系统编程

系统编程课程的内容是研究Linux操作系统的各项系统调用，通过学些这些系统调用，熟悉Linux操作系统应用层程序的开发，同时对操作系统的工作原理有一定的了解

一.安排

1.库的制作和使用

2.Linux进程相关

3.Linux进程间通信

4.Linux多线程开发

5.线程的互斥和同步

二.参考书籍

<<Unix环境高级编程>>

<<Linux内核设计与实现>>

三.内容特点

本门课程内容涉及到大量的系统函数，对于这些函数来说不要死记硬背，要理解性地记忆

四.库的制作和使用

1.什么是库，为什么需要库

库是二进制形式的目标模块包

a.o \

b.o - abc.a -------- 库

c.o /

使用库很方便对程序进行模块化设计，方便对某些固定功能的重复使用，方便对某些功能进行更新

2.库的类型

（1）静态库

扩展名.a 库中封装的二进制代码，在链接阶段被复制到调用模块中，调用模块运行时不再需要静态库的支持

（2）动态库

扩展名.so 库中封装二进制代码，在链接阶段并不会复制到调用模块，被嵌入到调用模块中仅仅是被调用函数在共享库中的相对地址

调用模块在运行时需要动态库

动态库和静态库相比，使用更加灵活，使用更为常见

3.库的制作和使用

（1）静态库的制作

a.编辑源程序

xxx.h xxx.c

b.编译成目标文件

gcc -c xxx.c ---->xxx.o

c.打包目标文件

ar -r libxxx.a xxx.o yyy.o....

注：ar命令

格式： ar [选项] 静态库名 目标文件列表

-r:将目标文件插入到静态库，若已存在就更新

-q:将目标文件追加到静态库的末尾

-d:从静态库中删除指定的目标文件

-t:列表显示静态库中的目标文件

-x:将静态库展开为目标文件

（2）静态库的使用

a.直接调用法

gcc main.c libmath.a

b.参数调用法

gcc main.c -lmath -L. // -l库名 -L库所在路径

c:设置环境变量法

LIBRARY\_PATH环境变量记录了链接是库的寻找路径

export LIBRARY\_PATH=$LIBRARY\_PATH:.

gcc main.c -lmath

为了是环境变量设置持久化，可以将设置语句写入 ~/.bashrc

（3）动态库的制作

a.编辑源程序

xxx.h xxx.c

b.编译成目标模块

gcc -fpic -c xxx.c -o xxx.o

-fPIC:大模式,代码量较大,速度较慢，支持所有平台

-fpic：小模式，代码量小，速度快，只支持一部分平台，比如Linux

c.链接成共享库

gcc -shared xxx.o .... -o libxxx.so

（4）动态库的使用

a.静态加载

gcc main.c libxxx.so

gcc main.c -lmath -L.

export LIBRARY\_PATH=$LIBRARY\_PATH:.

gcc main.c -lmath

程序使用了动态库，在运行的时候需要动态库的支持，需要将动态库的路径添加到环境变量LD\_LIBRARY\_PATH

b.动态加载

静态加载是在程序运行之前就记录好了函数在共享库中的位置，在运行程序是读取动态库的内容到内存，按照之前记录的位置去访问

动态加载是在程序运行期间加载和卸载动态库，可以根据用户的需要在需要的时候加载动态库，不需要的时候卸载，进一步提高了灵活性和效率

Linux中的dl库实现库的动态加载

加载共享库 ---- dlopen

参数：

filename:给路径按照路径加载，给文件名就根据LD\_LIBRARY\_PATH环境变量去查找

flag:加载标志

RTLD\_LAZY:延迟加载，什么时候真正使用的时候才加载

RTLD\_NOW:立即加载

返回值：

成功返回动态库的句柄,失败返回NULL

获取动态库的的函数地址 ------- dlsym

参数：

handle:动态库的句柄

symbol:函数名

返回值：

成功返回函数地址，失败返回NULL

卸载动态库 -------------dlclose

参数：

handle:动态库的句柄

获取出错信息 --------- dlerror

返回错误字符串

一.进程

1.Linux是一个多任务的操作系统

进程就是一种任务，任务参与任务调度，在内核中一个任务用task\_struct结构体表示

task\_struct{

unsigned long state;

unsigned long policy;

int prio;

struct task\_struct \*parent;

struct list\_head tasks;

pid\_t pid;

}

2.进程相关命令

ps命令 -------- 查看进程信息

-elf:查看所有进程

-aux：查看所有进程

pstree ----------- 查看进程树

kill ---------- 给进程发信号

kill -9 进程号

killall ---------- 给进程发信号

killall -信号 进程名

3.进程的相关信息

（1）父进程和子进程

如果进程a创建了进程b，那么a就是b的父进程,b就是a的子进程。其中1号进程(init)在系统启动时被内核创建，其他进程都是直接或者间接由他产生

除了1号进程以外，所有的进程都有父进程，如果一个进程的父进程结束，这种进程叫孤儿进程，子进程就会认1号进程作为父进程

（2）进程号(PID)

每个进程用进程号作为唯一标识,PID由系统管理,PID是一个正整数,进程结束后PID可以重复使用,但是需要延迟重用

Linux提供了相关函数来获取进程号

getpid函数返回当前进程PID

getppid返回父进程的PID

4.进程的创建

（1）fork()

fork函数通过复制父进程创建一个子进程，子进程会复制父进程的进程空间，除了代码段(只读)共享以外，其他内存区域都会复制一份(写拷贝)

fork函数创建子进程之后，父子进程同时运行，先后顺序不确定，谁先结束也不确定

fork函数调用一次，返回2次，父子进程返回后都从fork函数的下一条语句开始运行，父进程返回子进程ID，子进程返回0

fork函数复制父进程产生子进程时，复制文件描述符，但是不复制文件表

fork函数复制父进程产生子进程时，输入输出缓冲区会复制

fork函数复制父进程产生子进程时，不复制文件锁

当进程结束的时候，会给父进程发送一个信号，让父进程来回收资源，如果子进程结束后没有发送父进程信号，或者父进程没有处理该信号，子进程变成僵尸进程

vfork()

vfork的形式和fork完全一样，区别在于vfork不复制父进程任何资源，vfork创建的子进程抢占父进程的资源,导致父进程阻塞,父进程解除阻塞的条件是：

子进程结束，归还父进程资源

使用其他函数(exec)替换子进程的内容，归还父进程的资源

注:子进程应该用exit函数退出

5.父进程等待子进程结束(回收资源)

wait系列函数可以让父进程等待子进程结束，回收子进程资源

wait函数让父进程等待任意一个子进程结束,返回子进程的PID，把子进程的退出状态和退出码放入status参数中

如果没有子进程结束，wait函数就会阻塞父进程,直到有子进程结束为止，wait函数也可以回收僵尸子进程 殓尸工

WIFEXITED(status)

返回真表示子进程正常退出

WEXITSTATUS(status)

返回子进程的退出码

WIFSIGNALED(status)

返回真表示子进程被信号打断

练习：

使用多进程，子进程获取传入文件的大小，父进程获取文件的大小并且将文件的大小写入a.dat文件中

waitpid函数可以设置等待的方式和等待的子进程

参数：

pid: >0:等待ID为pid的子进程

-1:等待所有子进程

0:等待本组子进程

<-1:等待进程组为pid绝对值的子进程

status:用法同wait

options: 0:阻塞等待

WNOHANG:不等待

返回值：

成功返回子进程ID,使用不阻塞回收二没有收到子进程，返回0，失败返回-1

6.进程的退出

进程的退出分为正常退出和非正常退出

正常退出：

主函数返回

执行exit函数

\_exit()/\_Exit()函数

所有线程都结束

异常退出：

进程被信号打断(ctrl+c kill -9)

最后一个线程被取消

退出处理函数：

可以在程序中使用atexit()函数登记一个函数作为退出处理函数，该函数会在程序退出时被调用

退出处理函数可以登记多个，保存在退出处理函数栈中(先登记后执行，后登记先执行)

（1）exit函数和\_exit函数的区别

a.exit函数退出时会清空缓冲区，\_exit直接退出进程

b.exit函数退出时会调用退出处理函数,\_exit直接退出

7.exec函数簇

exec函数簇都是用来替换进程的内容，exec系列函数并不会产生新的进程

l：list -------- 程序的参数通过list传递(一个个传递)

v:vector ------- 程序的参数通过向量传递(数组)

p:path ------- 通过PATH环境变量查找程序文件

e:environment ------- 传入环境变量数组

返回值：

正常本函数不返回，失败返回-1

（1）exec函数说明

fork函数是用于创建一个子进程，该子进程几乎是父进程的副本，而有时我们希望子进程去执行另外的程序，exec函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换了。另外，这里的可执行文件既可以是二进制文件，也可以是Linux下任何可执行脚本文件。

（2）在Linux中使用exec函数族主要有以下两种情况

a. 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用任何exec 函数族让自己重生。

b. 如果一个进程想执行另一个程序，那么它就可以调用fork函数新建一个进程，然后调用任何一个exec函数使子进程重生。

（3）exec函数族语法

实际上，在Linux中并没有exec函数，而是有6个以exec开头的函数族，下表列举了exec函数族的6个成员函数的语法。

所需头文件： #include <unistd.h>

函数说明： 执行文件

函数原型：

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

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

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

int execve(const char \*path, char \*const argv[], char \*const envp[])

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

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

函数返回值：成功 -> 函数不会返回，出错 -> 返回-1，失败原因记录在error中。

这6 个函数在函数名和使用语法的规则上都有细微的区别，下面就可执行文件查找方式、参数表传递方式及环境变量这几个方面进行比较说明。

①    查找方式：上表其中前4个函数的查找方式都是完整的文件目录路径，而最后2个函数（也就是以p结尾的两个函数）可以只给出文件名，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。

②    参数传递方式：exec函数族的参数传递有两种方式，一种是逐个列举的方式，而另一种则是将所有参数整体构造成指针数组进行传递。

在这里参数传递方式是以函数名的第5位字母来区分的，字母为“l”（list）的表示逐个列举的方式，字母为“v”（vertor）的表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当做参数传给它，数组中的最后一个指针要求是NULL。读者可以观察execl、execle、execlp的语法与execv、execve、execvp的区别。

③    环境变量：exec函数族使用了系统默认的环境变量，也可以传入指定的环境变量。这里以“e”（environment）结尾的两个函数execle、execve就可以在envp[]中指定当前进程所使用的环境变量替换掉该进程继承的所以环境变量。

（4）PATH环境变量说明

PATH环境变量包含了一张目录表，系统通过PATH环境变量定义的路径搜索执行码，PATH环境变量定义时目录之间需用用“:”分隔，以“.”号表示结束。PATH环境变量定义在用户的.profile或.bash\_profile中，下面是PATH环境变量定义的样例，此PATH变量指定在“/bin”、“/usr/bin”和当前目录三个目录进行搜索执行码。

PATH=/bin:/usr/bin:.

export $PATH

（5）进程中的环境变量说明

    在Linux中，Shell进程是所有执行码的父进程。当一个执行码执行时，Shell进程会fork子进程然后调用exec函数去执行执行码。Shell进程堆栈中存放着该用户下的所有环境变量，使用execl、execv、execlp、execvp函数使执行码重生时，Shell进程会将所有环境变量复制给生成的新进程；而使用execle、execve时新进程不继承任何Shell进程的环境变量，而由envp[]数组自行设置环境变量。

（6）exec函数族关系

第4位统一为：exec

第5位

l：参数传递为逐个列举方式

execl、execle、execlp

v：参数传递为构造指针数组方式

execv、execve、execvp

第6位

e：可传递新进程环境变量

execle、execve

p：可执行文件查找方式为文件名

execlp、execvp

事实上，这6个函数中真正的系统调用只有execve，其他5个都是库函数，它们最终都会调用execve这个系统调用，调用关系如下图12-11所示：

char \*const ps\_argv[] ={"ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL};

char \*const ps\_envp[] ={"PATH=/bin:/usr/bin", "TERM=console", NULL};

execl("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);

execv("/bin/ps", ps\_argv);

execle("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL, ps\_envp);

execve("/bin/ps", ps\_argv, ps\_envp);

execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);

execvp("ps", ps\_argv);

请注意exec函数族形参展开时的前两个参数，第一个参数是带路径的执行码（execlp、execvp函数第一个参数是无路径的，系统会根据PATH自动查找然后合成带路径的执行码），第二个是不带路径的执行码，执行码可以是二进制执行码和Shell脚本。

（8）exec函数族使用注意点

在使用exec函数族时，一定要加上错误判断语句。因为exec很容易执行失败，其中最常见的原因有：

①    找不到文件或路径，此时errno被设置为ENOENT。

②    数组argv和envp忘记用NULL结束，此时errno被设置为EFAULT。

③    没有对应可执行文件的运行权限，此时errno被设置为EACCES。

（9）exec后新进程保持原进程以下特征

      环境变量（使用了execle、execve函数则不继承环境变量）；

      进程ID和父进程ID；

      实际用户ID和实际组ID；

      附加组ID；

      进程组ID；

      会话ID；

      控制终端；

      当前工作目录；

      根目录；

      文件权限屏蔽字；

      文件锁；

      进程信号屏蔽；

      未决信号；

      资源限制；

      tms\_utime、tms\_stime、tms\_cutime以及tms\_ustime值。

对打开文件的处理与每个描述符的exec关闭标志值有关，进程中每个文件描述符有一个exec关闭标志（FD\_CLOEXEC），若此标志设置，则在执行exec时关闭该描述符，否则该描述符仍打开。除非特地用fcntl设置了该标志，否则系统的默认操作是在exec后仍保持这种描述符打开，利用这一点可以实现I/O重定向。

（10）execlp函数举例

execlp.c源代码如下:

#include <stdio.h>

#include <unistd.h>

int main()

{

if(fork()==0){

if(execlp("/usr/bin/env","env",NULL)<0)

{

perror("execlp error!");

return -1 ;

}

}

return 0 ;

}

编译 gcc execlp.c –o execlp。

执行 ./execlp，执行结果如下：

HOME=/home/test

DB2DB=test

SHELL=/bin/bash

……

由执行结果看出，execlp函数使执行码重生时继承了Shell进程的所有环境变量，其他三个不以e结尾的函数同理。

（11）execle函数举例

利用函数execle，将环境变量添加到新建的子进程中去。

execle.c源代码如下：

#include <unistd.h>

#include <stdio.h>

int main()

{

/\*命令参数列表，必须以 NULL 结尾\*/

char \*envp[]={"PATH=/tmp","USER=sun",NULL};

if(fork()==0){

/\*调用 execle 函数，注意这里也要指出 env 的完整路径\*/

if(execle("/usr/bin/env","env",NULL,envp)<0)

{

perror("execle error!");

return -1 ;

}

}

return 0 ;

}

编译:gcc execle.c –o execle。执行./execle，执行结果如下：

PATH=/tmp

USER=sun

作业：

1.使用exec系列函数启动madplay播放音乐， 同时用文件记录开始播放时间

sudo apt-get update

sudo apt-get install madplay

2.编写程序，产生10个子进程，根据用户输入，选择性回收其中一个子进程的资源，得到退出码

一.进程间通信(IPC)

每个进程都有自己独立的进程空间，进程与进程之间不能直接访问对方的数据。但是在某些时候进程之间需要进行数据交互，实现进程间数据交互的机制就叫进程间通信

1.实现进程间通信的方式

文件

管道(pipe)

信号(signal)

标准IPC

共享内存

消息队列

信号量集

套接字(socket)

2.管道

管道是最古老的IPC，目前使用较少

管道分为无名管道和有名管道

无名管道只能用于有亲缘关系(父子,兄弟)之间的进程间通信

有名管道可以用于所有进程间的通信

管道本身只是作为交互的媒介，不存储任何数据，只有在同时有读端和写端才畅通，否则阻塞(管道文件不能使用open创建)

（1）无名管道

无名管道在文件系统中不可见，通过pipe函数创建

#include <unistd.h>

int pipe(int fd[2])

参数：

无名管道的读端和写端

返回值：

成功返回0，失败返回-1

两个进程同时获得读写描述符，但是在通信时应该选择其中一个

一段写，关闭读描述符，另一端读，关闭写描述符

pipe函数应该在父进程中调用，子进程进程父进程的描述符

管道读端断开，写端会收到SIGPIPE信号，导致写端进程结束

写端断开，读端返回0

练习：使用无名管道实现两个子进程之间的通信

（2）有名管道

有名管道在文件系统中可见，本身不占用磁盘空间,可用于所有进程之间的通信

mkfifo命令/函数都可以创建管道文件

用法：

命令 ------ mkfifo 路径

函数：

#include <sys/types.h>

#include <sys/stat.h>

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

参数：

pathname：路径

mode:权限

返回值：

成功返回0，失败返回-1

对有名管道文件的访问通过文件IO接口(open read write close),不能使用lseek系列函数

3.信号

（1）概念

信号是一种异步通信方式

信号本质上是一个非负的整数，在Linux系统一般是64个，中间不保证连续

每个信号都有一个宏名,编程的是偶尽量使用宏名而不使用数字，不同的系统中，一个宏名对应的值可能不同。宏名以SIG开头

SIGINT ------ 信号2的宏名

kill -l：查看系统所有的信号

（2）信号的分类

信号分为可靠信号和不可靠信号，1-31是不可靠信号，34-64是可靠信号

不可靠信号不支持排队，因此如果有多个相同的不可靠信号同时到来，可能出现信号丢失，可靠信号支持排队，不会有信号丢失

（3）信号处理

信号的处理方式有三种

a.默认处理 ------- 系统对每个信号都有默认处理方式，大多数信号的默认处理方式都是退出进程

b.忽略信号 ------- 不做任何处理，就像没有收到信号一样

c.捕获信号(自定义处理) --------- 信号处理改为执行我们自定义的函数

注：信号9(SIGKILL)和信号19(SIGSTOP)既不能被忽略，也不能被捕获，也不能屏蔽

Linux提供了设置信号处理的函数signal()/sigaction()

#include <signal.h>

signal(int handler, void (\*)(int));

第一个参数是我们要捕获的信号类型，第二个参数是我们要绑定的信号处理函数。

参数：

signnum : 设置的信号

handler: 设置信号的处理方式

SIG\_DFL ：默认处理

SIG\_IGN ：忽略信号

函数指针 : 捕获信号

返回值：

成功返回之前的信号处理方式,失败返回SIG\_ERR

注:对信号处理的设置一般写在程序的开头

signal函数的使用方法

a.写一个信号处理函数 void fa(int){...}

b.调用signal函数注册处理函数2. sigaction

#include <signal.h>

int sigaction(int, const struct sigaction \* \_\_restrict, struct sigaction \* \_\_orestrict);

相比较signal来说，sigaction与它的功能是类似的，但是更全面。

先说一下参数吧，第一个参数同signal，后两个参数类型是一个名为sigaction的结构体，该结构体表示对信号的处理方式，\_\_restrict为新设定的处理方式，\_\_orestrict参数则用于获取之前的信号处理方式。

我们来看一下这个结构体的内容：

#include <signal.h>

struct sigaction {

union \_\_sigaction\_u \_\_sigaction\_u; /\* signal handler \*/

sigset\_t sa\_mask; /\* signal mask to apply \*/

int sa\_flags; /\* see signal options below \*/

};

union \_\_sigaction\_u {

void (\*\_\_sa\_handler)(int);

void (\*\_\_sa\_sigaction)(int, siginfo\_t \*,

void \*);

};

可以看到该结构体主要由三部分构成，

sa\_mask，一个信号集

sa\_flags, 设置程序收到信号时的行为

signaction\_u 表示信号处理函数

sa\_mask和sa\_flags都是用二进制位的形式来保存信息的，前者代表着当前进程所能接收到的信号集合，后者代表信号处理的设置信息。

sa\_flags的值有好多，这里我们只说一个常用的SA\_RESTART，书上的解释是【重新调用被该信号终止的系统调用】，最开始我没有理解，写了个小例子尝试后，清晰了许多。

