中断非中断类型的休眠都可以唤醒，x为要唤醒的队列头指针，&wait

wake\_up\_interruptible(x) //唤醒可中断类型

wake\_up\_interruptible\_all(x) //唤醒所有可中断类型的休眠

wake\_up\_all(x) //唤醒所有休眠

wake\_up\_nr(x, nr) //唤醒指定个数的所有类型休眠

一般休眠函数放在向应用层传数据前，在所有数据获得后的最好一项前唤醒。然后根据

#### poll

当出现多输入情况下等待队列不能够满足要求，当打开了多个设备文件时，每个设备文件都有一套文件操作函数集合，都有中断服务函数和等待队列，而应用层代码在轮询时，是一个一个设备进行轮询，当上一个设备文件在等待唤醒，那么整个轮询都会休眠进行等待上一个设备唤醒完毕，所以引入了poll来解决这个问题

While(1)

{

Ioctl(fd1,READ,&data);//等待唤醒休眠。。。。(卡住)

Ioctl(fd2,READ,&data);//等待上个设备完成操作

Ioctl(fd3,READ,&data);

}

使用：

|  |
| --- |
| #include <stdio.h>  #include <sys/ioctl.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <poll.h>  #define READ\_KEY\_VALUE 1234  #define KEY\_NUM 4  int data;  int fd[KEY\_NUM];  char \*dev\_name[KEY\_NUM]={"/dev/wait\_key\_irq\_dev1","/dev/wait\_key\_irq\_dev2","/dev/wait\_key\_irq\_dev3","/dev/wait\_key\_irq\_dev4"};  // struct pollfd {  // int fd; /\* 文件描述符\*/  // short events; /\* 事件的类型POLLIN：事件或优先级可读 \*/  // short revents; /\* 事件的返回值,内核和编写的驱动赋值 \*/  // };  struct pollfd fds[KEY\_NUM];  int main(int argc,char \*\*argv)  {  int i=0;  for(i=0;i<KEY\_NUM;i++)  {  fd[i]=open(dev\_name[i],O\_RDWR);  if(fd[i]<0)  {  printf("%s 设备文件打开失败!\n",dev\_name[i]);  return 0;  }  fds[i].fd = fd[i];  fds[i].events = POLLIN;  }  //int poll(struct pollfd \*fds, nfds\_t nfds, int timeout); //返回成功事件的个数  int cnt = 0;  while(1)  {  cnt = poll(fds,KEY\_NUM,0);  if(cnt) //如果有外部中断事件产生  {  for(i=0;i<KEY\_NUM;i++)//遍历判断是哪个描述符对应的事件  {  if(fds[i].events & fds[i].revents)  {  ioctl(fd[i],READ\_KEY\_VALUE,&data);//获取按键值  printf("0x%x\n",data);  }  }  }  }    close(fd);  return 0;  } |

驱动层接口代码:

|  |
| --- |
| static unsigned int key\_poll(struct file \*file, struct poll\_table\_struct \*poll)  {  unsigned int mask = 0;  /\*添加进程到等待队列头\*/  poll\_wait(file,&wait,poll);  /\*判断按键是否按下\*/  if(condition)  {  condition=0;  mask|=POLLIN;  }  return mask;  } |

#### 3.select------poll的升级版

Select最多可以管理1024个文件描述符,在内核驱动中，select和poll的接口完全没有变化，还是poll。

int **select**(int nfds, fd\_set \*readfds, fd\_set \*writefds,fd\_set \*exceptfds, struct timeval \*timeout);

需要注意的是第一个参数填入的是所有文件描述符中最大的那个并+1.第2、3、4参数分别填入的是读变化类型的事件、写变化类型和其他变化类型。第5是超时时间，填NULL则等待事件集合中有变化才停止阻塞，并返回发生变化的事件个数，填0则立即返回，有变化则返回变化个数，没有返回0，填>0则是设置超时时间，超过设置的时间则返回，返回值和上面相同.