一起看看吧

void action(int a)

{

printf("%d -=-=\n", a);

}

int main()

{

int pid = fork();

if(pid > 0)

{

struct sigaction s;

memset(&s, 0, sizeof(s));

s.sa\_handler = action;

sigfillset(&s.sa\_mask);

//s.sa\_flags |= SA\_RESTART;

assert(sigaction(SIGCHLD, &s, NULL) != 1);

}

else if(pid == 0)

{

sleep(2);

exit(0);

}

char str[10];

scanf("%s", str);

printf("%s\n", str);

puts("End");

return 0;

}

上面这段代码是对SIGCHLD信号(当子进程状态发生变化时产生)进行了信号处理的设定。

当子进程执行exit后，会给父进程发送SIGCHLD信号，而父进程本来此时正在进行scanf函数，等待用户输入，在接受到SIGCHLD信号之后，中断目前正在进行的系统调用，然后去执行信号处理函数，也就是上述代码中设定的action函数。在执行完action之后，程序会继续往下执行，也就是下面的结果

练习：

修改有名管道的代码，在用户按下ctrl+c(SIGINT)时退出程序，删除管道文件

（4）信号发送

a.用键盘发送信号

ctrl+c ------ 信号2 SIGINT

ctrl+\ ------ 信号3 SIGQUIT

ctrl+z ------ 信号20 SIGTSTP

b.硬件故障/程序出错

段错误

总线错误

除以0

c.kill命令发送信号

kill -信号 进程PID

killall -信号 进程名 ----------- 给所有进程名符合的进程发信号

如果killall不指定发送的信号，默认发送SIGTERM信号

d.信号发送函数

kill() raise() alarm() sigqueue()

kill函数

参数：

pid:要发送信号的进程

sig:要发送的信号

返回值：

成功返回0，失败返回-1

raise函数

给本进程发信号

alarm函数

给本进程在指定的秒数后发送SIGALRM信号

该函数会取消之前存在的闹钟,返回之前闹钟剩余的秒数

sigqueue函数

功能类似于kill函数，区别在于发送信号是可以多带一个数据

该数据是一个联合类型union sigval

（5）子进程对父进程信号处理的继承

fork和vfork创建子进程，子进程会继承父进程的信号处理方式(父进程默认的，子进程默认，父进程忽略，子进程忽略，父进程捕获，子进程捕获)

fork/vfork + exec创建子进程替换进程内容，父进程默认的，子进程默认，父进程忽略的，子进程忽略，父进程捕获的，子进程变为默认

（6）信号集

信号集是信号集合，可以保存多个信号,类型是sigset\_t，本质上是一个超大整数

信号集是一种数据结构，除了存储信号以外，还要提供了一组接口函数来操作信号集

sigaddset() ---------- 增加一个信号

sigdelset() ---------- 删除一个信号

sigemptyset() ------ 删除全部信号

sigfillset() -------- 填满全部信号

sigismember() ------- 查询有没有某个信号

（7）信号屏蔽

信号不确定什么时候回来,因此可能在非常重要的场合(关键代码)信号到来，如果此时中断当前的操作而去处理信号，可能导致非常严重的错误。

进程本身不能阻止信号的到来，，但是可以屏蔽信号,就是信号到来之后暂时不去处理，等待关键代码执行完毕，接触信号屏蔽再做处理

信号屏蔽的实现

sigprocmask函数 -------- 设置信号屏蔽

参数：

how:屏蔽的方式

SIG\_BLOCK - 相当于旧的屏蔽+新的屏蔽

A B C + C D E = A B C D E

SIG\_UNBLOCK - 相当于旧的屏蔽-新的屏蔽

A B C - C D E = A B

SIG\_SETMASK - 直接使用新的屏蔽

set：传入的新的屏蔽信号集

oldset：传出参数,传出旧的屏蔽信号集

返回值：

成功返回0，失败返回-1

sigpending ----------- 查询在信号屏蔽期间，有没有收到某个信号

将屏蔽期间来过得信号放入信号集

作业：

1.改进有名管道通信程序，实现任意先执行读端和写端，运行时创建管道文件，ctrl+c退出时删除管道文件

一.信号

1.sigaction和sigqueue的用法(了解)

sigaction是一个设置信号处理方式的函数

参数：

signum:要设置的信号

act:要设置成的处理方式

oldact:传出参数,传出之前的信号处理方式

struct sigaction {

void (\*sa\_handler)(int);//信号处理函数1

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);//信号处理函数2

sigset\_t sa\_mask;//信号处理期间要屏蔽的信号集

int sa\_flags;//标识

void (\*sa\_restorer)(void);

};

sa\_flags:

SA\_NOCLDWAIT(SIGCHLD):子进程结束后自己回收资源，不会变为僵尸进程

SA\_ONSTACK/SA\_RESETHAND:对信号的处理方式的设置只生效一次，处理一次后恢复为默认

SA\_SIGINFO：使用信号处理函数2

sigqueue函数在发送信号时可以带额外的数据

二.标准IPC(XSI IPC)

标准IPC包括 共享内存 消息队列 信号量集，他们遵循同样的标准：

（1）所有的IPC结构都有一个内部ID作为唯一标识

（2）内部ID的获取借助外部的key，类型key\_t（相同的key找到同一个IPC）

（3）key的获取方式有三种

a.使用宏 IPC\_PRIVATE 作为key,这种方式外部无法获取,因此不推荐使用

b.使用ftok函数获取一个key

c.在头文件中统一定义所有的key

（4）用key获取ID的函数都是xxxget(),比如 shmget(), msgget(),semget()

（5）每一种IPC结构多提供一个xxxctl()函数，这个函数的功能至少包括：查询 修改和删除

这些函数都有一个cmd参数：

IPC\_STAT：查询

IPC\_SET：修改

IPC\_RMID：通过ID删除

（6）IPC结构共用的命令

ipcs:查询当前的IPC结构

ipcrm:删除当前IPC结构(通过ID删除)

-m:共享内存

-q:消息队列

-s：信号量集

三.共享内存

1.特性和原理

系统已一块物理内存作为媒介，通常情况下，两个进程不允许映射相同的内存，共享内存允许多个进程同时映射同一片物理内存

实现原理：

（1）内核取出一块物理内存，内核负责管理

（2）允许所有的进程对这块物理内存进行映射

（3）这样就实现了多个进程访问同一片内存，实现了进程间信息的交互

由于内存访问速度很快，共享内存是效率最高的IPC

共享内存允许多个内存同时访问，需要额外的互斥机制进行保护，防止由于多个进程同时操作造成数据混乱

shmget函数：用来创建共享内存

头文件：

 #include <sys/ipc.h>

 #include <sys/shm.h>

函数原型：

    int shmget(key\_t key, size\_t size, int shmflg);

参数：

    key：进程间通信的键值，ftok的返回值

    size：共享内存的大小

    shmflg：标识函数的行为及共享内存的权限。取值如下：

           IPC\_CREAT：如果不存在就创建共享内存

           IPC\_EXCL:和IPC\_CREAT搭配使用，如果已经存在，则返回失败

           权限位：设置共享内存的访问权限

    返回值：成功返回一个非负整数，即共享内存的标识符；失败返回-1

shmat函数：将共享内存段映射到进程地址空间。

即让进程和共享内存之间建立一种联系，让进程的某个指针指向此共享内存。

头文件：

#include <sys/types.h>

#include <sys/shm.h>

函数原型：

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

参数：

    shmid：共享内存标识符，shmget的返回值

    shmaddr：指定映射地址（若为NULL，则由系统自动指定）

    shmflg：共享内存的访问权限和映射条件（通常为0），取值如下：

         0：共享内存具有可读可写权限

            SHM\_RDONLY：映射的内存只读

            SHM\_RND:shmaddr非空时才有效。

                    shmaddr不为NULL且shmflg无IPC\_RND标记，则以参数shmaddr为连接地址；

                    shmaddr不为NULL且shmflg指定了IPC\_RND标记，则自动将参数shmflg调整为SHMLBA的整数倍。

                    公式：shmaddr = shmaddr-(shmaddr % SHMLBA)

返回值： 成功返回一个指针，指向共享内存第一个节；失败返回-1

SHMLBA的定义的两种情况：

#define   SHMLBA   PAGE\_SIZE

#define   SHMLBA   (4 \* PAGE\_SIZE)

shmdt函数：将共享内存段与当前进程脱离

头文件：

#include <sys/types.h>

#include <sys/shm.h>

函数原型：

    int shmdt(const void \*shmaddr);

参数：

    shmaddr: 由shmat所返回的指针

返回值：成功返回0；失败返回-1

注意：将共享内存段与当前进程脱离不等于删除共享内存段。

shmctl函数：用于控制共享内存

头文件：

#include <sys/ipc.h>

#include <sys/shm.h>

函数原型：

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

参数：

    shmid:由shmget返回的共享内存的标识码

    cmd:将要采取的动作，有如下取值：

        IPC\_RMID:删除共享内存

        IPC\_SET:在进程有足够权限的前提下，把共享内存的当前关联值设置为shmid\_ds数据结构中给出的值

        IPC\_STAT:把shmid\_ds结构中的数据设置为共享内存的当前关联值

        SHM\_LOCK:锁定共享内存段（超级用户）

        SHM\_UNLOCK:解锁共享内存段

    buf：指向一个保存着共享内存的模式状态和访问权限的数据结构

返回值：成功返回0；失败返回-1

注意：

SHM\_LOCK 用于锁定内存，禁止内存交换。并不代表共享内存被锁定后禁止其它进程访问。其真正的意义是：被锁定的内存不允许被交换到虚拟内存中。这样做的优势在于让共享内存一直处于内存中，从而提高程序性能。

应用小例子：客户端循环26次，每次循环由'A'向后+1，服务器端循环26次，接收客户端的数据并打印出来。

comm.h

#ifndef \_COMM\_H\_

#define \_COMM\_H\_

#include<stdio.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#define PATHNAME "."

#define PROJ\_ID 0x0001

int CreateShm(int size);

int DestroyShm(int shmid);

int GetShm(int size);

#endif

comm.c

#include "comm.h"

static int CommShm(int size,int flags){

key\_t key = ftok(PATHNAME,PROJ\_ID);

if(key < 0){

perror("ftok");

return -1;

}

int shmid = 0;

if((shmid = shmget(key,size,flags)) < 0){

perror("shmget");

return -1;

}

return shmid;

}

int CreateShm(int size){

return CommShm(size,IPC\_CREAT|IPC\_EXCL|0666);

}

int GetShm(int size){

return CommShm(size,IPC\_CREAT);

}

int DestroyShm(int shmid){

if(shmctl(shmid,IPC\_RMID,NULL) < 0){

perror("shmctl");

return -1;

}

return 0;

}

server.c

#include "comm.h"

int main()

{

int shmid = CreateShm(4096);

char \*addr = shmat(shmid,NULL,0);

sleep(2); //2 秒

int i = 0;

while( i++ < 26){

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

sleep(1);

}

shmdt(addr);

sleep(2);

DestroyShm(shmid);

return 0;

}

client.c

#include "comm.h"

int main()

{

int shmid = GetShm(4096);

char \*addr = shmat(shmid,NULL,0);

sleep(2);

int i = 0;

while(i < 26){

addr[i] = 'A'+ i;

i++;

addr[i] = 0;

sleep(1);

}

shmdt(addr);

sleep(2);

return 0;

}

2.使用方法

（1）获取key，通过ftok函数

参数：

pathname:存在的，可访问的路径(/ .)

proj\_id：8位非0数据(‘a’...)

返回值：

成功返回key，失败返回-1

（2）通过key创建/获取共享内存ID

参数：

key:上一步获取的key

size：创建时有效，共享内存的大小

shmflg:获取时给0，创建时用：IPC\_CREAT|0666

返回值：

成功返回共享内存ID，失败返回-1

（3）使用shmat()函数挂接(映射)共享内存

参数：

shmid:共享内存ID

shmaddr:映射的目标地址(NULL/0 让系统自动选择映射地址)

shmflg:SHM\_RDONLY表示只读，给0表示可读可写

返回值：

成功返回映射地址，失败返回(void \*)-1

（4）使用共享内存

。。。

（5）使用完之后脱接共享内存(解除映射)

shmdt -------- 传入shmat的返回值

（6）如果不再使用可以使用shmctl删除共享内存

参数：

shmid:共享内存的ID

cmd:

IPC\_STAT：查询

IPC\_SET：修改

IPC\_RMID：通过ID删除

buf：修改时传入要修改的信息，查询传出查询的信息

注：修改时只能修改uid gid 和权限，删除时只是添加一个删除标志，只有当挂接进程数为0时才真正删除

共享内存虽然速度很快，但是当多个进程同时写数据时，造成互相覆盖，导致数据混乱

消息队列就可以解决多个进程同时写数据的问题

四.消息队列

消息队列就是存放消息的队列，队列是一种先进先出的线性数据结构，消息队列中存放的时一个个消息，所以我们的传输数据应该先放入消息中，再放入队列

1.编程方法

（1）通过ftok获取外部的key

（2）通过key 创建/获取 消息队列ID，函数msgget()

参数：

key：上一步获取的key

msgflag：标识(创建时有效)

IPC\_CREAT|0666

返回值：

成功返回消息队列ID，失败返回-1

（3）使用消息队列 存入消息/取出消息 -------- msgsnd()/msgrcv()

struct msgbuf {

long mtype; /\* 消息类型，>0 \*/

char mtext[1]; /\* 消息数据 \*/

};

参数：

msqid:上一步获取的消息队列ID

msgp:消息的地址，消息是一个结构体，包含消息类型和消息数据两个部分

msgsz：消息中消息数据的长度

msgflg:选项,0标识阻塞等待(队列满了)，IPC\_NOWAIT表示不阻塞(队列满了立即返回错误)

返回值：

成功返回0，失败返回-1

参数：

前三个参数和msgsnd一样

msgtyp:可以让接收者有选择地接收消息

0:接收任意类型的第一个消息

>0:接收类型为msgtyp的第一个消息

<0:接收类型 小于等于 msgtyp绝对值 的消息，从小到大接收

msgflg:0表示阻塞(消息队列空)，IPC\_NOWAIT表示不阻塞

返回值：

返回成功读取消息中消息数据的大小，失败返回-1

（4）如果不再使用消息队列，可以使用msgctl()删除

用法和shmctl类似

五.信号量集

信号量就是一个计数器，用于控制共享资源的最大并行数，信号量集计时信号量的集合(数组)

信号量不是用于进程间传递数据，而是用于进程之间的互斥和同步

1.信号量的工作方式

设置一个初始技术，代表资源的个数，每来一个进程访问共享资源，计数值-1(P操作)，每离开一进程，计数值+1(V操作)，

计数值为0时候不允许进程访问，直到计数值重新大于0位置

2.互斥和同步

互斥就是只有一个共享资源，这样访问资源的进程就构成互斥的关系

同步表示多个进程之间必须按照一定的顺序执行

3.信号量集的编程方法

（1）获取key

（2）通过key创建/获取信号量集ID

参数：

key:上一步

nsems:信号量集中信号量的个数(数组长度)

semflg:和共享内存，消息队列一样 IPC\_CREAT|0666

返回值：

成功返回ID，失败返回-1

（3）设置信号量的初始计数semctl()

参数：

semid:上一步

semnum:信号量在信号量集中的下标

cmd:

IPC\_STAT：查询

IPC\_SET：修改

IPC\_RMID：通过ID删除

SETVAL:将第四个参数设置为信号量集中第semnum个信号量的初始值

...:

设置初始值：semctl(semid,index,SETVAL,value/\*初始值\*/);

返回值：

成功返回0，失败返回-1

（4）使用semop()函数对信号量进行PV操作

sembuf结构中包含以下成员：

unsigned short sem\_num; /\* 操作信号的下标 \*/

short sem\_op; /\* 操作方式 1 or -1 \*/

short sem\_flg; /\* 0:等待 IPC\_NOWAIT：不等待 \*/

参数：

semid:信号量集ID

sops:操作结构体数组首地址

nsops:结构数组元素个数

返回值：

成功返回0，失败返回-1

（5）如果不再使用可以用semctl()删除

信号量的使用

P操作

访问共享资源/执行临界区代码

V操作

作业：

使用信号量集对共享内存的访问进行互斥保护

一.信号

1.sigaction和sigqueue的用法(了解)

sigaction是一个设置信号处理方式的函数

参数：

signum:要设置的信号

act:要设置成的处理方式

oldact:传出参数,传出之前的信号处理方式

struct sigaction {

void (\*sa\_handler)(int);//信号处理函数1

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);//信号处理函数2

sigset\_t sa\_mask;//信号处理期间要屏蔽的信号集

int sa\_flags;//标识

void (\*sa\_restorer)(void);

};

sa\_flags:

SA\_NOCLDWAIT(SIGCHLD):子进程结束后自己回收资源，不会变为僵尸进程

SA\_ONSTACK/SA\_RESETHAND:对信号的处理方式的设置只生效一次，处理一次后恢复为默认

SA\_SIGINFO：使用信号处理函数2

sigqueue函数在发送信号时可以带额外的数据

二.标准IPC(XSI IPC)

标准IPC包括 共享内存 消息队列 信号量集，他们遵循同样的标准：

（1）所有的IPC结构都有一个内部ID作为唯一标识

（2）内部ID的获取借助外部的key，类型key\_t（相同的key找到同一个IPC）

（3）key的获取方式有三种

a.使用宏 IPC\_PRIVATE 作为key,这种方式外部无法获取,因此不推荐使用

b.使用ftok函数获取一个key

c.在头文件中统一定义所有的key

（4）用key获取ID的函数都是xxxget(),比如 shmget(), msgget(),semget()

（5）每一种IPC结构多提供一个xxxctl()函数，这个函数的功能至少包括：查询 修改和删除

这些函数都有一个cmd参数：

IPC\_STAT：查询

IPC\_SET：修改

IPC\_RMID：通过ID删除

（6）IPC结构共用的命令

ipcs:查询当前的IPC结构

ipcrm:删除当前IPC结构(通过ID删除)

-m:共享内存

-q:消息队列

-s：信号量集

三.共享内存

1.特性和原理

系统已一块物理内存作为媒介，通常情况下，两个进程不允许映射相同的内存，共享内存允许多个进程同时映射同一片物理内存

实现原理：

（1）内核取出一块物理内存，内核负责管理

（2）允许所有的进程对这块物理内存进行映射

（3）这样就实现了多个进程访问同一片内存，实现了进程间信息的交互

由于内存访问速度很快，共享内存是效率最高的IPC

共享内存允许多个内存同时访问，需要额外的互斥机制进行保护，防止由于多个进程同时操作造成数据混乱

2.使用方法

（1）获取key，通过ftok函数

参数：

pathname:存在的，可访问的路径(/ .)

proj\_id：8位非0数据(‘a’...)

返回值：

成功返回key，失败返回-1

（2）通过key创建/获取共享内存ID

参数：

key:上一步获取的key

size：创建时有效，共享内存的大小

shmflg:获取时给0，创建时用：IPC\_CREAT|0666

返回值：

成功返回共享内存ID，失败返回-1

（3）使用shmat()函数挂接(映射)共享内存

参数：

shmid:共享内存ID

shmaddr:映射的目标地址(NULL/0 让系统自动选择映射地址)

shmflg:SHM\_RDONLY表示只读，给0表示可读可写

返回值：

成功返回映射地址，失败返回(void \*)-1

（4）使用共享内存

。。。

（5）使用完之后脱接共享内存(解除映射)

shmdt -------- 传入shmat的返回值

（6）如果不再使用可以使用shmctl删除共享内存

参数：

shmid:共享内存的ID

cmd:

IPC\_STAT：查询

IPC\_SET：修改

IPC\_RMID：通过ID删除

buf：修改时传入要修改的信息，查询传出查询的信息

注：修改时只能修改uid gid 和权限，删除时只是添加一个删除标志，只有当挂接进程数为0时才真正删除

共享内存虽然速度很快，但是当多个进程同时写数据时，造成互相覆盖，导致数据混乱

消息队列就可以解决多个进程同时写数据的问题

四.消息队列

消息队列就是存放消息的队列，队列是一种先进先出的线性数据结构，消息队列中存放的时一个个消息，所以我们的传输数据应该先放入消息中，再放入队列

1.编程方法

（1）通过ftok获取外部的key

（2）通过key 创建/获取 消息队列ID，函数msgget()

参数：

key：上一步获取的key

msgflag：标识(创建时有效)

IPC\_CREAT|0666

返回值：

成功返回消息队列ID，失败返回-1

（3）使用消息队列 存入消息/取出消息 -------- msgsnd()/msgrcv()

struct msgbuf {

long mtype; /\* 消息类型，>0 \*/

char mtext[1]; /\* 消息数据 \*/

};

参数：

msqid:上一步获取的消息队列ID

msgp:消息的地址，消息是一个结构体，包含消息类型和消息数据两个部分

msgsz：消息中消息数据的长度

msgflg:选项,0标识阻塞等待(队列满了)，IPC\_NOWAIT表示不阻塞(队列满了立即返回错误)

返回值：

成功返回0，失败返回-1

参数：

前三个参数和msgsnd一样

msgtyp:可以让接收者有选择地接收消息

0:接收任意类型的第一个消息

>0:接收类型为msgtyp的第一个消息

<0:接收类型 小于等于 msgtyp绝对值 的消息，从小到大接收

msgflg:0表示阻塞(消息队列空)，IPC\_NOWAIT表示不阻塞

返回值：

返回成功读取消息中消息数据的大小，失败返回-1

（4）如果不再使用消息队列，可以使用msgctl()删除

用法和shmctl类似

五.信号量集

信号量就是一个计数器，用于控制共享资源的最大并行数，信号量集计时信号量的集合(数组)

信号量不是用于进程间传递数据，而是用于进程之间的互斥和同步

1.信号量的工作方式

设置一个初始技术，代表资源的个数，每来一个进程访问共享资源，计数值-1(P操作)，每离开一进程，计数值+1(V操作)，

计数值为0时候不允许进程访问，直到计数值重新大于0位置

2.互斥和同步

互斥就是只有一个共享资源，这样访问资源的进程就构成互斥的关系

同步表示多个进程之间必须按照一定的顺序执行

3.信号量集的编程方法

（1）获取key

（2）通过key创建/获取信号量集ID

参数：

key:上一步

nsems:信号量集中信号量的个数(数组长度)

semflg:和共享内存，消息队列一样 IPC\_CREAT|0666

返回值：

成功返回ID，失败返回-1

（3）设置信号量的初始计数semctl()

参数：

semid:上一步

semnum:信号量在信号量集中的下标

cmd:

IPC\_STAT：查询

IPC\_SET：修改

IPC\_RMID：通过ID删除

SETVAL:将第四个参数设置为信号量集中第semnum个信号量的初始值

...:

设置初始值：semctl(semid,index,SETVAL,value/\*初始值\*/);

返回值：

成功返回0，失败返回-1

（4）使用semop()函数对信号量进行PV操作

sembuf结构中包含以下成员：

unsigned short sem\_num; /\* 操作信号的下标 \*/

short sem\_op; /\* 操作方式 1 or -1 \*/

short sem\_flg; /\* 0:等待 IPC\_NOWAIT：不等待 \*/

参数：

semid:信号量集ID

sops:操作结构体数组首地址

nsops:结构数组元素个数

返回值：

成功返回0，失败返回-1

（5）如果不再使用可以用semctl()删除

信号量的使用

P操作

访问共享资源/执行临界区代码

V操作

作业：

使用信号量集对共享内存的访问进行互斥保护

一.线程

1.概念

主流操作系统，都是支持多进程，每个进程内部可以启动多线程完成代码的并行，每个进程内部可以无限启动多个线程

线程是一种轻量级的代码并行，，不需要创建过多的内存空间，线程共享所在进程的内存空间。线程只有自己一个额外的独立的线程栈

在Linux操作系统，本身不提供线程功能，需要借助外部线程库(pthread)来实现线程

Linux中进程和线程都对应一个任务(task\_struct),参与任务调度。

每个进程都有一个主线程---main函数，主线程结束，进程结束，进程的所有线程结束

多线程之间既相互独立，也会互相影响

本质上来说，运行一个线程就是运行对应的线程函数。

注:man手册安装pthread系列函数的说明

虚拟机网络畅通

选择国内软件源

sudo apt-get update

sudo apt-get install manpages-dev manpages-posix manpages-posix-dev

2.线程编程

（1）创建线程

pthread线程库的函数声明在pthread.h中，代码封装在libpthread.so中，线程相关的函数都以pthread\_开头

参数：

thread:传出参数，传出线程ID

attr:线程属性，一般我们使用默认属性(给0)

start\_routine:线程函数，就是线程执行代码

arg:传递给线程函数的参数

返回值：

成功返回0，失败返回错误码(使用strerror获取错误字符串)

线程启动以后，只执行线程函数，主线程main也在运行，他们以并行的方式执行

（2）获取线程ID

返回本线程ID

（3）等待线程结束

参数：

thread:要等待的线程ID

retval:传出参数，获取线程的返回值(void \*)

返回值：

成功返回0，失败返回错误码

该函数的作用是等待某个线程结束，回收资源

练习：

创建一个线程，传入圆的半径，在线程中打印圆的面积，同时将圆的面积返回交给主线程

（4）线程的状态

线程可以分为join状态和分离状态

join状态：

线程调用了pthread\_join函数，就处于join状态，也就是线程结束后暂不回收四中院，直到pthread\_join结束后才回收

detach(分离)状态：

线程一旦结束，不用管其他线程，直接回收资源，函数pthread\_detach()设置线程的分离状态

传入要设置的线程ID

（5）线程的退出

正常退出：

线程函数中执行return语句

执行pthread\_exit函数

非正常退出：

自身出现错误

被其他进程/线程 终止/取消 pthread\_cancel()

（6）线程退出处理函数 pthread\_cleanup\_push()/pthread\_cleanup\_pop()

参数：

routine:退出处理函数

arg:传递给退出处理函数的参数

参数：

execute: 非0：取出一个退出处理函数执行

0：取出一个退出处理函数不执行

注意事项：

pthread\_cleanup\_push和pthread\_cleanup\_pop必须成对出现，如果不希望执行pthread\_cleanup\_pop可以将该语句写到线程函数返回之后

线程退出处理函数在以下三种情况会被调用

a.线程执行pthread\_exit()

b.线程被取消

c.执行pthread\_cleanup\_pop(非0)

（7）进程函数和线程函数的比较

进程 线程

创建 fork()/vfork() pthread\_create()

获取ID getpid() pthread\_self()

退出 exit()/\_exit() pthread\_exit()

等待结束 wait()/waitpid() pthread\_join()

取消 kill()/... pthread\_cancel()·

设置退出处理函数 atexit() pthread\_cleanup\_push()/pthread\_cleanup\_pop()

替换进程内容 exec系列

设置线程属性 pthread\_attr系列

（8）函数的可重入

函数的可重入性指的是在多线程编程中，是否可以被多个线程同时调用

如果在函数内部访问了全局数据或者静态数据并且没有做任何的保护措施，那么该函数就不可重入

练习：

使用多线程编程和消息队列，实现两个进程之间的聊天

二.线程的互斥和同步

互斥锁 ---------- 线程官方提供的保护机制

信号量 ---------- 线程保护机制

条件变量 -------- 通常用来实现同步

作业：

将上面练习中的聊天改为任意多进程的聊天室

一.线程的互斥和同步

1.互斥锁

互斥锁是线程官方提供的同步互斥技术，可以在线程中直接使用，不需要额外维护

编程步骤

（1）声明互斥锁

pthread\_mutex\_t lock;

（2）初始化互斥锁

pthread\_mutex\_init(&lock,0);

注：可以再声明时初始化

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

（3）加锁

pthread\_mutex\_lock(&lock);

（4）访问共享资源

...

（5）解锁

pthread\_mutex\_unlock(&lock);

（6）不再使用释放锁的资源

pthread\_mutex\_destroy(&lock);

2.信号量

信号量是一个计数器，用于控制访问共享资源的最大并行 进程/线程 计数

信号量不属于线程的范畴,不再线程库中，只是一个线程计数辅助工具，头文件semaphore.h，需要额外进行维护

如果信号量的初始值设置为1，效果等同于互斥锁

编程步骤：(无名信号量)

（1）声明信号量

sem\_t sem;

（2）初始化信号量

sem\_init()

参数：

sem:要初始化的信号量

pshared: 0 ---------- 用于线程之间

非0 --------- 用于进程之间(在Linux中没有实现)

value:信号量的初始值

返回值：

0成功，-1失败

（3）计数减1(P操作)

sem\_wait(&sem);

sem\_wait获取不到阻塞

sem\_trywait获取不到立即返回(判断返回值)

sem\_timedwait获取不到进入有超时时间的阻塞

（4）访问共享资源

....

（5）计数加1(V操作)

sem\_post(&sem)

（6）不再使用释放信号量

sem\_destroy(&sem);

练习：

将互斥锁的代码修改为用信号量控制

有名信号量的用法和无名信号量类似

（1）有名信号量的创建.打开

（2）关闭和销毁

注:有名信号量的P V操作函数和无名信号量一样

3.条件变量

条件变量常用于线程同步,条件变量的实现需要互斥锁(mutex)的支持

编程方法：

（1）声明初始化

pthread\_cond\_t cond;

pthread\_cont\_init(&cond,0);

or

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

（2）等待条件

pthread\_cond\_wait(&cond,&mutex);

（3）完成操作

....

（4）唤醒条件

pthread\_cond\_signal --------- 唤醒一个等待条件

pthread\_cond\_broadcast ------ 唤醒所有的条件(惊群)

（5）不再使用可以销毁

二.守护进程

1.概念

前台进程:占用终端，在前台运行的进程

后台进程，从属于终端，不占用终端前台，在后台运行

前台进程和后台进程一般都有一个控制终端，控制终端默认是启动进程的终端

当一个进程的控制终端结束时，该进程也会终止

守护进程也叫精灵进程，守护进程是一种后台进程，但是没有控制终端，可以在后台独立运行

2.创建守护进程

把一个进程设置为守护进程

参数：

nochdir:给0，将工作目录改为根目录

noclose：给0，将0,1,2重定向到/dev/null

返回值：

成功返回0，失败返回-1

用法：

daemon(0,0);

3.系统日志(syslog)

系统日志保存在 /var/log/syslog

（1）打开日志 ---- openlog

ident:日志前缀

option:选项

facility:设施名

（2）打印日志 -------- syslog

参数：

priority:优先级

后面的参数了私语printf

（3）关闭日志 ------- closelog()

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

补充：手动启动守护进程

进程组:多个进程可以构成一个进程组，一个进程组有一个组长进程，只有非进程组组长才可以成立新的会话期

会话期:会话期是多个进程租的集合，会话期的首进程就是该会话的领头进程，可以分为前台进程组和后台进程组，一个会话可以有一个控制终端，也可以没有

控制终端:属于一个会话，当控制终端退出，导致所在会话的所有进程退出

守护进程的创建过程

1.第一次fork，保留子进程(防止是进程组长)，为成立新的会话做准备

2.调用setsid()成立新的会话,成为新会话的领头进程，也是进程组组长(没有控制终端)

3.忽略信号SIGHUP(终端发送的信号)，进行第二次fork，保留子进程(防止打开控制终端)

4.关闭所有文件描述符

for(i=0;i<sysconf(\_SC\_OPEN\_MAX),i++)

close(i);

将标准输入，标准输出,标准错误重定向 /dev/null

int fd = open("/dev/null",O\_RDWR);

dup(fd);

dup(fd);

5.消除umask影响

umask(0);

6.修改进程工作目录

chdir("/");

7.使用文件锁保证服务器互斥运行

int res = fcntl(fd,F\_SETLK,&lock);

if(res==-1)

exit(0);

8.使用系统日志来记录日志信息

9.完成服务工作

三.系统编程总结

四.项目

1.使用线程池拷贝目录

2.贪吃蛇游戏

800X480 20X20

蛇头 按照触摸坐标方向移动，蛇头以外的移动到前一块，蛇尾去掉一块

目标生成 20X20 随机位置

碰撞处理 --------- 撞墙 撞目标 撞自己

if(evt.type==EV\_SYN)

改为

if(evt.type==EV\_KEY&&evt.value==1)//按下

课程：Linux驱动

1.ARM原理和裸机开发

计算机原理和嵌入式概念

ARM体系结构和指令集

接口开发

2.Linux内核驱动

系统移植介绍

内核模块的使用及相关

字符设备驱动实现

内核IO操作及使用

内核竞态和并发

内核中断使用

内核时间流

内核内存相关

设备驱动模型(platform)

输入子系统

混杂设备(framebuffer,V4L2)

块设备和网络设备驱动分析

一.ARM原理

1.嵌入式相关概念

以应用为中心，以计算机技术为基础，软硬件可裁剪，适用应用系统对功能，可靠性，功耗，成本，体积有严格要求的专用计算机系统

2.具体应用

消费类电子 ----------- 手持媒体设备(手机，平板)，数码相机，打印机，智能家电

工业控制 ----------- 工业过程控制(流水线)，数字机床，电力系统，电网设备

交通运输 ---------- 车辆导航 流量控制 信息监控和汽车服务，行车记录仪，停车厂和告诉路口的收费系统

医疗设备 ---------- CT B超 口腔镜 ..............

环境监控 ---------- 水文实时监控，防洪和水土质量监控，堤坝安全监控，地震监控，气象信息监测，水源和空气监测

军事和航天 -------- .........

3.计算机原理

计算机由输入设备，输出设备，存储器，运算器，控制器五部分组成

（1）运算器和控制器共同组成CPU

运算器进行信息处理和运算，核心部件是ALU(算术逻辑运算单元)

控制器是整个计算机的指挥中心

（2）输入设备

输入设备的任务是将人编好的程序和原始数据传送到计算机中，并且将其转换成计算机内部能识别的信息方式

比如:鼠标 键盘 触摸屏，扫描仪，摇杆...

（3）输出设备

输出设备的任务是将计算机处理的结果以人活着其他设备能够接受的方式送出计算机

比如:显示器，打印机，绘图仪

（4）存储器

存储器用来存放程序和数据的部件，也是计算机能够实现"存储程序控制"的基础

三级存储结构：

a.高速缓存

速度最快，价格最贵，容量最小，断电数据丢失，CPU可以直接访问

存放正在执行的程序中的活跃部分，以便快速响应CPU

b.主存储器(内存)

速度，价格，容量介于高速缓存和辅助存储器之间,CPU可以直接访问

存放正在执行的程序

c.辅助存储器(磁盘)

速度最慢，价格最低，容量最大,CPU不能直接访问

存放暂时不使用的程序和数据，使用时在传送到主存

4.总线

总线为多个部件实现公共信息传输的线路

总线按照传递的信息的类型分为:数据总线 地址总线 控制总线

5.冯诺依曼架构和哈佛架构

6.ARM的历史

center process unit

cambridge process unit

Acorn RISC Machines

Advanced RISC Machines ---------- 1991

ARM公司的业务进行CPU架构的设计-----------ARM架构

ARM intel mips powerpc

不生产设计具体的芯片，架构授权给芯片公司

华为海思 瑞芯微 炬力 全志 联发科

三星 intel 意法半导体 高通 TI 苹果

芯片公司拿到ARM的授权自己设计芯片，交给生产芯片的公司生产

intel 三星 台积电

7.RISC和CISC

RISC:精简指令集

结构简单，选择使用频率比较高的简单指令，指令长度固定，一般为单周期指令，复杂操作通过多条指令组合实现

在功耗，体积，价格有很大优势，常用于嵌入式领域(ARM MIPS)

CISC：复杂指令集

侧重于执行指令的功能性，处理器硬件结构复杂

指令复杂，指令长度和执行周期不固定，在处理能力上有优势，常用于PC，服务器领域(x86)

8.ARM处理器的版本

指令集版本 ARM核版本 对应处理器

V1-V3

V4 ARM7

V5 ARM9 Samsung 2440

V6 ARM11 Samsung 6440

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

ARM11之后，ARM的产品改用cortex命令，分为A R M三个分支，面向不同的应用场景

cortex-A(Application)：面向尖端应用，使用虚拟内存的操作系统和用户应用

A8---->A9----->A7 A15----->A53 A57(64)---->A55 A72 A75 A33--->A76

cortex-R(Realtime):针对可靠性，实时性响应的嵌入式系统

R4--->R5---->R7---->R8

cortex-M(microchip)：针对成本，功耗敏感的MCU和中断应用，一般不运行操作系统

M0--->M3---->M4----->M7

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

V7 cortex-A8 A9 A7 A15 samsung 210 4412 4418

V8(64) cortex-A57 A53 A72 A55 samsung 6818

stm32f407处理器使用32位cortex-m4架构,指令集版本 ARMv7-M

s5p6818处理器使用64位8核cortex-A53架构，指令集版本 ARMv8-A

9.ARM指令中的数据长度

Byte(字节): 8bits

HalfWord： 16bits

Word(字)：32bits

DoubleWord：64bits

10.ARM指令集分类

大部分ARM处理器支持ARM指令集和THUMB指令集

ARM指令集:32bits(64bits),每条指令占用32位的存储空间

THUMB指令集:16bits,每条指令占用16位的存储空间

ARM处理器在ARM状态执行ARM指令，THUMB状态下执行THUMB指令

11.ARM处理器的工作模式

ARM处理器有7种工作模式，A系列有8种

User:用户模式，一般在执行应用程序处于该模式

System:系统模式，有特权的用户模式

IRQ:中断模式，处理中断处于该模式

FIQ:快速中断模式,处理快速中断处于该模式

SVC：管理模式，当系统复位/软中断进入该模式

Undef:未定义模式，当执行未定义指令进入该模式

Abort:中止模式,当数据指令存取异常进入该模式

Monitor(A系列独有)：监控模式,为安全扩展执行安全代码模式

模式分类：

特权模式和非特权模式：

除User以外的模式都是特权模式

正常模式和异常模式：

IRQ,FIQ,Undef,Abort,Svc属于异常模式

12.ARM寄存器组织

ARM的寄存器是处理器留给用户的接口

处于CPU内部，没有地址

寄存器中的数据可以在指令中直接使用，每个模式下使用的寄存器有区别

ARM一共有37个32位寄存器，A系列有40个32位寄存器

分类:

R0-R7：未分组的通用寄存器，所有的模式下共用同一组

R8-R12：分组通用寄存器,快中断模式下有单独的一组

R13：通用寄存器，通常用作堆栈指针(SP)

R14(LR)：链接寄存器，用于保存程序返回地址

R15(PC)：程序计数器，指向当前取指指令的地址

CPSR:当前程序状态寄存器，表示处理器当前的状态

关注其中高4位和低8位

0-4位:模式位，决定处理器处于什么模式

5位:状态位，决定处理器处于哪种状态

0:ARM状态

1:THUMB状态

6位:快中断禁止位，决定是否禁止快中断

1:禁止

0:允许

7位:中断禁止位,决定是否禁止中断

1:禁止

0:允许

28-31位:条件代码标志位，用于控制指令是否执行，(指令执行可以修改这些位)

N=1:运算结果小于0

Z=1:运算结果为0

C=1:运算发生进位/借位

V=1:运算结果发生溢出

SPSR:备份程序状态寄存器

在发生异常来备份CPSR

13.指令和数据对齐,多字节数据存储

ARM指令必须字对齐

THUMB指令必须半字对齐

大端格式：低地址保存数据高有效位，高地址保存数据低有效位

小端格式：低地址保存数据低有效位，高地址保存数据高有效位

ARM一般使用小端格式

14.指令流水线

流水线技术能大大提高指令执行效率

ARM7是3级流水，ARM9是5级流水

3级流水线：

将一条指令的执行分为三个阶段(三段独立电路):取指 译码 执行

取指:根据pc的值读取对应内存地址的指令到CPU

译码:指令译码器对指令进行识别，将指令翻译成具体的运算操作

执行:执行指令操作并且将执行结果写回寄存器

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

第n条指令: 取指 译码 执行

第n+1条指令： 取指 译码 执行

第n+2条指令: 取指 译码 执行

........