void **FD\_CLR**(int fd, fd\_set \*set); //将 fd 从 set 结构中清除  
int **FD\_ISSET**(int fd, fd\_set \*set); //检测事件，如果 FD 状态发生变化返回真  
void **FD\_SET**(int fd, fd\_set \*set); //添加文件描述符，可以重复添加多个！  
void **FD\_ZERO**(fd\_set \*set); //每次循环都要清空集合，否则不能检测描述符变化

使用:

|  |
| --- |
| #include <stdio.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <sys/ioctl.h>  #include <poll.h>  #include <sys/select.h>  #include <sys/time.h>  #include <sys/types.h>  #include <unistd.h>  #define GET\_KEY\_VALUE 0x12345  char \*DEV\_NAME[]={"/dev/key\_dev1","/dev/key\_dev2","/dev/key\_dev3","/dev/key\_dev4"};  int fd[4];  int data;  fd\_set readfds; //定义文件操作集合  int max\_fd=0; //保存最大的文件描述符  int event\_cnt; //发生事件的数量  int main(int argc,char \*\*argv)  {  int i;  /\*1. 打开4个设备文件\*/  for(i=0;i<4;i++)  {  fd[i]=open(DEV\_NAME[i],O\_RDWR);  if(fd[i]<0)  {  printf("%s 设备文件打开失败!\n",DEV\_NAME[i]);  return 0;  }  if(fd[i]>max\_fd)max\_fd=fd[i];//得到最大的文件描述符  }  while(1)  {  /\*1. 添加文件描述符到操作集合\*/  for(i=0;i<4;i++)  {  FD\_SET(fd[i],&readfds);  }  /\*2. 进行事件监控\*/  event\_cnt=select(max\_fd+1,&readfds,NULL,NULL,NULL);  if(event\_cnt)  {  for(i=0;i<4;i++)  {  //判断对应的文件描述符是否产生了读事件  if(FD\_ISSET(fd[i],&readfds))  {  ioctl(fd[i],GET\_KEY\_VALUE,&data);  printf("select\_data=0x%X\n",data);  }  }  }  /\*清空文件操作集合中所有文件描述符\*/  FD\_ZERO(&readfds);  }  for(i=0;i<4;i++)  {  close(fd[i]);  }  return 0;  } |

#### epoll

同样是poll的升级版，理论可以管理无限个事件，只要内存空间够大，Linux默认每个线程空间10M。接口同样不变.

等待事件:

int **epoll\_wait**(int epfd, struct epoll\_event \*events,int maxevents, int timeout);

epfd :由 epoll\_create 生成的 epoll 专用的文件描述符；  
epoll\_event :用于回传已经检测成功的事件数组；  
maxevents :每次监控的最大事件个数；  
Timeout （单位是毫秒） :-1 相当于永久阻塞， 0 相当于非阻塞。一般用-1 即可

返回值:事件变化个数

int **epoll\_create**(int size);

该函数生成一个 epoll 专用的文件描述符。它其实是在内核申请一空间，用来存放你监控的描述符 fd 上是否产生了什么事件。 size 就是你在这个 epoll fd 上能存放的最大文件描述符 fd 个数。只要有空间，随便定

注册epoll事件:

int **epoll\_ctl**(int epfd, int op, int fd, struct epoll\_event \*event);

函数功能：该函数用于设置某个 epoll 文件描述符上的事件，可以注册事件，批改事件，删除事件。  
参数：  
epfd：由 epoll\_create 生成的 epoll 专用的文件描述符；  
op：要进行的事件操作，可能的取值 EPOLL\_CTL\_ADD 注册、 EPOLL\_CTL\_MOD 修 改、 EPOLL\_CTL\_DEL删除。  
事件类型定义如下：  
#define EPOLL\_CTL\_ADD 1 //注册  
#define EPOLL\_CTL\_DEL 2 //删除  
#define EPOLL\_CTL\_MOD 3 //修改  
fd：将要监控的文件描述符；  
event：指向 epoll\_event 的指针；  
返回值：若是调用成功返回 0，不成功返回-1