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

15.寻址方式

寻址方式就是找到操作数的方式

（1）立即寻址

操作数是立即数，立即数就是保存在指令中的数据

（2）寄存器直接寻址

操作数保存在寄存器中，指令记录寄存器编号

（3）寄存器移位寻址

寄存器中的数据通过移位得到操作数，指令记录寄存器编号

（4）寄存器间接寻址

操作数保存在内存中，寄存器记录的是操作数的内存地址

（5）基址寻址

操作数保存在内存中，寄存器记录的内存地址通过偏移得到操作数的地址

（6）多寄存器寻址

操作数保存在内存中，一条指令实现多个寄存器数据的传送

（7）堆栈寻址

以堆栈的方式操作寄存器中的数据

（8）相对寻址

跳转指令中以pc作为基地址前后偏移

二.ARM指令集

1.指令

编译完成后作为一条指令保存在内存中，CPU指令时完成某种操作

2.伪指令

本身不是指令，编译器在编译时使用指令替换他

3.伪操作

编译不会生成代码，也不占用内存，仅仅用于辅助编译

4.指令格式

执行条件：

一.ARM指令集

1.伪操作

.text ----------- 代码段

.data ------------ 数据段

.rodata ---------- 只读数据段

.bss ------------ bss段

.code 32/16 ---------- ARM指令/THUMB指令

.global ------------- 声明为全局

.extern ------------ 声明为外部

.space ------------- 分配指定大小的空间

.word ------------- 存储字数据到当前位置

.end ------------- 汇编结束

.entry ------------ 汇编入口

.equ -------------- 类似于C的宏定义

.align ------------ 强制对齐

2.C语言和汇编混合编程

（1）汇编调用C语言

在汇编中将C语言的函数声明为外部

将C语言中函数当做汇编中的标签(标号)，使用跳转指令调用

（2）C语言调用汇编

汇编中标签要声明为全局，C语言中声明为外部

将汇编中的标号当做C语言的函数，使用函数调用语句调用

汇编和C互相调用时传参遵循ATPCS规则

不超过4个参数的部分用r0,r1,r2,r3传递

超过4个的部分使用堆栈传递

返回值在r0中

1.volatile

用于修饰易变的变量，防止编译器优化

编译器在编译时，遇到运算的数据已经存在于寄存器中，并且没有被修改，那就会直接使用在寄存器中数据备份，而不去内存中再取一次，这种操作就叫编译器优化被volatile修饰的变量在使用时候不使用在寄存器的备份，必须去原始地址读取

使用场景：

（1）用于修饰多线程程序中的全局变量

（2）用于修饰中断程序中的自动变量

（3）用于修饰硬件寄存器的地址

2.ARM(cortex-A)异常处理流程

（1）异常源---------导致异常的情况

IRQ FIQ Reset 软中断 DataAbort PrefechAbort Undef

（2）使用对应异常模式来处理异常，对应关系

异常源 IRQ FIQ Reset 软中断 DataAbort PrefechAbort Undef

异常模式 IRQ FIQ SVC SVC Abort Abort Undef

（3）异常向量表

放在0地址，当异常发生时，处理器自动跳转到异常向量表对应的位置

（4）异常发生后，ARM处理器的动作(自动完成)

1.拷贝CPSR--->SPSR\_mode

2.修改CPSR：

a.将处理器状态切换为ARM状态

b.改变处理器模式进入对应的异常模式

c.如果需要可以禁止相应中断

3.保存返回地址到LR\_mode

4.设置PC指向异常向量表对应的位置

（5）异常处理完成后,返回时需要的工作(开发者完成)

1.从SPSR\_mode--->CPSR,使处理器恢复到异常发生前的状态

2.从LR\_mode恢复PC，程序返回到被异常打断的位置继续执行

3.快中断比中断快的原因

（1）FIQ的优先级高于IRQ，当两个中断同时发生时优先响应FIQ

（2）FIQ中有5个(R8-R12)私有寄存器，在处理中断是无需保护这几个寄存器

（3）FIQ处于异常向量表末尾，可以将异常处理代码写在异常向量表之后，省去跳转时间

二.ARM接口技术

samsung s5p6818 ---------- ARM cortex-A53 8核 64位 --------------- ARMv8-A

1.LED

（1）看原理图

高灭低亮

GPIOE13 GPIOC17 8 7

（2）看硬件芯片手册(略) CPU的芯片手册

配置输出高低电平

选择输入输出模式

选择复用功能，复用功能表在2.3.1章

（3）编写驱动代码(闪烁)

初始化：

设置复用功能为GPIO

设置GPIO为输出功能

闪烁：

while(1){

高电平

延时

低电平

延时

}

（4）编译代码-------交叉编译工具

arm-linux-gcc -nostartfiles -nostdlib -Ttext 40008000 -e led led.c -o led

-nostartfiles:不加入起始文件

-nostdlib:不加入标准库

-Ttext:指定代码运行地址

-e:程序入口

arm-linux-objcopy -O binary led led.bin

（5）下载开发板运行

修改uboot网络参数

print

set gatewayip 192.168.12.1

set ipaddr 192.168.12.4

set serverip 192.168.12.5

save

uboot命令行下载 tftp下载

uboot命令行:tftp 40008000 led.bin

出现TTTTTTTTTTTTTT

sudo /etc/init.d/tftpd-hpa restart ---------- 重启ubuntu的tftp服务

loadb下载

go 40008000

三.驱动开发环境搭建

1.安装source insight,解压内核源代码

2.创建内核源代码工程

项目--->新项目---->项目名字和项目文件路径----->内核源代码路径

文档选项---->C源代码---->过滤器加上 ;\*.S;\*.s

项目---->加入和删除项目文件--->将内核源代码文件全部加进去

项目--->同步文件

3.操作系统的组成

（1）引导程序

bootloader --------- boot+loader

完成硬件自启动，将硬件设置到一个合适的状态，为启动内核做准备

加载操作系统内核到内存，跳转到系统内核代码运行

uboot是一种引导程序，开源，支持多种CPU架构，支持引导多种操作系统

编译uboot

设置顶层Makfile ,设置编译器

选择配置文件

make fs6818\_config

编译

make CROSS\_COMPILE=arm-linux-

（2）操作系统内核

提供了操作系统的各个管理功能

Linux内核：

任务调度

内存管理

设备驱动

网络管理

虚拟文件系统

（3）根文件系统

系统和程序运行的工具和库.....

busybox ---------- 制作根文件系统的工具

制作常用的命令

一.安装交叉编译工具4.6.4

1.在主目录解压工具压缩包

tar -xJvf gcc-4.6.4.tar.xz

2.修改配置文件 /etc/environment

把旧版本交叉编译工具路径替换成 /home/tarena/gcc-4.6.4/bin

3.重启Ubuntu

二.编译uboot

make fs6818\_config

make

生成ubookpak.bin的二进制文件，这就是uboot镜像

拷贝到/tftpboot

将tools目录下的mkimage文件拷贝到/usr/bin

三.烧写uboot

下载到内存

tftp 48000000 ubootpak.bin

烧写到emmc

update\_mmc 2 2ndboot 0x48000000 0x200 0x55378

2：第二个分区

2ndboot:分区名

0x48000000:内存地址(源数据地址)

0x200:写入分区的偏移地址

0x55378：写入数据大小(字节) = uboot大小

三.编译内核

1.配置编译器

修改顶层Makefile

2.默认配置

make fs6818\_defconfig

3.配置内核

make menuconfig

Device Drivers --->

Input device support --->

[ ] Keyboards ---> (去掉自带按键驱动)

4.编译内核

make uImage

在arch/arm/boot生成一个文件uImage,就是Linux内核镜像

拷贝到/tftpboot

四.配置nfs根文件系统

1.将rootfs.tar.gz拷贝到主目录，解压

2.将rootfs通过nfs挂载出来 ----- nfs-kernel-server

修改/etc/exports配置文件,加上以下一行

/home/tarena//rootfs \*(rw,sync,no\_subtree\_check,no\_root\_squash)

3.重启nfs服务

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

五.设置开发板启动参数，使用我们制作的内核和根文件系统

设置uboot环境变量 bootcmd ------- 加载内核

bootargs ------ 传递给内核的参数

set bootcmd tftp 48000000 uImage\;bootm 48000000

set bootargs root=/dev/nfs nfsroot=192.168.12.5:/home/tarena/rootfs rw console=ttySAC0,115200 init=/linuxrc ip=192.168.12.4 loglevel=8

自己的64位系统

装

sudo apt-get install nfs-kernel-server

sudo apt-get install tftpd-hpa

sudo apt-get install gcc-multilib

sudo apt-get install lib32z1

sudo apt-get install ncurses-dev

五.linux驱动

Linux驱动属于内核一部分，但是我们可以将驱动设计成模块

内核模块时一种可以在系统运行时加载和卸载的机制

1.模块编程注意事项

（1）不能使用C库和C标准头文件

（2）使用GNU C

（3）没有内存保护机制

（4）不能处理浮点运算

（5）注意互斥并发和可移植性问题

2.模块编程源代码组成

必须实现加载函数和卸载函数

加载函数：

int xxx(void){

返回0表示正常

}

卸载函数：

void yyy(void){

}

//使用内核中提供的宏来修饰这两个函数

module\_init(xxx);//修饰加载函数

module\_exit(yyy);//修饰卸载函数

加载模块时模块加载函数会被自动调用

卸载模块的时候卸载函数会被自动调用

注:在内核中打印使用printk函数，用法和printf一样

2.编译模块

使用它内核中的编译模块的方法来编译模块

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

1 KERNELDIR=/home/tarena/gec6818/kernel-3.4.39

2

3 obj-m += hello.o

4

5 default:

6 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

7 install:

8 $(MAKE) -C $(KERNELDIR) M=$(PWD) INSTALL\_MOD\_PATH=/home/tarena/rootfs modules\_install

9

10 clean:

11 rm -rf \*.o \*.ko \*.mod\* \*.ord\* \*.sy\* .\*cmd .tmp\*

~

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

3.模块命令

insmod ----------- 加载模块

rmmod ------------ 卸载模块

lsmod ------------ 查看已加载模块信息

modprobe --------- 加载已安装的模块

我们在编写模块代码必须加上模块许可证

MODULE\_LICENSE("GPL");

除了许可证可以外，还可以加上模块作者 版本 描述信息 设备表 别名.....

MODULE\_AUTHOR("zzy");//作者

MODULE\_VERSION("1.0");//版本

MODULE\_DESCRIPTION("This is my first module!");//说明信息

4.内核打印级别

内核默认打印级别在/proc/sys/kernel/printk中可以查询到

方法1：我们可以通过修改该文件来修改内核的打印级别(数字越小,级别越高)

echo 5 > /proc/sys/kernel/printk

方法2：可以在uboot传参传递打印级别过来

set bootargs root=/dev/nfs nfsroot=192.168.12.5:/home/tarena/rootfs rw console=ttySAC0,115200 init=/linuxrc ip=192.168.12.4 loglevel=8 //级别8

set bootargs root=/dev/nfs nfsroot=192.168.12.5:/home/tarena/rootfs rw console=ttySAC0,115200 init=/linuxrc ip=192.168.12.4 quiet //级别4

set bootargs root=/dev/nfs nfsroot=192.168.12.5:/home/tarena/rootfs rw console=ttySAC0,115200 init=/linuxrc ip=192.168.12.4 debug //级别10

printk输出的内容是否会被打印出来取决于printk的打印级别是否高于内核的打印级别

printk的打印级别定义在printk.h中，如下

#define KERN\_EMERG "<0>" /\* system is unusable \*/

#define KERN\_ALERT "<1>" /\* action must be taken immediately \*/

#define KERN\_CRIT "<2>" /\* critical conditions \*/

#define KERN\_ERR "<3>" /\* error conditions \*/

#define KERN\_WARNING "<4>" /\* warning conditions \*/

#define KERN\_NOTICE "<5>" /\* normal but significant condition \*/

#define KERN\_INFO "<6>" /\* informational \*/

#define KERN\_DEBUG "<7>" /\* debug-level messages \*/

5.内核模块参数

内核模块参数不但可以在编写代码时设置

也可以在加载模块时设置

甚至可以在模块加载以后设置

（1）使用方法

声明一些变量

使用以下语法将这些变量设置成模块参数

module\_param

module\_param\_array ----------- 数组

在加载模块时通过 模块参数名=值 的方式可以设置模块参数的值

模块加载成功后，哪些权限非0的模块参数在 /sys/module/模块名/parameters/文件

通过修改文件中保存的数据可以实现对模块参数的修改

6.模块导出符号

模块导出符号可以将模块中的变量/函数导出，供其他模块使用

使用一下宏导出：

EXPORT\_SYMBOL

EXPORT\_SYMBOL\_GPL -------- 只有遵循GPL协议的模块才可以使用

如果一个模块使用了另一个模块中的函数/变量，该模块就依赖于另一个模块

在使用modprobe加载模块时，会自动加载依赖模块

insmod是不会做这件事情

一.字符设备驱动

1.Linux设备驱动分类

（1）字符设备

按字节流访问，一般按顺序访问

按键 串口 LCD 触摸屏....

访问字符设备驱动通过字符设备文件

（2）块设备

按数据块访问，块的大小(512Bytes 1K 2K)

具有随机访问能力

访问块设备驱动通过块设备文件

硬盘 U盘 SD卡 TF卡 nandflash DRAM

（3）网络设备

网卡设备，驱动实现结合网络协议栈

没有对应的设备文案，通过套接字访问

2.字符设备驱动

（1）驱动的作用

a.管理操作的硬件

b.给用户程序提供操作硬件的接口

（2）应用程序如何访问设备文件

设备文件也是文件一种，访问设备文件使用Linux提供的文件访问系统调用(文件IO)

访问设备文件和普通文件的方法是一样的

open close read write ioctl mmap.....

（3）应用程序如何通过设备文件找到对应设备驱动

设备文件和字符设备驱动都包含一个内容：设备号

通过设备文件中包含的设备号能够在内核中找到设备号相同设备驱动

设备号分为主设备号和次设备号：

主设备号用于找打对应的驱动

次设备号用于区分使用同一个驱动的不同的设备

内核中如何使用设备号？

数据类型:dev\_t(32位)

高12位:主设备号

低20位:次设备号

内核提供的设备号相关宏：

MKDEV --------- 通过主设备号次设备号构造设备号

MAJOR ---------- 通过设备号获取主设备号

MINOR ---------- 通过设备号获取次设备号

由于驱动内部有一个设备号，我们编写字符设备驱动先申请一个设备号

如何向内核申请设备号？

a.静态申请

先通过查看 /proc/devices 文件，找一个没有被使用主设备号

Character devices:

1 mem

5 /dev/tty

5 /dev/console

5 /dev/ptmx

.....

Block devices:

259 blkext

7 loop

8 sd

11 sr

....

根据设备个数分配次设备号，从0开始

构造设备号：

dev\_t dev = MKDEV(主设备号,次设备号);

调用register\_chrdev\_region向内核申请

b.动态申请(推荐)

调用alloc\_chrdev\_region直接向内核申请

释放设备号

调用unregister\_chrdev\_region将设备号还给内核

3.内核中和字符设备驱动相关的四个数据结构

（1）struct inode

作用:描述一个文件的物理属性

创建文件内核中就会创建一个对应inode结构，直到文件删除之后才销毁

dev\_t i\_rdev;//保存设备号

struct cdev \*i\_cdev;//指向一个字符设备驱动

一个文件只有一个inode

（2）struct file

作用:描述打开文件以后的状态属性

打开文件内核创建file结构，关闭文件时删除file结构

const struct file\_operations \*f\_op;//文件的操作函数集合，指向对应文件IO系统调用的内核实现

void \*private\_data;//驱动传递额外数据

一个文件可以有多个file

（3）struct cdev

代表一个字符设备驱动

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

struct cdev {

struct kobject kobj;

struct module \*owner;

const struct file\_operations \*ops;//驱动中操作函数集合

struct list\_head list;

dev\_t dev;//设备号

unsigned int count;

};

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

我们实现字符设备驱动起始就是往内核中注册cdev结构

cdev\_init--------初始化cdev

cdev\_add --------将cdev添加到内核

cdev\_del -------将cdev从内核删除

（4）struct file\_operations

内核中文件操作函数集合，在用户控件调用文件的系统调用，最终一定会调用struct file结构中的struct file\_operations中的函数接口

4.从访问字符设备文件到字符设备驱动的原理

字符设备文件带有设备号，当打开设备文件时(open),通过设备号去内核中遍历cdev数组，和其中cdev结构进行比较，找到那个和设备文件设备号相同的cdev结构，将对应cdev结构中操作函数集合(file\_operations )赋值给file结构中的(file\_operations ),调用file\_operations 中open函数，以后再通过文件系统调用访问设备文件，相当于访问cdev结构中的file\_operations对应的函数接口

我们实现字符设备驱动只需要向内核添加cdev结构

5.创建设备文件

（1）使用mknod命令 -------- /dev

mknod 设备文件名 设备文件类型 主设备号 次设备号

比如:mknod console c 5 1

mknod cdd c 244 0

添加函数接口ioctl ------- unlocked\_ioctl

（2）使用代码创建设备文件

a.创建设备类

class\_create

b.创建设备文件

device\_create

5.用户空间和内核空间数据传递

内核提供了实现该功能的函数，这些函数可能导致睡眠

copy\_to\_user ---------- 内核--->用户

copy\_from\_user--------- 用户--->内核

6.ioctl函数的用法

ioctl通常用于发送一些控制信息

ioctl中cmd参数分为4个部分

内核中提供了一些宏来构造cmd

\_IO(type,nr) ----------- 构造纯控制的命令

7.内核中gpio操作函数

gpio\_request -------------- 申请

gpio\_free -------------- 释放

gpio\_direction\_output --------- 输出功能

gpio\_direction\_input --------- 输入功能

gpio\_set\_value ------------- 设置输出值

gpio\_get\_value ------------- 读取输入值

s3c\_gpio\_setpull ----------- 设置上下拉

6818在内核中的gpio对应的值如下：

/\* gpio group pad start num. \*/

enum {

PAD\_GPIO\_A = (0 \* 32),

PAD\_GPIO\_B = (1 \* 32),

PAD\_GPIO\_C = (2 \* 32),

PAD\_GPIO\_D = (3 \* 32),

PAD\_GPIO\_E = (4 \* 32),

PAD\_GPIO\_ALV = (5 \* 32),

};

是第几个脚就将名字加上几

二.内核的竞态和并发

多个对象同时使用共享资源就会造成竞态，一个操作需要等待另一个操作完成叫并发

1.产生竞态的情形

（1）SMP ---------- 多核CPU

（2）进程和进程之间

（3）进程和中断

（4）中断和中断

访问共享资源的代码称为临界区

2.内核提供的互斥机制

（1）中断屏蔽

可以解决SMP以外的所有情况

使用方法：

在临界区前屏蔽中断，临界区代码之前完之后解除屏蔽

unsigned long flags;

local\_irq\_save(&flags);//local\_irq\_disable()

临界区代码.....

local\_irq\_restore(&flags);//local\_irq\_enable()

频繁使用中断屏蔽会造成系统响应变慢，长时间屏蔽中断非常危险，

尽量少使用中断屏蔽，如果一定他要使用要尽快解除

（2）原子操作

可以解决所有情况竞态

原子变量分为 位原子操作 和 整型原子操作

位原子操作：

set\_bit

clear\_bit

change\_bit

test\_bit

整型原子操作：

类型：atomic\_t

使用：

a.分配初始化

atomic\_t v = ATOMIC\_INIT(1);//v = 1

或者

atomic\_t v;

atomic\_set(&v,1);

b.使用整型原子变量

atomic\_set

atomic\_read

atomic\_add

atomic\_sub

atomic\_inc

atomic\_dec

atomic\_inc\_and\_test

atomic\_dec\_and\_test

......