使用:

|  |
| --- |
| #include <stdio.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <sys/ioctl.h>  #include <sys/epoll.h>  char \*DEV\_NAME[]={"/dev/key\_dev1","/dev/key\_dev2","/dev/key\_dev3","/dev/key\_dev4"};  int fd[4],data;  int event\_cnt; //发生事件的数量  int nfds,epollfd;  #define MAX\_EVENTS 4  struct epoll\_event ev[MAX\_EVENTS],events[MAX\_EVENTS];  #define GET\_KEY\_VALUE 0x12345  int main(int argc,char \*\*argv)  {  int i;  /\*1. 打开4个设备文件\*/  for(i=0;i<4;i++)  {  fd[i]=open(DEV\_NAME[i],O\_RDWR);  if(fd[i]<0)  {  printf("%s 设备文件打开失败!\n",DEV\_NAME[i]);  return 0;  }  }  /\*2. 创建存放文件描述符的空间\*/  epollfd=epoll\_create(4);  /\*3. 初始化事件监听结构体并注册监听的事件\*/  for(i=0;i<4;i++)  {  ev[i].events = EPOLLIN;  ev[i].data.fd = fd[i];  epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,fd[i],&ev[i]);  }  while(1)  {  /\*4. 监听事件: 等待事件发生\*/  nfds=epoll\_wait(epollfd,events,MAX\_EVENTS,-1);  for(i=0;i<nfds;i++)  {  ioctl(events[i].data.fd,GET\_KEY\_VALUE,&data);  printf("epoll\_data=0x%X\n",data);  }  }  for(i=0;i<4;i++)  {  close(fd[i]);  }  return 0;  } |

5.异步IO----驱动向应用层发送信号

例如当内核驱动代码产生中断事件时，可以向应用层代码发送一个信号，发送信号函数(kill\_fasync(&async,SIGIO,POLL\_IN);)只能发送内核中file结构体中struct fown\_struct f\_owner结构体内的信号类型,参数2、3没有用处，什么时候使用:选择在适当的适合发送信号到应用层。可通过应用层fcntl函数修改文件描述符的信息

异步中断IO的接口是

static **struct fasync\_struct** \*async;

static int key\_**fasync**(int fd, struct file \*filp, int on)

{

return f**async\_helper**(fd, filp, on, &async);

}

应用层使用：

|  |
| --- |
| #include <stdio.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <sys/ioctl.h>  #include <sys/epoll.h>  #include <signal.h>  #define DEV\_NAME "/dev/key\_dev"  int fd;  int data;  #define GET\_KEY\_VALUE 0x12345  //信号处理函数  void sighandler(int sig)  {  printf("sig=%d\n",sig);  ioctl(fd,GET\_KEY\_VALUE,&data);  printf("signal\_data=0x%X\n",data);  }  int main(int argc,char \*\*argv)  {  int state\_flag;  /\*1. 打开设备文件\*/  fd=open(DEV\_NAME,O\_RDWR);  if(fd<0)  {  printf("%s 设备文件打开失败!\n",DEV\_NAME);  return 0;  }  /\*2. 绑定要捕获信号\*/  signal(SIGIO,sighandler);  /\*3. 给文件描述符设置进程的PID\*/  fcntl(fd,F\_SETOWN,getpid());  /\*4. 设置当前文件描述符支持异步IO属性\*/  state\_flag=fcntl(fd,F\_GETFL);  fcntl(fd,F\_SETFL,state\_flag|O\_ASYNC);  while(1)  {  /\*.....\*/  }  close(fd);  return 0;  } |

### 14.12. 寄存器操作API

物理地址转换虚拟地址: 初始化设备时使用

|  |
| --- |
| void \_\_iomem \*ioremap (unsigned long phys\_addr, unsigned long size)  \_\_iomem \*: 返回虚拟地址首地址  unsigned long phys\_addr: 物理地址，查询手册得到的地址  unsigned long size：转换长度，一般为一个寄存器的长度，4个字节 |

取消虚拟地址映射：卸载初始化(退出函数)时使用

|  |
| --- |
| Void iounmap (volatile void \_\_iomem \*addr);  volatile void \_\_iomem \*addr: 填入初始化时生成的虚拟地址首地址 |

### 14.13. 内核的同步与互斥

1. 原子操作

2.信号量

3.互斥锁

4.自旋锁

1. 读写锁
2. 顺序锁