(3)自旋锁

(4)信号量

一.自旋锁

1.标准自旋锁可以解决除了中断以外的所有竞态，使用衍生自旋锁可以解决所有情况的竞态

，如果获取不到就原地自旋(死循环，忙等待)，直到获取成功为止

临界区代码执行完成之后再释放自旋锁

2.自旋锁的使用

数据类型:spinlock\_t

使用步骤:

（1）分配初始化自旋锁

spinlock\_t lock;

spin\_lock\_init(&lock);

（2）获取锁

spin\_lock(&lock);//获取不到锁，忙等待

或者

spin\_trylock(&lock);//获取不到锁，函数返回false,获取成功返回true,必须判断返回值

（3）执行临界区代码

临界区代码执行一定要快，内部不能调用阻塞/睡眠 函数

如:copy\_xxx，kmalloc.....

（4）释放锁

spin\_unlock(&lock);

以上是标准自旋锁的用法，如果竞态中涉及到中断，必须使用衍生自旋锁

获取锁：

spin\_lock\_irqsave(&lock, flags);

释放锁：

spin\_unlock\_irqrestore(&lock, flags);

三.信号量

内核中的信号量本质上是睡眠锁

在访问临界区时允许睡眠

1.信号量的使用

（1）分配初始化

struct semaphore sema;

sema\_init(&sema,1);//互斥信号量

（2）获取信号量

down(&sema);//如果进程获取不到信号量，进入不可中断的睡眠状态

或者

down\_interruptible(&sema);//如果进程获取不到信号量，进入可中断的睡眠状态

当进程获取到信号量或者收到信号该函数返回，返回0表示获取到了信号量，返回非0表示被信号打断

或者

down\_trylock(&sema);//获取不到信号量，返回非0，获取到信号量，返回0，可用于中断上下文

（3）临界区代码

临界区执行之间可以很长，允许睡眠

（4）释放信号量

up(&sema);

//释放信号量

//唤醒睡眠的进程

四.自旋锁和信号量的扩展

1.自旋锁还有读写自旋锁，顺序锁

读写自旋锁把锁分为读锁和写锁，获取读锁之后允许继续获取读锁，不允许获取写锁

获取写锁之后既不允许获取读，也不允许获取写

顺序锁也分为读锁和写锁，获取读锁之后允许继续获取读锁，也允许获取写锁，但是在释放读锁之前必须判断读过程中是否被获取过写锁

如果获取过写锁，就应该重新进行读操作

获取写锁之后既不允许获取读，也不允许获取写

2.信号量也有读写信号量

读写信号量分为获取读信号量和获取写信号量，获取读信号量之后允许继续获取读信号量，不允许获取写信号量

获取写信号量后，既不允许获取读，也不允许获取写

3.另外还有互斥体

互斥体就相当于一个初始值为1的信号量

五.互斥机制的选择

六.Linux系统中的IO模型

1.同步阻塞式IO

系统调用read/write,内核同步响应，如果操作条件不满足，则应用程序阻塞直到条件满足解除阻塞，阻塞阶段让出CPU

2.同步非阻塞时IO

系统调用read/write,内核同步响应，不管操作条件是否满足，无论成功失败都立即返回

3.异步阻塞式IO

阻塞并不是读写操作本身，阻塞的是文件描述符本身，当描述符的状态变为可操作之后，解除描述符的阻塞状态，用户再对描述符进行读写操作

4.异步非阻塞时IO

对文件描述符的操作立即返回，具体读写操作在后台完成，完成之后通过信号/回调函数通知用户程序

七.同步阻塞式IO和非阻塞时IO

1.用户程序打开设备文件时，可以选择阻塞/非阻塞

默认使用阻塞式IO，如果需要使用非阻塞式，需要加上一个打开标志，O\_NONBLOCK

int fd = open("xxx",O\_RDWR);//阻塞式

int fd = open("xxx",O\_RDWR|O\_NONBLOCK);//阻塞式

2.内核中实现阻塞通常使用等待队列

等待队列通常用于阻塞同步操作，不涉及竞态

等待队列的实现机制：

（1）创建等待队列头并初始化

wait\_queue\_head\_t wq;

init\_waitqueue\_head(&wq);

或者

DECLARE\_WAIT\_QUEUE\_HEAD(wq);

方法一:

（2）分配等待队列

wait\_queue\_t wait;//等待队列

//初始化等待队列，将当前进程添加到等待队列中

init\_queue\_entry(&wait,current);

或者

DECLARE\_WAITQUEUE(wait,current);

（3）将等待队列加入等待队列头

add\_wait\_queue(&wait,&wq);

(4)使用等待队列

while(1){

//将当前任务设置为可中断状态

current->state = TASK\_INTERRUPTIBLE;

//判断设备是否可用

if(...)//可用

break;

if(收到信号)

break;

//让出CPU，进入睡眠

schedule();

}

//将进程设置为TASK\_RUNNING

current->state = TASK\_RUNNING;

//将当前进程从等待队列中移除

remove\_wait\_queue(&wq,&wait);

方法二：

（2）等待唤醒

wait\_event\_interruptible(&wq,condition);//根据条件是否满足决定是否进入睡眠

wait\_event

wait\_event\_timout

wait\_event\_interruptible\_timout

（3）唤醒休眠的进程

wake\_up\_interruptible

wake\_up

八.异步阻塞式IO

异步阻塞式IO对应应用空间的多路复用(select poll epoll)

异步阻塞式IO在内核对应的接口是poll函数

所以要实现异步阻塞式IO应该在内核中实现fp中的poll函数

1.poll函数的实现方法

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

unsigned int xxx\_poll(struct file \*filp, struct poll\_table\_struct \*wait);

unsigned int mask;

poll\_wait(filp,&wq,wait);

poll\_wait(filp,&rq,wait);

if(....){//可读

mask |= POLLIN | POLLRDNORM;

}

if(...){//可写

mask |= POLLOUT | POLLWRNORM;

}

return mask;

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

select,poll,epoll的区别

一.异步通知

异步通知是并不对读写操作进行阻塞，也不采用阻塞描述符的方式，当用户空调用系统调用后，内核立即返回，

然后接下来在内核中完成读写具体操作，操作完成之后，通过给用户程序发送信号(SIGIO)的方式通知其操作完成

1.异步通知的实现

（1）用户程序

捕获SIGIO信号,信号处理函数中完成具体的操作

指定当前进程为文件属主

fcntl(fd, F\_SETOWN, getpid());

为文件描述符添加FASYNC标志

oflags = fcntl(fd, F\_GETFL);

fcntl(fd, F\_SETFL, oflags | FASYNC);

（2）内核驱动

实现操作函数集合中的fasync接口

该函数内部调用fasync\_helper,最终会发送信号到应用程序

在条件满足之后，调用kill\_fasync函数进行通知

一.中断

1.ARM中断相关

中断有优先级

中断时异常的一种，通过异常向量表来处理

处理中断的时候要注意保护线程和恢复现场

执行实现写好的中断处理程序

2.内核中的中断

中断属于一种资源，使用前必须先申请

申请中断函数：

request\_irq

释放中断：

free\_irq

关于中断的大部分机制内核中已经实现，我们你只需要提供要申请的中断号，编写中断处理函数，设置触发方式....

中断处理函数注意事项：

（1）中断处理的时间越快越好

（2）中断处理处于内核空间，不属于任何进程，也不参与任务调度

（3）中断中原则上不允许和用户空间进行数据交互，如果一定要交互需要使用系统调用

（4）不能在中断处理函数中调用引起睡眠的函数 copy\_xxx kmalloc...

3.以按键为例实现中断程序

脚：GPIOA28 GPIOB9 GPIOB30 GPIOB31

按下低电平，松开高电平

4.中断的顶半部，底半部机制

理想中的中断处理希望越快越好，在某些情形下无法达到这个要求

比如网卡接收和发送该数据，网卡对数据的处理比较占时间，网卡数据收发用中断实现

Linux操作系统为了解决这种状况，对中断采用顶半部和顶半部的实现方法

起始就是将中断处理程序分为2个部分：顶半部和底半部

顶半部:就是原来的中断处理函数,用来处理比较紧急，必须立即处理的事务，比如网卡将数据从缓存拷贝到主存的过程

这个过程不可中断，一定要在顶半部中登记底半部，CPU在合适的时候(空闲)去执行底半部的内容

底半部:做一些不紧急，相对比较耗时的工作，比如网卡将数据包交给协议层解析的过程

如何实现底半部？顶半部和底半部如何关联？

tasklet

工作队列

软中断

5.tasklet使用

（1）数据结构

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

struct tasklet\_struct

{

struct tasklet\_struct \*next;

unsigned long state;

atomic\_t count;

void (\*func)(unsigned long);//tasklet处理函数

unsigned long data;//传递给处理函数的参数

};

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

（2）如何使用

分配初始化：

struct tasklet\_struct mytasklet;

tasklet\_init(&mytasklet,tasklet\_func,data);

或者：

DECLARE\_TASKLET(mytasklet,tasklet\_func,data,);

在顶半部(中断处理函数)中登记tasklet

tasklet\_schedule((&mytasklet);

注:tasklet还是处于中断上下文，不能睡眠

如果在底半部中需要睡眠，不能使用tasklet，需要使用工作队列

6.工作队列的使用

（1）数据结构

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

//工作队列

struct work\_struct {

atomic\_long\_t data;

struct list\_head entry;

work\_func\_t func;//工作队列处理函数

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

//延时工作队列

struct delayed\_work {

struct work\_struct work;

struct timer\_list timer;

};

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

（2）使用方法

分配初始化：

struct work\_struct mywork;

struct delayed\_work mydwork;

INIT\_WORK(&mywork,mywork\_func):

INIT\_DELAYED\_WORK(&mydwork,mydwork\_func):

在顶半部登记工作队列：

schedule\_work(&mywork);//CPU选择在合适的时候执行处理处理函数

schedule\_delayed\_work((&mywork,延时时间);//CPU在指定的时间之后才执行处理函数

工作和延时工作处于进程上下文，使用系统默认内核线程来执行，如果系统所有的工作和延时工作都交给默认的内核线程

加重默认内核线程的负担，为了避免这种情况，我们可以创建自己的内核线程去执行自己工作

如何实现：

（1）分配初始化工作和延时工作

（2）创建自己的工作队列和内核线程

struct workqueue\_struct \*xxx(工作队列指针) = create\_workqueue(线程名);

（3）在中断处理函数中登记自己的工作和延时工作

queue\_work(自己的工作队列,自己的工作);

queue\_delayed\_work(自己的工作队列,自己的延时工作，延时);

二.内核中的时间流

1.相关概念

tick:是内核中的心跳时钟，每一次时钟中断完成系统和时间相关工作

1/HZ = 10ms

HZ:用于设置硬件定时器的工作频率

ARM:HZ=100,也就是1秒钟发生100次时钟中断，间隔=10ms

jiffies:内核32位全局变量，用来记录自开机以来系统经过了多少次时钟中断，内核通常使用该变量来表示时间

2.内核中提供了一系列和jiffies相关时间处理函数：

3.内核定时器

内核中提供了定时器结构，可以用于定时

（1）数据结构

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

struct timer\_list {

/\*

\* All fields that change during normal runtime grouped to the

\* same cacheline

\*/

struct list\_head entry;

unsigned long expires;//超时时间，超时时间点的jiffies值

struct tvec\_base \*base;

void (\*function)(unsigned long);//超时处理函数

unsigned long data;//传递给超时处理函数的参数

int slack;

};

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

（2）如何使用

分配内核定时器：

struct timer\_list my\_timer;

初始化定时器：

init\_timer(&mytimer);

mytimer.expires = jiffies + 5\*HZ;//5s之后的时间,通常在启动定时器之前设置

mytimer.function = xxx\_func;

mytimer.data = xxxx;

向内核添加启动定时器：

add\_timer(&mytimer);

不再使用删除定时器：

del\_timer(&mytimer)

修改定时器:

mod\_timer(&mytimer,expires);

= del\_timer + timer.expires=xxx +add\_timer

（3）使用内核定时器对按键消抖

按键按下或者松开的时候，不是一个理想的跳变沿，会有多次抖动，抖动时间<10ms,

每一次跳变会触发中断，如果触发中断后10ms内有下一次中断发生，说明这是一次抖动，否则才是真实按键事件

为了去掉这种抖动，我们可以在中断处理函数中配置定时器，超时时间设为10ms,在超时处理函数中处理真实按键事件，去除按键抖动

4.内核中如何延时

<100ms

ndelay

udelay

mdelay

>100ms

msleep

msleep\_interruptible

ssleep

一.Linux内核中内存相关

1.Linux内核物理内存管理

内核在启动时，会为每一个物理内存页分配一个struct page结构，如果在内核中一个page结构代表一个物理内存页

2.内核虚拟地址

内核中使用虚拟地址访问物理内存，划分：

用户空间:0x00000000 - 0xbfffffff

内核空间:0xc0000000 - 0xffffffff

内核空间的虚拟地址和物理地址之间的关系通常是一个线性关系,在系统启动时已经实现了虚拟地址和物理地址的映射

用户空间的虚拟内存并不会立即映射，而是动态进行映射

3.MMU

虚拟地址转换成物理地址

检查访问权限

控制cache

4.内核虚拟地址划分

实际的物理内存大于1G，需要对内核的虚拟地址进行划分

x86:

低端内存(直接内存映射区)：

虚拟地址和物理地址还是线性关系

高端内存：起始地址有物理内存大小决定

动态内存映射区

永久内存映射区

固定内存映射区

ARM：

[ 0.000000] Virtual kernel memory layout:

[ 0.000000] vector : 0xffff0000 - 0xffff1000 ( 4 kB)

[ 0.000000] fixmap : 0xfff00000 - 0xfffe0000 ( 896 kB)

[ 0.000000] vmalloc : 0xef800000 - 0xfee00000 ( 246 MB)//动态内存映射区

[ 0.000000] lowmem : 0xc0000000 - 0xef600000 ( 758 MB)//低端内存

[ 0.000000] pkmap : 0xbfe00000 - 0xc0000000 ( 2 MB)

[ 0.000000] modules : 0xbf000000 - 0xbfe00000 ( 14 MB)

[ 0.000000] .text : 0xc0008000 - 0xc0902654 (9194 kB)

[ 0.000000] .init : 0xc0903000 - 0xc093d100 ( 233 kB)

[ 0.000000] .data : 0xc093e000 - 0xc09ca8e0 ( 563 kB)

[ 0.000000] .bss : 0xc09ca904 - 0xc0babad8 (1925 kB)

5.内核中申请分配内存的方法：

kmalloc/kfree (128K--32M)

\_\_get\_free\_pages/free\_pages

以上两种分配方法得到的都是低端内存，在物理上和虚拟上都是连续的

vmalloc/vfree

分配在动态内存映射区，在物理上不保证连续，用于分配空间较大的内存

6.IO内存映射

内核中我们能够直接访问的是虚拟地址，而操作硬件，需要访问硬件寄存器

硬件寄存器地址属于物理地址，不能直接访问，需要先将寄存器物理地址映射到虚拟地址空间才能访问

这种将寄存器物理地址映射到虚拟地址就叫做IO内存映射，这种映射属于动态映射

映射方法：

映射虚拟地址 = ioremap(物理地址,长度);

映射完成之后，访问对应的虚拟地址就相当于访问物理地址

不再使用可以解除映射：

iounmap(虚拟地址);

可以使用访问指针的语法访问映射的虚拟地址，这种方法不是标准用法

标准访问放应该使用内核中提供的访问函数

读:

readb/readw/readl --------8/16/32 位读

writeb/writew/writel

二.设备驱动模型

1.内核分离思想

之前写的硬件驱动，如果硬件平台(连接)发生变化(GPIOE13---->GPIOB10)

之前的驱动就会失效，需要重新修改编写，需要修改得部分都是和硬件相关的代码，软件框架无需做任何改变

所以内核为了解决可移植性问题，将驱动中软件部分和硬件部分分离开来，这样移植的时候就只需要修改其中硬件部分

内核使用什么机制实现软硬件分离？

设备-总线-驱动模型

以平台设备总线驱动模型(platform)为例介绍设备-总线-驱动模型工作机制

2.platform

在内核中，应实现一条虚拟总线platform\_bus\_type

这条总线上维护两个链表:dev链表 drv链表

dev链表中的硬件信息节点类型:struct platform\_device

drv链表中的软件信息节点类型:struct platform\_driver

只有软件节点和硬件节点合在一起才是一个完整的驱动，如何匹配？

匹配过程发生在往链表中添加节点的过程：

如果往dev链表中添加一个硬件节点，内核就会去遍历drv链表，取出drv链表每一个节点和加入的硬件节点进行匹配(id 名字)

如果匹配成功，证明硬件节点找到了对应的软件节点，就会去调用软件节点中的probe函数，并且将硬件节点的地址传递给函数

probe函数完成的事情由开发人员决定

如果往drv链表中添加一个软件节点，内核就会去遍历dev链表，取出dev链表每一个节点和加入的软件节点进行匹配(id 名字)

如果匹配成功，证明软件节点找到了对应的硬件节点，就会去调用软件节点中的probe函数，并且将硬件节点的地址传递给函数

probe函数完成的事情由开发人员决定

节点匹配顺序：

(1)使用dev中设备树传递的节点信息和drv中匹配列表进行匹配

(2)使用dev中的name和drv节点中的id\_table(名字数组)进行匹配

(3)使用dev中的name和drv节点中的name进行匹配

3.使用

其中匹配管理机制内核全部已经实现，我们开发只需要添加struct paltform\_device和struct platform\_driver就可以了

（1）如何添加platform\_device

a.分配初始化 resource结构，填充硬件相关信息

struct resource led\_res[] = {

[0] = {

.start = 硬件信息起始数据;//起始地址,起始端口，起始中断号

.end = 硬件信息结束数据;

.flags = 硬件信息类型;//IO地址信息 IORESOURCE\_MEM 中断信息 IORESOURCE\_IRQ...

}

};

b.分配初始化platform\_device

struct platform\_device led\_dev = {

.name = "myled",//和drv中名字一样

.id = -1,//用来区分同名资源，如果只有一个资源，=-1

.resource = led\_res,

.num\_resources = ARRAY\_SIZE(led\_res),

.dev{

.release = xxxx,//删除硬件节点自动调用函数

}

};

c.调用platform\_device\_register向内核注册

（2）如何使用platform\_driver

a.分配初始化platform\_driver

struct platform\_driver led\_drv = {

.id\_table = //名字数组首地址，数组以""结束,用于匹配，优先级高于name

.driver = {

.name = "myled",//和dev中名字一样

}

.probe = xxx\_probe,//匹配成功调用的函数

.remove = xxx\_remove,//解除匹配调用的函数

.suspend = xxxx\,//电源管理

。。。。

};

b.调用platform\_driver\_register向内核注册

一.Linux内核中内存相关

1.Linux内核物理内存管理

内核在启动时，会为每一个物理内存页分配一个struct page结构，如果在内核中一个page结构代表一个物理内存页

2.内核虚拟地址

内核中使用虚拟地址访问物理内存，划分：

用户空间:0x00000000 - 0xbfffffff

内核空间:0xc0000000 - 0xffffffff

内核空间的虚拟地址和物理地址之间的关系通常是一个线性关系,在系统启动时已经实现了虚拟地址和物理地址的映射

用户空间的虚拟内存并不会立即映射，而是动态进行映射

3.MMU

虚拟地址转换成物理地址

检查访问权限

控制cache

4.内核虚拟地址划分

实际的物理内存大于1G，需要对内核的虚拟地址进行划分

x86:

低端内存(直接内存映射区)：

虚拟地址和物理地址还是线性关系

高端内存：起始地址有物理内存大小决定

动态内存映射区

永久内存映射区

固定内存映射区

ARM：

[ 0.000000] Virtual kernel memory layout:

[ 0.000000] vector : 0xffff0000 - 0xffff1000 ( 4 kB)

[ 0.000000] fixmap : 0xfff00000 - 0xfffe0000 ( 896 kB)

[ 0.000000] vmalloc : 0xef800000 - 0xfee00000 ( 246 MB)//动态内存映射区

[ 0.000000] lowmem : 0xc0000000 - 0xef600000 ( 758 MB)//低端内存

[ 0.000000] pkmap : 0xbfe00000 - 0xc0000000 ( 2 MB)

[ 0.000000] modules : 0xbf000000 - 0xbfe00000 ( 14 MB)

[ 0.000000] .text : 0xc0008000 - 0xc0902654 (9194 kB)

[ 0.000000] .init : 0xc0903000 - 0xc093d100 ( 233 kB)

[ 0.000000] .data : 0xc093e000 - 0xc09ca8e0 ( 563 kB)

[ 0.000000] .bss : 0xc09ca904 - 0xc0babad8 (1925 kB)

5.内核中申请分配内存的方法：

kmalloc/kfree (128K--32M)

\_\_get\_free\_pages/free\_pages

以上两种分配方法得到的都是低端内存，在物理上和虚拟上都是连续的

vmalloc/vfree

分配在动态内存映射区，在物理上不保证连续，用于分配空间较大的内存

6.IO内存映射

内核中我们能够直接访问的是虚拟地址，而操作硬件，需要访问硬件寄存器

硬件寄存器地址属于物理地址，不能直接访问，需要先将寄存器物理地址映射到虚拟地址空间才能访问

这种将寄存器物理地址映射到虚拟地址就叫做IO内存映射，这种映射属于动态映射

映射方法：

映射虚拟地址 = ioremap(物理地址,长度);

映射完成之后，访问对应的虚拟地址就相当于访问物理地址

不再使用可以解除映射：

iounmap(虚拟地址);

可以使用访问指针的语法访问映射的虚拟地址，这种方法不是标准用法

标准访问放应该使用内核中提供的访问函数

读:

readb/readw/readl --------8/16/32 位读

writeb/writew/writel

二.设备驱动模型

1.内核分离思想

之前写的硬件驱动，如果硬件平台(连接)发生变化(GPIOE13---->GPIOB10)

之前的驱动就会失效，需要重新修改编写，需要修改得部分都是和硬件相关的代码，软件框架无需做任何改变

所以内核为了解决可移植性问题，将驱动中软件部分和硬件部分分离开来，这样移植的时候就只需要修改其中硬件部分

内核使用什么机制实现软硬件分离？

设备-总线-驱动模型

以平台设备总线驱动模型(platform)为例介绍设备-总线-驱动模型工作机制

2.platform

在内核中，应实现一条虚拟总线platform\_bus\_type

这条总线上维护两个链表:dev链表 drv链表

dev链表中的硬件信息节点类型:struct platform\_device

drv链表中的软件信息节点类型:struct platform\_driver

只有软件节点和硬件节点合在一起才是一个完整的驱动，如何匹配？

匹配过程发生在往链表中添加节点的过程：

如果往dev链表中添加一个硬件节点，内核就会去遍历drv链表，取出drv链表每一个节点和加入的硬件节点进行匹配(id 名字)

如果匹配成功，证明硬件节点找到了对应的软件节点，就会去调用软件节点中的probe函数，并且将硬件节点的地址传递给函数

probe函数完成的事情由开发人员决定

如果往drv链表中添加一个软件节点，内核就会去遍历dev链表，取出dev链表每一个节点和加入的软件节点进行匹配(id 名字)

如果匹配成功，证明软件节点找到了对应的硬件节点，就会去调用软件节点中的probe函数，并且将硬件节点的地址传递给函数

probe函数完成的事情由开发人员决定

节点匹配顺序：

(1)使用dev中设备树传递的节点信息和drv中匹配列表进行匹配

(2)使用dev中的name和drv节点中的id\_table(名字数组)进行匹配

(3)使用dev中的name和drv节点中的name进行匹配

3.使用

其中匹配管理机制内核全部已经实现，我们开发只需要添加struct paltform\_device和struct platform\_driver就可以了

（1）如何添加platform\_device

a.分配初始化 resource结构，填充硬件相关信息

struct resource led\_res[] = {

[0] = {

.start = 硬件信息起始数据;//起始地址,起始端口，起始中断号

.end = 硬件信息结束数据;

.flags = 硬件信息类型;//IO地址信息 IORESOURCE\_MEM 中断信息 IORESOURCE\_IRQ...

}

};

b.分配初始化platform\_device

struct platform\_device led\_dev = {

.name = "myled",//和drv中名字一样

.id = -1,//用来区分同名资源，如果只有一个资源，=-1

.resource = led\_res,

.num\_resources = ARRAY\_SIZE(led\_res),

.dev{

.release = xxxx,//删除硬件节点自动调用函数

}

};

c.调用platform\_device\_register向内核注册

（2）如何使用platform\_driver

a.分配初始化platform\_driver

struct platform\_driver led\_drv = {

.id\_table = //名字数组首地址，数组以""结束,用于匹配，优先级高于name

.driver = {

.name = "myled",//和dev中名字一样

}

.probe = xxx\_probe,//匹配成功调用的函数

.remove = xxx\_remove,//解除匹配调用的函数

.suspend = xxxx\,//电源管理

。。。。

};

b.调用platform\_driver\_register向内核注册

一.GEC6818外设编程

1.GPIO

2.ADC

AD/ DA指的是 模拟量和数字量之前的转换

模拟量就是现实生活中各种有单位的数据(电压 电流 电阻 重量 压强....)

计算机无法表示模拟数据，只识别数字数据

（1）A/D转换器的类型

积分型：

速度慢，电路简单

逐次比较型：

速度快于积分型，电路复杂度高于积分型

并行/串行比较型

使用多个比较器同时开始比较，速度最快，电路更复杂

（2）6818上的ADC

一.看门狗

二.PWM

2.使用混杂设备模型实现pwm驱动

混杂设备是一类无法归类的设备

混杂设备是一种字符设备，主设备号固定是10

注册：

分配初始化 struct miscdevice结构

struct miscdevice {

int minor;//MISC\_DYNAMIC\_MINOR

const char \*name;//dev name

const struct file\_operations \*fops;//f\_ops

struct list\_head list;

struct device \*parent;

struct device \*this\_device;

const char \*nodename;

umode\_t mode;

};

调用misc\_register注册 ----------- 自动创建设备文件，设备类,cdev,设备号已经申请完成

注销：

调用misc\_deregister注销

一.输入子系统

1.Linux系统中的输入设备的驱动实现，其中有很多内容是相似的，Linux内核将驱动中固定的部分实现好了

放在Linux内核中，我们使用时只需要实现驱动中不同的部分

输入子系统对应的设备文件/dev/events0 1 2.....

2.input子系统可以用来实现的设备

鼠标 键盘 触摸屏 游戏摇杆....

3.Linux子系统在内核中的框架

app: read write open.....

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

input\_handler层 ------------------ drivers/input/evdev.c

(1)实现了用户访问驱动的统一的接口

(2)创建设备文件

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

input核心层(input handler) ---------- drivers/input/input.c

(1)提供注册input\_handler 的接口

(2)对下提供了驱动注册硬件信息到核心层的统一方法(input\_dev)

(3)注册了设备类和cdev

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

设备驱动层:关注和硬件以及上报相关信息

4.input子系统驱动实现

（1）分配input\_dev结构

struct input\_dev \*btn\_dev = input\_allocate\_device();

（2）初始化input\_dev

指定发生事件类型：

.evbit =

#define EV\_SYN 0x00 //同步事件

#define EV\_KEY 0x01 //按键事件

#define EV\_REL 0x02 //相对坐标事件

#define EV\_ABS 0x03 //绝对坐标事件

指定键值/坐标类型....

.keybit =

上报按键事件中包含哪些键值

.relbit =

上报相对坐标事件中包含哪些坐标

.absbit =

上报绝对坐标事件中包含哪些坐标

......

注:使用set\_bit函数刦设置

（3）调用input\_register\_device将input\_dev注册到内核

（4）当输入设备某些操作发生之后，调用input子系统提供的接口(input\_event函数)，通知input子系统上报

input\_event函数实现两个功能：

唤醒被阻塞的读进程

提供上报数据内容

5.input子系统上报信息

是一个 struct input\_event结构

struct input\_event {

struct timeval time;//时间戳

\_\_u16 type;//事件类型

\_\_u16 code;//键值/坐标类型...

\_\_s32 value;//按键状态/坐标值

};

测试方法1：

hexdump 设备文件名 ------------- 读设备文件，将读到信息以16进制格式打印出来

编号 事件戳 事件类型 键值 按键状态低位 高位

0000080 9003 54a4 2688 0001 0001 0005 0001 0000

0000090 9003 54a4 2690 0001 0000 0000 0000 0000

00000a0 9003 54a4 e864 000b 0001 0005 0000 0000

00000b0 9003 54a4 e86a 000b 0000 0000 0000 0000

测试方法2：

编写访问/dev/event1的代码

二.input子系统内核源码分析

1.input核心层:drivers/input/input.c

subsys\_initcall(input\_init); ----------- input\_init函数在系统初始化被调用

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

static int \_\_init input\_init(void)：

err = register\_chrdev(INPUT\_MAJOR, "input", &input\_fops);//注册字符设备驱动

对应操作函数集合：

static const struct file\_operations input\_fops = {

.owner = THIS\_MODULE,

.open = input\_open\_file,

.llseek = noop\_llseek,

};

问题:其中只实现了open函数，其他的函数接口哪去了？------------ input\_open\_file

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

static int input\_open\_file(struct inode \*inode, struct file \*file):

struct input\_handler \*handler;

const struct file\_operations \*old\_fops, \*new\_fops = NULL;

//根据设备号从input\_table数组中取出一项input\_handler

handler = input\_table[iminor(inode) >> 5];

if (handler)

new\_fops = fops\_get(handler->fops);//将handler结构中的fops赋值给new\_fops

old\_fops = file->f\_op;

file->f\_op = new\_fops;//将new\_fops赋值给了file结构中的f\_op,以后访问得失handler的fops

err = new\_fops->open(inode, file);//同时调用handler的open函数

问题：

input\_handler何时注册到内核的？input\_table在哪填充的

input\_handler通过input\_register\_handler函数注册，该函数在evdev\_init函数中被调用

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

int input\_register\_handler(struct input\_handler \*handler)：

//在input\_table数组通过设备号找到对应位置，将input\_handler放进去

if (handler->fops != NULL) {

if (input\_table[handler->minor >> 5]) {

retval = -EBUSY;

goto out;

}

input\_table[handler->minor >> 5] = handler;

}

//将handler添加内核链表input\_handler\_list

//遍历另外一个链表input\_dev\_list，取出其中每一个节点(input\_dev)和input\_handler进行匹配

list\_add\_tail(&handler->node, &input\_handler\_list);

list\_for\_each\_entry(dev, &input\_dev\_list, node)

input\_attach\_handler(dev, handler);

如果匹配成功，调用handler中的connect函数

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

input\_handler的实现和注册可以自己实现，也可以使用内核中提供的标准框架 ---------- drivers/input/evdev.c

struct input\_handler结构：

static struct input\_handler evdev\_handler = {

.event = evdev\_event,//调用input\_event会调用的函数

.connect = evdev\_connect,//匹配成功会调用的函数

.disconnect = evdev\_disconnect,

.fops = &evdev\_fops,//操作函数集合

.minor = EVDEV\_MINOR\_BASE,

.name = "evdev",

.id\_table = evdev\_ids,

};

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

static int evdev\_connect(struct input\_handler \*handler, struct input\_dev \*dev,

const struct input\_device\_id \*id):

//构造evdev结构

evdev = kzalloc(sizeof(struct evdev), GFP\_KERNEL);

//初始化等待队列

init\_waitqueue\_head(&evdev->wait);

//将handler和dev记录到evdev中的handle结构中

evdev->handle.dev = input\_get\_device(dev);

evdev->handle.name = dev\_name(&evdev->dev);

evdev->handle.handler = handler;

evdev->handle.private = evdev;

//注册input\_handle,就是将d\_node加入到dev->h\_list

error = input\_register\_handle(&evdev->handle);

//将evdev放入到evdev\_table数组中

error = evdev\_install\_chrdev(evdev);

创建设备文件

device\_initialize(&evdev->dev);

....

error = device\_add(&evdev->dev);

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

open函数实现：

static int evdev\_open(struct inode \*inode, struct file \*file)：

struct evdev\_client \*client;

//从evdev\_table数组中取出evdev

evdev = evdev\_table[i];

//为cilent分配空间

client = kzalloc(sizeof(struct evdev\_client) +

bufsize \* sizeof(struct input\_event),

GFP\_KERNEL);

//将evdev记录到cilent

client->evdev = evdev;

//将cilent记录到file中，方便其他的函数接口使用

file->private\_data = client;

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

read函数实现：

static ssize\_t evdev\_read(struct file \*file, char \_\_user \*buffer,

size\_t count, loff\_t \*ppos)：

//取出file中evdev

struct evdev\_client \*client = file->private\_data;

struct evdev \*evdev = client->evdev;

struct input\_event event;

//阻塞方式

if (!(file->f\_flags & O\_NONBLOCK)) {

//阻塞，等待唤醒

retval = wait\_event\_interruptible(evdev->wait,

client->packet\_head != client->tail ||

!evdev->exist);

if (retval)

return retval;

}

//拷贝数据(input\_event到用户空间)

while (retval + input\_event\_size() <= count &&

evdev\_fetch\_next\_event(client, &event)) {//从cilent中获取event数据

if (input\_event\_to\_user(buffer + retval, &event))//将event数据拷贝到用户空间

return -EFAULT;

retval += input\_event\_size();

}

问题:cilent中的数据在哪填充的？等待队列在哪里唤醒的

在input\_event函数

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

input\_event--- input\_handle\_event:

input\_pass\_event(dev, type, code, value);//调用input\_hanlder中event函数

input\_pass\_event:

//通过dev找到handle,找到handler,调用handler中的event函数

list\_for\_each\_entry\_rcu(handle, &dev->h\_list, d\_node) {

if (!handle->open)

continue;

handler = handle->handler;

if (!handler->filter) {

if (filtered)

break;

handler->event(handle, type, code, value);//调用evdev\_event

evdev\_event:

event.type = type;

event.code = code;

event.value = value;

evdev\_pass\_event(client, &event, time\_mono, time\_real);//将event数据填充cilent中buffer

//如果是同步信号，唤醒阻塞的进程

if (type == EV\_SYN && code == SYN\_REPORT) {

evdev->hw\_ts\_sec = -1;

evdev->hw\_ts\_nsec = -1;

wake\_up\_interruptible(&evdev->wait);

}

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

input\_dev通过input\_register\_device函数注册

int input\_register\_device(struct input\_dev \*dev)：

//将dev添加内核链表input\_dev\_list

//遍历另外一个链表input\_handler\_list，取出其中每一个节点(input\_handler)和input\_dev进行匹配

list\_add\_tail(&dev->node, &input\_dev\_list);

list\_for\_each\_entry(handler, &input\_handler\_list, node)

input\_attach\_handler(dev, handler);

二.IIC

1.

S5P6818内部集成了IIC控制器,IIC总线上传输数据，所需时序要求，都可以通过控制器来实现

控制器发起的时序，有些是固定的，有些是可变的

固定的:start stop ack R/W

可变的:从设备地址，读写时的数据

可变的数据来自于从设备(从设备芯片手册)

我们将这些信息交给IIC控制器，开始控制就可以完成对应的传输

2.

Linux内核基于以上信息，实现IIC驱动：

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

IIC总线驱动：

管理的硬件是IIC控制器，关心如何进行数据传输，不关心数据具体含义，总线驱动通常由芯片厂家实现，我们只需要在内核中配置玄宗即可

make menuconfig

Device Drivers---------->

I2C support-------->

I2C Hardware Bus support --->

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

IIC设备驱动：

操作的对象的IIC从设备，关心从设备地址，关心从设备传输数据具体含义，不关心如何进行传输

传输依靠IIC总线驱动，我们只需要和总线实现数据交互即可

3.IIC驱动框架

app:read write open...........

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

IIC设备驱动:关心传输数据具体含义，不关心如何进行传输

xxxopen xxx\_read xxx\_write....

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

内核提供了统一的借口实现设备驱动和总线驱动的数据交互:

i2c\_transfer();

或者使用smbus接口

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

IIC总线驱动:关心如何进行数据传输，不关心数据具体含义

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

IIC控制器<===>从设备

4.IIC设备驱动实现要使用 设备 总线 驱动模型

内核中实现了一条IIC虚拟总线:i2c\_bus\_type

总线上挂着两条链表，dev链表和drv链表

dev链表中的节点保存是硬件信息，对应数据结构:struct i2c\_client

drv链表中的节点保存是软件信息，对应的数据结构:struct i2c\_driver

每当往其中一条链表添加节点的时候，就会遍历另外一条链表，取出每一个节点和新节点进行匹配(client.name driver.id\_table.name)

匹配成功调用drv节点中的probe函数，并且将client节点的首地址传递给该函数

5.如何往内核中注册i2c\_drvier

(1)分配初始化i2c\_driver结构

struct i2c\_driver eeprom\_drv = {

.id\_table = 名字;

.probe = xxx\_probe;

.remove = xxx\_remove,

};

(2)调用i2c\_add\_driver向内核添加

6.如何往内核中注册i2c\_client

i2c\_client的注册都是由内核完成，我们只需要对i2c\_board\_info进行分配初始化

内核根据i2c\_board\_info的信息来填充i2c\_client

(1)分配初始化i2c\_board\_info

其中包含name和从设备地址

.type = xxx,

.addr = xxx,

(2)将i2c\_board\_info添加到内核

i2c\_register\_board\_info

1.at24lc04编程

设备地址：

1010000 = 0x50

写时序：

读时序：

二.mma8653fc传感器

三向加速度传感器

1.从设备地址

0011101 -------- 0x1d

2.读写时序

读操作：

写操作：

3.相关寄存器

三.其他驱动

SPI驱动

USB驱动

frambuffer驱动

flash驱动

块设备驱动

网卡驱动

一.项目

使用6818开发板实现智能家居控制系统

1.用户界面

使用QT在开发板上实现操作界面

2.在开发板上实现各个硬件的驱动

LED BEEP 烟雾传感器

超声波

红外

按键 eeprom 3-AXIES

串口蓝牙 RFID

注:编写模块驱动指定和开发板内核匹配的内核源码，通过网络/串口将.ko文件到开发板，再insmod安装

和其他硬件通信(32开发板),控制硬件和交换数据

恢复开发板为之前的系统的方法：

set bootcmd ext4load mmc 2:1 0x48000000 uImage\;bootm 0x48000000

set bootargs lcd=at070tn92 tp=gslx680-linux root=/dev/mmcblk0p2 rw rootfstype=ext4

save

修改内核源代码，使内核版本说明和开发板上内核版本一致(如果不一致加载模块报错)

include/linux/vermagic.h

一.块设备驱动

1.块设备驱动层次结构

2.Linux内核的块设备驱动

（1）块设备设备号

（2）块设备结构

抽象:

分区信息：

实际块设备：gendisk

（3）如何构造一个gendisk

a.分配初始化gendisk

b.添加gendisk到内核

c.操作gendisk函数

d.设置gendsik容量

c.操作函数集合(并没有具体硬件操作)

（4）gendisk中的请求队列

a.请求队列结构

b.请求结构

c.构造请求队列

两种构造方法的区别：

请求相关处理函数：

（5）request中的bio结构

a.bio结构

b.bio相关操作函数

c.bio中核心结构是bio\_vec数组，IO操作传输数据就在bio\_cev中

3.块设备驱动的总体结构

二.网络设备驱动

1.sk\_buff结构

2.网卡驱动层次结构

（3）内嵌汇编

五.进程

1.概念

进程就是程序在内存中动态的执行过程

2.进程和程序的异同

程序是一个静态的文件

进程是一个动态执行的过程

进程有创建，运行，消亡的过程

进程是一个独立可调度的任务(宏观并行，微观串行)

进程是资源管理的最小单位

每个进程都是自己独立的4G进程虚拟地址空间，其中3G(0-3)是用户空间，1G(3-4)是内核空间,他们相互独立

进程在内存中也是分段存储的，存储顺序和程序(ELF)是一样

3.进程的状态

就绪状态：准备完毕，等待调度器调度

运行状态：进程正在CPU运行

暂停状态：进程收到暂停信号，停止运行

睡眠状态：进程无法获取IO资源，进入睡眠

僵尸状态：进程已经结束，资源没有被回收

Linux 网络编程学习笔记

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

domain(域)确定通信的特性

域 描述

AF\_INET IPV4

AF\_INET6 IPV6

AF\_UNIX UNIX域

AF\_UPSPEC 未指定

Type确定套接字的类型

类型 描述

SOCK\_DGRAM UDP， 固长，无连接的不可靠的数据报文

SOCK\_RAW IP协议的数据报接口

SOCK\_STREAM TCP,有序可靠双向面向连接的字节流

SOCK\_SEQPACKET 固定长度的有序的可靠的面向连接的报文传递

Protocol:通常为0表示给定的域和套接字类型默认选择协议

协议 描述

IPPROTO\_IP IPV4网际协议

IPPROTO\_IPv6 IPV6网际协议

IPPROTO\_ICMP ICMP（internet control message protocol）协议

IPPROTO\_RAM 原始IP数据包协议

IPPROTO\_TCP

IPPROTO\_UDP

#include <sys/socket.h>

int shutdown (int sockfd, int how);

how:

SHUT\_RD(关闭读端)，无法从套接字读取数据

SHUT\_WR（关闭写端），无法使用套接字发送数据

SHUT\_RDWR， 关闭读写两端

#include <arpa/inet.h>

uint32\_t htonl( \_t hostint 32) 返回值：以网络字节序表示32位整数

uint16\_t htons(uint16\_t hostint 16) 返回值：以网络字节序表示16位整数

uint32\_t ntohl(uint32\_t netint 32) 返回值：以主机字节序表示32位整数

uint16\_t ntohs(uint16\_t netint 16) 返回值：以主机字节序表示16位整数

地址格式(允许自由添加成员并且定义sa\_data)：

struct sockaddr{

sa\_family\_t sa\_family; /\*address family\*/

char sa\_data[]; /\*\*variable-length address\*/

}

linux 中的格式

struct sockaddr{

sa\_family\_t sa\_family; /\*address family\*/

char sa\_data[14]; /\*\*variable-length address\*/

}

因特网定义在<netinet/in.h>头文件中。IPV4因特网（AF\_INET）套接字

结构用sockaddr\_in；

struct sockaddr\_in{

sa\_family\_t sin\_family; /\*address family\*/

in\_port\_t sin\_port; /\*port number\*/

struct in\_addr sin\_addr ;/\*IPV4 address\*/

};

struct in\_addr {

in\_addr\_t s\_addr; /\*IPV4 addrsss\*/

};

IPV6因特网（AF\_INET6）套接字

结构用sockaddr\_in；

struct sockaddr\_in6{

sa\_family\_t sin6\_family; /\*address family\*/

in\_port\_t sin6\_port; /\*port number\*/

struct in6\_addr sin6\_addr ;/\*IPV4 address\*/

uint32\_t sin6\_folwinfo; /\*traffic class and flow info\*/

uint32\_t sin6\_scope\_id; /\*set of interfaces for scope\*/

};

struct in6\_addr {

uint8\_t s6\_addr[16]; /\*IPV6 addrsss\*/

};

inet\_addr,inet\_ntoa二进制地址格式转换为点分十进制字符表示

(只支持IPV4)

inet\_ntop , inet\_pton (同时支持IPV4和IPV6)

#include <arpa/inet.h>

const char\* inet\_ntop(int domain, const void \*restrict addr,

char \*restrict str, socklen\_t size);

成功：地址字符串指针；出错，返回NULL；

size ； 保存文本字符串的缓冲区（str）大小

常用常数INET\_ADDRSTRLEN/INET6\_ADDRSTRLEN

const char\* inet\_pton(int domain, const void \*restrict str,

void \*restrict addr);

成功：返回 1； 格式无效返回 -1； 出错，返回-1

inet\_ntop 网络字节序的二进制地址转换成文本字符串格式

inet\_pton 点分十进制转换为网络字节序的二进制

**返回地址为主机字节序**

#include <netdb.h>

struct hostent \*gethostent(void); //成功返回指针， 出错返回NULL

void sethostent(int stayopen);

void endhostent(void);

struct hostent{

char **h\_name; /**\*name of host \*/

char \*\*h\_aliases; /\*pointer to alternate(备用) host name array\*/

int h\_addrtype; /\*address type\*/

int h\_length; /\*length in byte of address\*/

char \*\*h\_addr\_list; /\*pointer to array of network addresses\*/

};

**返回地址为网络字节序**

struct netent \*getnetbyaddr (uint32\_t net, int type);

struct netent \*getnetbyname(const char \*name);

struct netent \*getnetent(void);

void setnetent(int stayopen);

void endentent(void);

struct netent{

char \***n\_name; /**\*network name\*/

char \*\*n\_aliases; /\* pointer to alternate(备用) network name array \*/

int n\_addrtype; /\*address type\*/

uint32\_t n\_net; /\*network number\*/}

网络编号按照网络字节序返回

struct protent \*getprotobyname(const char \*name);

struct protent \*getprotobynumber(int proto);

struct protent \*getprotoent();

void setprotent(int stayopen);

void endprotent(void);

struct protoent{

char \***p\_name; /**\*protocol name\*/

char \*\*p\_aliases; /\* pointer to alternate(备用) protocol name array \*/

int p\_proto; /\* protocol number\*/

};

服务是由地址的端口号表示，每个服务由唯一的众所周知的

端口号来决定

struct servent \*getservbyname(const char \*name);

struct servent \*getservbynumber(int proto);

struct servent \*getservent();

void setservent(int stayopen);

void endservtent(void);

struct servent{

char \***s\_name; /**\*service name\*/

char \*\*s\_aliases; /\* pointer to alternate(备用) service name array \*/

char \* s\_proto; /\* protocol number\*/

int s\_port; /\* port number\*/

};

允许一个应用程序将主机名和一个服务名映射到一个地址，或者反之

getaddrinfo函数允许一个主机名和一个服务名映射到一个地址

#include <sys/socket.h>

#include <netdb.h>

int getaddrinfo(const char \*restrict host,

const char \*restrict service,

const struct addrinfo \*restrict hint,

struct addrinfo \*\*restrict res);

成功返回 0； 出错返回非0错误码；

void freeaddrinfo(struct addrinfo \*ai);

需要主机名， 服务名，或者二者都提供。主机名可以是节点名

或者点分格式的主机地址

struct addrinfo{

int ai\_flags; /\* customize behavior\*/

int ai\_family;

int ai\_socktype;

int ai\_protocol;

socklen\_t ai\_addrlen;

struct sockaddr \*ai\_addr;

char \*ai\_canonname;

struct addrinfo **ai\_next; /**\*next in list\*/

};

如果getaddrinfo 失败，不能使用perror strerror 生产错误信息

而通过gai\_strerror将错误码转换

#include <netdb.h>

const char \*gai\_strerror(int error);

#include <sys/socket.h>

int getnameinfo(const sockaddr \*restrict addr, socklen\_t alen,

char \*restrict host, socklen\_t hostlen,

char \*restrict serice, socklen\_t servlen, int flags);

#include <sys/socket>

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t len);

成功返回0；失败返回-1

限制：

1.在进程正在运行的计算机上，指定的地址必须有效；不能指定一个其他机器地址

2.地址必须和创建套接字时的地址族所支持的格式相匹配

3.绑定的port必须不小于1024，除非是超级用户（拥有相应特权）

4.一般只能将一个套接字端点绑定到给定的地址上，尽管有些协议运行多重绑定

INADDR\_ARNY（<netinet/in.h>中定义的）

可以调用 getsockname 函数来发现绑定到套接字上的地址

int getsockname(int sockfd, struct sockaddr \*restrict addr,

socklen\_t \*restrict alenp);

成功返回0； 失败返回-1；

getpeername : 查找对方地址(前提建立对等连接)

int getpeername(int sockfd, struct sockaddr \*restrict addr,

socklen\_t \*restrict alenp);

成功返回0； 失败返回-1;

**建立连接connect（）**

#include <sys/socket.h>

int connect(int sockfd, const struct sock\_addr \*addr, socklen\_t len);

成功：返回0；失败返回 -1；

（此函数用于client端主要用于连接服务器）

connect 指数补偿算法

#include “apue.h”

#include <sys/socket.h>

#define MAXSLEEP 128

int connect\_retry(int sockfd, const struct sockaddr \*addr

socklen\_t alen)

{

int numsec;

// Try ro connect with exponential backoff.

For(numsec = 1; numsec <= MAXSLEEP; numsec <<= 1)

{

if((fd = sock(AF\_INET, SOCK\_STREAM, 0)) < 0)

return -1;

/\*之前由于connect 失败会继续使用失败socket，因此

\* 每次创建socket 如果失败再close（）；

\*/

if (connect(sockfd, addr, alen) == 0)

{

// connection accepted

return 0;

}

// delay before trying again

sockfd.close();

if (numsec <= MAXSLEEP/2)

sleep(numsec);

}

return -1;

}

服务端调用

#include <sys/socket.h>

int listen(int sockfd, int backlog);

backlog提供一个提示，提示系统该进程所要入队的未完成连接请求数

实际值由系统决定，上限由<sys/socket.h>中的SOMAXCONN(默认128)

accept获取连接请求，并且建立连接

accept(int sockfd, struct sockaddr \*restrict addr,

socklen\_t \*restrict len);

成功返回client\_socket; 出错返回 -1；

(如果没有连接请求，accept会阻塞直到一个请求到来。如果sockfd

处于非阻塞模式，accpet会返回-1， 并且将errno设置为EAGAIN或者

EWOULDDBLOCK)

服务器也可以使用poll select 来等待请求到来，在这种情况下，一个

带有等待连接请求的套接字会以可读方式出现

**用于SOCKET 传输**

**TCP使用 send/recv()或者read（），write（）；**

**UDP 使用recvfrom（），sendto（）；**

#include <sys/socket.h>

ssize\_t send(int sockfd, const void \*buf, size\_t nbytes, int flags);

成功：返回发送的字节数；若出错，返回 -1

类似于write，使用send时必须已经连接。

Flags： Discribe：

MSG\_CONFIRM 提供链路层反馈以保持地址映射有效

MSG\_DONTWAIT 勿将数据包路由出本地网络

MSG\_DONTWAIT 允许非阻塞操作==（O\_NONBLOCK）

MSG\_EOF 发送数据后关闭套接字的发送端

MSG\_EOR 如果协议支持，标记记录结束

MSG\_MORE 延迟发送数据包允许些更多数据

MSG\_NOSIGNAL 写无连接的套接字时不产生SIGPIPE信号

MSG\_OOB 如果协议支持，发送带外数据

当send成功只能说明无误发送数据到网络驱动，并不能保证对端

接收成功

对于支持报文边界协议，发生单个报文长度超过协议最大长度，

send失败，errno 设为EMSGSIZE；

#include <sys/socket.h>

ssize\_t sendto(int sockfd, const void \*buf, size\_t nbytes, int falgs

const struct sockaddr \*desaddr, socklet\_t deslen);

成功返回发送字节数， 失败返回0；

#include <sys/socket.h>

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg

int flags);

struct msghdr{

void \***msg\_name; /**\* optional address\*/

socklen\_t msg\_namelen;/\*address size in bytes\*/

struct iovec **msg\_iov /**\*array of I/O buffers\*/

int msg\_iovlen; /\*number of elements in array\*/

:

:

}

#include <sys/socket.h>

ssize\_t recv(int sockfd, void \*buf, size\_t nbytes, int flags);

返回值：返回数据的字节长度；若无数据或对等方结束，返回0；

若出错返回-1；

ssize\_t recnfrom(int sockfd, void \*buf, size\_t len, int flags,

struct sockaddr \*restrict addr

socklen\_t \*restrict addrlen);

返回值：返回数据的字节长度；若无数据或对等方结束，返回0；

若出错返回-1；

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

返回值：返回数据的字节长度；若无数据或对等方结束，返回0；

若出错返回-1；

struct msghdr{

void \***msg\_name; /**\* optional address\*/

socklen\_t msg\_namelen;/\*address size in bytes\*/

struct iovec **msg\_iov /**\*array of I/O buffers\*/

int msg\_iovlen; /\*number of elements in array\*/

:

:

}

**套接字选项**

获取和设置的3种选项

（1）通用选项，工作在所有套接字类型上

（2）在套接字层次管理选项，但是依赖于下层协议的支持

（3）特定于某些协议的选项，每个协议独有。

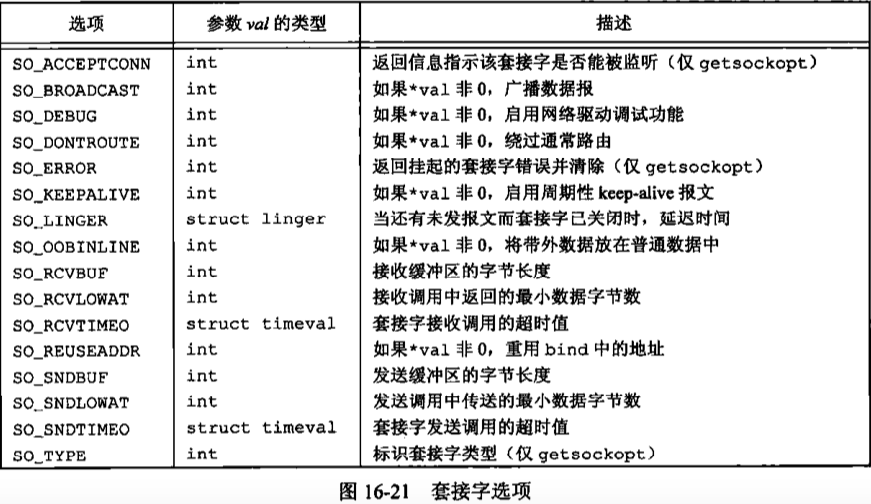
使用

回0， 出错返回-1

level：

通用套接字层次选项 设置SOL\_SOCKET

通用套接字层次选项



TCP选项 IPPRORTO\_TCP

IP 选项 IPPRORTO\_IP

Val : 根据选项的不同指向一个数据结构或者一个整数。

一些选项是on/off开关。如果整数非0，则启用选项。

如果整数为0，则禁止选项。

参数len指定了val指向对象的大小

getsockopt 查看当前选项的当前值

#include <sys/socket.h>

int getsockopt(int sockfd, int level, int option, void \*restrict val

socklen\_t \*restrict lenp;)

**IO多路复用**

**主要使用poll ，select， epoll函数；**

**I/O多路复用，I/O就是指的我们网络I/O,多路指多个TCP连接(或多个Channel)，复用指复用一个或少量线程。串起来理解就是很多个网络I/O复用一个或少量的线程来处理这些连接**

对一个**IO**端口，两次调用，两次返回，比阻塞**IO**并没有什么优越性；关键是能实现同时对多个**IO**端口进行监听

. select、poll、epoll简介

epoll跟select都能提供多路I/O复用的解决方案。在现在的**Linux**内核里有都能够支持，其中epoll是**Linux**所特有，而select则应该是POSIX所规定，一般操作系统均有实现

**select：**

select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。这样所带来的缺点是：

1、 单个进程可监视的fd数量被限制，即能监听端口的大小有限。

      一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048.

2、 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低：

       当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。

3、需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

**poll：**

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：

1、大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。                                                                                                                                      2、poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。

epoll:

epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就需态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知

epoll的优点：

**1、没有最大并发连接的限制，**能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）；  
**2、效率提升**，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；  
      即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的**网络**环境中，Epoll的效率就会远远高于select和poll。

**3、 内存拷贝**，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。  
  
**select、poll、epoll 区别总结：**

1、支持一个进程所能打开的最大连接数

|  |  |
| --- | --- |
| select | 单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32\*32，同理64位机器上FD\_SETSIZE为32\*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。 |
| poll | poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的 |
| epoll | 虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接 |

2、FD剧增后带来的**IO**效率问题

|  |  |
| --- | --- |
| select | 因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。 |
| poll | 同上 |
| epoll | 因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。 |

3、 消息传递方式

|  |  |
| --- | --- |
| select | 内核需要将消息传递到用户空间，都需要内核拷贝动作 |
| poll | 同上 |
| epoll | epoll通过内核和用户空间共享一块内存来实现的。 |

**总结：**

综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点。

1、表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

2、select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善

**select函数详解：**

#include <sys/select.h>

#include <sys/time.h>

#include select (int maxfdpl, fd\_set \*readset, fd\_set \*writeset,

fd\_set \*exceptset, const struct timeval \*timeout);

maxfdp1 = 最大文件描述符 + 1；（select内部的实现是一个循环遍历1024个文件描述符， [0, maxfdp1)];

readset = 读事件集合 writeset = 写事件集合 exceptset = 异常事件集合（传入传出参数） 每次调用select都需要重置这三个参数

#include <sys/select.h>

typedef long int \_\_fd\_mask;

#define \_\_NFDBITS (8 \* (int) sizeof (\_\_fd\_mask))

typedef struct

{

/\* XPG4.2 requires this member name. Otherwise avoid the name

from the global namespace. \*/

#ifdef \_\_USE\_XOPEN

\_\_fd\_mask fds\_bits[\_\_FD\_SETSIZE / \_\_NFDBITS];

# define \_\_FDS\_BITS(set) ((set)->fds\_bits)

#else

\_\_fd\_mask \_\_fds\_bits[\_\_FD\_SETSIZE / \_\_NFDBITS];

# define \_\_FDS\_BITS(set) ((set)->\_\_fds\_bits)

#endif

} fd\_set;

fd\_set结构体的定义实际包含的是fds\_bits位数组，该数组的每个元素的每一位标记一个文件描述符其大小固定，由FD\_SETSIZE指定（/usr/include/bits/typesizes.h中），在当前内核中数值为1024，可见每次select系统调用可监听处理的文件描述符最大数量为1024。

**使用宏：**

**FD\_ZERO(fd\_set \*fdset)**;将指定的文件描述符集清空。在对文件描述符集合进行设置前，必须对其进行初始化，如果不清空，由于在系统分配内存空间后，通常并不作清空处理，所以结果是不可知的。

FD\_SET(int fd,fd\_set \*fdset); 用于在文件描述符集合中增加一个新的文件描符。FD\_CLR(int fd,fd fd\_set \*fdset); 用于在文件描述符集合中删除一个文件描述符。FD\_ISSET(int fd,fd\_set fdset);用于测试指定的文件描述符是否在该集合中

最后一个参数是超时时间，是一个struct timeval结构体指针，该结构体定义如下：

　　有两个成员，一个是秒，一个是毫秒，

struct timeval{

　　long tv\_sec;　　　　//second

　　long tv\_usec;　　　//microseconds

}

超时时间可以设置到微秒级别，有三种设置情况：

NULL：阻塞等待，直到某个文件描述符上发生了事件。

0：仅检测描述符集合的状态，然后立即返回。

> 0： 指定超时时间，如果在该时间段里没有事件发生，select将超时返回。

####返回值

　　成功：返回就绪的文件描述符（可读、可写、异常）的总数。

　　超时：返回0

失败：返回-1，并设置errno，若在等待的过程中被信号打断，也返回-1，errno设置为EINTR。

网络程序中，select能处理的异常情况只有一种：socket上接收到带外数据。

不足：Select监听的最大文件描述符受限于FD\_SETSIZE，UNIX系统通常会在头文件 “sys/select.h” 中定义常量FD\_SETSIZE，一般为1024，要想更改需要重新编译内核。而且因为select采取的是轮询机制，当监听的文件描述符过多的话，效率会大大折扣。

一、select/poll中的读写事件

  1.下列四个条件中的任何一个满足时，套接口准备好多：

      a. 有数据可读，专业的说法是：套接字接收缓冲区中的数据字节数大于等于套接字接收缓冲区低潮限度的当前值。可以使用套接字选项SO\_RCVLOWAT来设置低潮限度，对于TCP和UDP套接字，其值缺省为1

     b. 连接的度这一半关闭，也就是说接收了FIN的TCP连接。对这样的套接字的套接字将不阻塞且返回0（即文件结束符）

    c.套接字是一个监听套接字且已完成的连接数为非0，即连接建立后可读

    d. 有一个套接字错误待处理。对这样的套接字的读操作将不阻塞且返回一个错误（-1），errno则设置成明确的错误条件。这些待处理的错误也可以通过指定套接口选项SO\_ERROR调用getsockopt来取得并清除。

  2.下列三个条件中的任一个满足时，套接字准备好写：

   a. 缓冲区可写，专业的说法是：套接字发送缓冲区中的可用字节数大于等于套接字发送缓冲区低潮限度的当前值，且或者套接字已连接或者套接字不要求连接（例如UDP套接字），对于TCP和UDP套接字，其缺省值一半为2048

  b. 连接的写这一半关闭。对这样的套接字的写操作将产生信号SIGPIPE

  c.有一个套接字错误待处理。

**Poll：** 没有最大文件描述符数量的限制。**poll**和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

#include <poll.h>

int poll (struct pollfd \*fds, unsigned int nfds, int timeout);

struct pollfd{

int fd; /\*文件描述符\*/

short events; /\*等待的事件\*/

short revent； /\*实际发生的事件\*/

}；

每一个pollfd结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示**poll**()监视多个文件描述符。每个结构体的events域是监视该文件描述符的事件掩码，由用户来设置这个域。revents域是文件描述符的操作结果事件掩码，内核在调用返回时设置这个域。events域中请求的任何事件都可能在revents域中返回。合法的事件如下：

　　POLLIN 　　　　　　　　有数据可读。

　　POLLRDNORM 　　　　  有普通数据可读。

　　POLLRDBAND　　　　　 有优先数据可读。

　　POLLPRI　　　　　　　　 有紧迫数据可读。

　　POLLOUT　　　　　　      写数据不会导致阻塞。

　　POLLWRNORM　　　　　  写普通数据不会导致阻塞。

　　POLLWRBAND　　　　　   写优先数据不会导致阻塞。

　　POLLMSGSIGPOLL 　　　　消息可用。

　　此外，revents域中还可能返回下列事件：  
　　POLLER　　   指定的文件描述符发生错误。

　　POLLHUP　　 指定的文件描述符挂起事件。

　POLLNVAL　　指定的文件描述符非法。

POLLIN | POLLPRI等价于select()的读事件，POLLOUT |POLLWRBAND等价于select()的写事件。POLLIN等价于POLLRDNORM |POLLRDBAND，而POLLOUT则等价于POLLWRNORM。例如，要同时监视一个文件描述符是否可读和可写，我们可以设置 events为POLLIN |POLLOUT。在**poll**返回时，我们可以检查revents中的标志，对应于文件描述符请求的events结构体。如果POLLIN事件被设置，则文件描述符可以被读取而不阻塞。如果POLLOUT被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可能被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。

　　timeout参数指定等待的毫秒数，无论I/O是否准备好，**poll**都会返回。timeout指定为负数值表示无限超时，使**poll**()一直挂起直到一个指定事件发生；timeout为0指示**poll**调用立即返回并列出准备好I/O的文件描述符，但并不等待其它的事件。这种情况下，**poll**()就像它的名字那样，一旦选举出来，立即返回。

　　返回值和错误代码  
　　成功时，**poll**()返回结构体中revents域不为0的文件描述符个数；如果在超时前没有任何事件发生，**poll**()返回0；失败时，**poll**()返回-1，并设置errno为下列值之一：  
　　EBADF　　       一个或多个结构体中指定的文件描述符无效。

　　EFAULTfds　　 指针指向的地址超出进程的地址空间。

　　EINTR　　　　  请求的事件之前产生一个信号，调用可以重新发起。

　　EINVALnfds　　参数超出PLIMIT\_NOFILE值。

　　ENOMEM　　     可用内存不足，无法完成请求。

**//（iocp (windows), kqueue(freebsd), 或是 epoll(linux) ）**

**EPOLL详解**

epoll是什么？按照man手册的说法：是为处理大批量句柄而作了改进的poll。当然，这不是2.6内核才有的，它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linuxkernel 2.5.44)，它几乎具备了之前所说的一切优点，被公认为Linux2.6下性能最好的多路I/O

二、epoll的相关系统调用

epoll只有**epoll\_create**,**epoll\_ctl**,**epoll\_wait** 3个系统调用。

<sys/epoll.h>

int epoll\_create(int size);

参数size：用来告诉内核要监听的数目一共有多少个。

返回值：成功时，返回一个非负整数的文件描述符，作为创建好的epoll句柄。调用失败时，返回-1，错误信息可以通过errno获得。  说明：创建一个epoll句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

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

参数epfd：**epoll\_create**()函数返回的epoll句柄。

参数op：操作选项。

参数fd：要进行操作的目标文件描述符。

参数event：struct epoll\_event结构指针，将fd和要进行的操作关联起来。

返回值：成功时，返回0，作为创建好的epoll句柄。调用失败时，返回-1，错误信息可以通过errno获得。

说明：epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。

参数op的可选值有以下3个：

EPOLL\_CTL\_ADD：注册新的fd到epfd中；  
EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；  
EPOLL\_CTL\_DEL：从epfd中删除一个fd；  
struct epoll\_event结构如下：

1. typedef union epoll\_data {
2. void \*ptr;
3. int fd;
4. \_\_uint32\_t u32;
5. \_\_uint64\_t u64;
6. } epoll\_data\_t;
8. struct epoll\_event {
9. \_\_uint32\_t events; /\* Epoll events \*/
10. epoll\_data\_t data; /\* User data variable \*/
11. };

events可以是以下几个宏的集合：  
EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；  
EPOLLOUT：表示对应的文件描述符可以写；  
EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；  
EPOLLERR：表示对应的文件描述符发生错误；  
EPOLLHUP：表示对应的文件描述符被挂断；  
EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。  
EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

**int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);//相当于select(int maxfdp1, struct fd\_set \*read\_set, struct fd\_set \*write\_set, struct fd\_set \*execpt\_set,** const struct timeval \*timeout**)**

  参数epfd：**epoll\_create**()函数返回的epoll句柄。

参数events：struct epoll\_event结构指针，用来从内核得到事件的集合。

参数 maxevents：告诉内核这0个events有多大

参数 timeout: 等待时的超时时间，以毫秒为单位。

返回值：成功时，返回需要处理的事件数目。调用失败时，返回0，表示等待超时

说明：等待事件的产生。

三、epoll工作原理

    epoll同样只告知那些就绪的文件描述符，而且当我们调用epoll\_wait()获得就绪文件描述符时，返回的不是实际的描述符，而是一个代表就绪描述符数量的值，你只需要去epoll指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射（mmap）技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。

**内存映射：简而言之就是将用户空间的一段内存区域映射到内核空间，映射成功后，用户对这段内存区域的修改可以直接反映到内核空间，同样，内核空间对这段区域的修改也直接反映用户空间。那么对于内核空间<---->用户空间两者之间需要大量数据传输等操作的话效率是非常高的**

    另一个本质的改进在于epoll采用基于事件的就绪通知方式。在select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll\_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll\_wait()时便得到通知。

Epoll的2种工作方式-水平触发（LT）和边缘触发（ET）

假如有这样一个例子：

1. 我们已经把一个用来从管道中读取数据的文件句柄(RFD)添加到epoll描述符

2. 这个时候从管道的另一端被写入了2KB的数据

3. 调用epoll\_wait(2)，并且它会返回RFD，说明它已经准备好读取操作

4. 然后我们读取了1KB的数据

5. 调用epoll\_wait(2)......

EdgeTriggered工作模式：

    如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll\_wait(2)之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。在上面的例子中，会有一个事件产生在RFD句柄上，因为在第2步执行了一个写操作，然后，事件将会在第3步被销毁。因为第4步的读取操作没有读空文件输入缓冲区内的数据，因此我们在第5步调用epoll\_wait(2)完成后，是否挂起是不确定的。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用ET模式的epoll接口，在后面会介绍避免可能的缺陷。

  i  基于非阻塞文件句柄

  ii 只有当read(2)或者write(2)返回EAGAIN时才需要挂起，等待。但这并不是说每次read()时都需要循环读，直到读到产生一个EAGAIN才认为此次事件处理完成，当read()返回的读到的数据长度小于请求的数据长度时，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

LevelTriggered工作模式

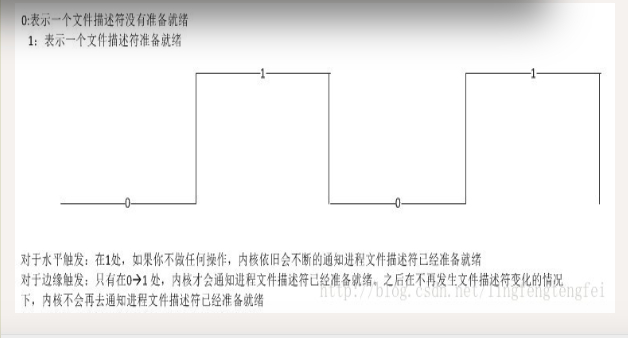
相反的，以LT方式调用epoll接口的时候，它就相当于一个速度比较快的poll(2)，并且无论后面的数据是否被使用，因此他们具有同样的职能。因为即使使用ET模式的epoll，在收到多个chunk的数据的时候仍然会产生多个事件。调用者可以设定EPOLLONESHOT标志，在 epoll\_wait(2)收到事件后epoll会与事件关联的文件句柄从epoll描述符中禁止掉。因此当EPOLLONESHOT设定后，使用带有EPOLL\_CTL\_MOD标志的epoll\_ctl(2)处理文件句柄就成为调用者必须作的事情。

LT(level triggered)是epoll缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表．

ET (edge-triggered)是高速工作方式，只支持no-block socket，它效率要比LT更高。ET与LT的区别在于，当一个新的事件到来时，ET模式下当然可以从epoll\_wait调用中获取到这个事件，可是如果这次没有把这个事件对应的套接字缓冲区处理完，在这个套接字中没有新的事件再次到来时，在ET模式下是无法再次从epoll\_wait调用中获取这个事件的。而LT模式正好相反，只要一个事件对应的套接字缓冲区还有数据，就总能从epoll\_wait中获取这个事件。

因此，LT模式下开发基于epoll的应用要简单些，不太容易出错。而在ET模式下事件发生时，如果没有彻底地将缓冲区数据处理完，则会导致缓冲区中的用户请求得不到响应。

图示说明：



*Nginx默认采用ET模式来使用epoll*

epoll的优点：

1.支持一个进程打开大数目的socket描述符(FD)

    select 最不能忍受的是一个进程所打开的FD是有一定限制的，由FD\_SETSIZE设置，默认值是2048。对于那些需要支持的上万连接数目的IM服务器来说显然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

2.IO效率不随FD数目增加而线性下降

    传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个"伪"AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，相反，如果过多使用epoll\_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。

3.使用mmap加速内核与用户空间的消息传递

这点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

4.内核微调

这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk\_buff结构，那么可以在运行时期动态调整这个内存pool(skb\_head\_pool)的大小--- 通过echoXXXX>/proc/sys/net/core/hot\_list\_length完成。再比如listen函数的第2个参数(TCP完成3次握手的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

linux下epoll如何实现高效处理百万句柄的

开发高性能网络程序时，windows开发者们言必称iocp，linux开发者们则言必称epoll。大家都明白epoll是一种IO多路复用技术，可以非常高效的处理数以百万计的socket句柄，比起以前的select和poll效率高大发了。我们用起epoll来都感觉挺爽，确实快，那么，它到底为什么可以高速处理这么多并发连接呢？

使用起来很清晰，首先要调用epoll\_create建立一个epoll对象。参数size是内核保证能够正确处理的最大句柄数，多于这个最大数时内核可不保证效果。

epoll\_ctl可以操作上面建立的epoll，例如，将刚建立的socket加入到epoll中让其监控，或者把 epoll正在监控的某个socket句柄移出epoll，不再监控它等等。

epoll\_wait在调用时，在给定的timeout时间内，当在监控的所有句柄中有事件发生时，就返回用户态的进程。

从上面的调用方式就可以看到epoll比select/poll的优越之处：因为后者每次调用时都要传递你所要监控的所有socket给select/poll系统调用，这意味着需要将用户态的socket列表copy到内核态，如果以万计的句柄会导致每次都要copy几十几百KB的内存到内核态，非常低效。而我们调用epoll\_wait时就相当于以往调用select/poll，但是这时却不用传递socket句柄给内核，因为内核已经在epoll\_ctl中拿到了要监控的句柄列表。

所以，实际上在你调用epoll\_create后，内核就已经在内核态开始准备帮你存储要监控的句柄了，每次调用epoll\_ctl只是在往内核的数据结构里塞入新的socket句柄。

当一个进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关

**/\***

**\*This structure is stored inside the "private\_data" member of the file**

**\*structure and represents the main data structure for the eventpoll**

**\*interface.**

**\*/**

**structeventpoll {**

**/\* Protect the access to thisstructure \*/**

**spinlock\_t lock;**

**/\***

**\* This mutex is used to ensurethat files are not removed**

**\* while epoll is using them. Thisis held during the event**

**\* collection loop, the filecleanup path, the epoll file   \* exit code and the ctl operations.**

**\*/**

**struct mutex mtx;**

**/\* Wait queue used bysys\_epoll\_wait() \*/**

**wait\_queue\_head\_t wq;**

**/\* Wait queue used byfile->poll() \*/**

**wait\_queue\_head\_t poll\_wait;**

**/\* List of ready file descriptors\*/**

**struct list\_head rdllist;**

**/\* RB tree root used to storemonitored fd structs \*/**

**/\*红黑树根节点，这棵树存储着所有添加到epoll中的事件，也就是这个epoll监控的事件 \*/**

**struct rb\_root rbr;**

**/\***

**\* This is a single linked listthat chains all the "struct epitem" that**

**\* happened while transferringready events to userspace w/out**

**\* holding ->lock.**

**\*/**

**struct epitem\*ovflist;**

**/\* wakeup\_source used whenep\_scan\_ready\_list is running \*/**

**struct wakeup\_source\*ws;**

**/\* The user that created theeventpoll descriptor \*/**

**struct user\_struct\*user;**

**struct file\*file;**

**/\* used to optimize loopdetection check \*/**

**int visited;**

**/\*双向链表中保存着将要通过epoll\_wait返回给用户的、满足条件的事件 \*/**

**struct list\_head visited\_list\_link;**

**};**

**每一个epoll对象都有一个独立的eventpoll结构体，这个结构体会在内核空间中创造独立的内存，用于存储使用epoll\_ctl方法向epoll对象中添加进来的事件。这样，重复的事件就可以通过红黑树而高效的识别出来。**

**在epoll中，对于每一个事件都会建立一个epitem结构体：**

**/\***

**\* Each file descriptor added to the eventpoll interfacewill**

**\* have an entry of this type linked to the"rbr" RB tree.**

**\* Avoid increasing the size of this struct, there can bemany thousands**

**\* of these on a server and we do not want this to takeanother cache line.**

**\*/**

**struct epitem {**

**/\* RB tree node used to link this structure to theeventpoll RB tree \*/**

**struct rb\_node rbn;**

**/\* List header used to link this structure to the eventpollready list \*/**

**struct list\_head rdllink;**

**/\***

**\* Workstogether "struct eventpoll"->ovflist in keeping the**

**\* singlelinked chain of items.**

**\*/**

**struct epitem\*next;**

**/\* The file descriptor information this item refers to \*/**

**struct epoll\_filefd ffd;**

**/\* Number of active wait queue attached to poll operations\*/**

**int nwait;**

**/\* List containing poll wait queues \*/**

**struct list\_head pwqlist;**

**/\* The "container" of this item \*/**

**struct eventpoll\*ep;**

**/\* List header used to link this item to the "structfile" items list \*/**

**struct list\_head fllink;**

**/\* wakeup\_source used when EPOLLWAKEUP is set \*/**

**struct wakeup\_source \_\_rcu\*ws;**

**/\* The structure that describe the interested events andthe source fd \*/**

**struct epoll\_event event;**

**};**

**此外，epoll还维护了一个双链表，用户存储发生的事件。当epoll\_wait调用时，仅仅观察这个list链表里有没有数据即eptime项即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll\_wait非常高效。**

**而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll\_wait仅需要从内核态copy少量的句柄到用户态而已，如何能不高效？！**

**那么，这个准备就绪list链表是怎么维护的呢？当我们执行epoll\_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。**

**如此，一颗红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。执行epoll\_create时，创建了红黑树和就绪链表，执行epoll\_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。执行epoll\_wait时立刻返回准备就绪链表里的数据即可。**

**四、epoll的使用方法**

**那么究竟如何来使用epoll呢？其实非常简单。**

**通过在包含一个头文件#include <sys/epoll.h> 以及几个简单的API将可以大大的提高你的网络服务器的支持人数。**

**首先通过create\_epoll(intmaxfds)来创建一个epoll的句柄。这个函数会返回一个新的epoll句柄，之后的所有操作将通过这个句柄来进行操作。在用完之后，记得用close()来关闭这个创建出来的epoll句柄。**

**之后在你的网络主循环里面，每一帧的调用epoll\_wait(int epfd, epoll\_event events, int max events, inttimeout)来查询所有的网络接口，看哪一个可以读，哪一个可以写了。基本的语法为：**

**nfds = epoll\_wait(kdpfd, events, maxevents, -1);**

**其中kdpfd为用epoll\_create创建之后的句柄，events是一个epoll\_event\*的指针，当epoll\_wait这个函数操作成功之后，epoll\_events里面将储存所有的读写事件。max\_events是当前需要监听的所有socket句柄数。最后一个timeout是 epoll\_wait的超时，为0的时候表示马上返回，为-1的时候表示一直等下去，直到有事件返回，为任意正整数的时候表示等这么长的时间，如果一直没有事件，则返回。一般如果网络主循环是单独的线程的话，可以用-1来等，这样可以保证一些效率，如果是和主逻辑在同一个线程的话，则可以用0来保证主循环的效率。**

**epoll\_wait返回之后应该是一个循环，遍历所有的事件。**

**几乎所有的epoll程序都使用下面的框架：**

**for( ; ; )**

**{**

**nfds = epoll\_wait(epfd,events,20,500);**

**for(i=0;i<nfds;++i)**

**{**

**if(events[i].data.fd==listenfd)//有新的连接**

**{**

**connfd = accept(listenfd,(sockaddr\*)&clientaddr,&clilen);//accept这个连接**

**ev.data.fd=connfd;**

**ev.events=EPOLLIN|EPOLLET;**

**epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev);//将新的fd添加到epoll的监听队列中**

**}**

**else if( events[i].events&EPOLLIN )//接收到数据，读socket**

**{**

**n = read(sockfd, line, MAXLINE))<0    //读**

**ev.data.ptr = md;     //md为自定义类型，添加数据**

**ev.events=EPOLLOUT|EPOLLET;**

**epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，异步处理的精髓**

**}**

**else if(events[i].events&EPOLLOUT)//有数据待发送，写socket**

**{**

**struct myepoll\_data\* md= (myepoll\_data\*)events[i].data.ptr;    //取数据**

**sockfd = md->fd;**

**send( sockfd, md->ptr, strlen((char\*)md->ptr),0 );        //发送数据**

**ev.data.fd=sockfd;**

**ev.events=EPOLLIN|EPOLLET;**

**epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时接收数据**

**}**

**else**

**{**

**//其他的处理**

**}**

**}**

**}**