(1条消息)linux串口编程 select - storyteller的博客 - CSDN博客

1、串口的阻塞和非阻塞

阻塞的定义：

对于read，block指当串口输入缓冲区没有数据的时候，read函数将会阻塞在这里，一直到串口输入缓冲区中有数据可读取，read读到了需要的字节数之后，返回值为读到的字节数，然后整个程序才继续运行下去；（收）

对于write，block指当串口输出缓冲区满，或剩下的空间小于将要写入的字节数，则write将阻塞，一直到串口输出缓冲区中剩下的空间大于等于将要写入的字节数，执行写入操作，返回写入的字节数，然后整个程序才继续运行下去。（发）

非阻塞的定义：

对于read，no block指当串口输入缓冲区没有数据的时候，read函数立即返回，返回值为0。

对于write，no block指当串口输出缓冲区满，或剩下的空间小于将要写入的字节数，则write将进行写操作（不会等待在这里），写入当前串口输出缓冲区剩下空间允许的字节数，然后返回写入的字节数。

控制方法：

有两个方法可以控制串口阻塞性（同时控制read和write）：一个是在打开串口的时候，open函数是否带O_NDELAY；第二个是可以在打开串口之后通过fcntl()函数进行控制。

open方式：

阻塞：fd = open(devname, O_RDWR | O_NOCTTY);

非阻塞：fd = open(devname, O_RDWR | O_NOCTTY | O_NDELAY);

fcntl函数：

阻塞：fcntl(fd,F_SETFL,0)

非阻塞：fcntl(fd,F_SETFL,FNDELAY)

```
1.  if(fcntl(fd,F_SETFL,FNDELAY) < 0)
2.   {
3.      printf("fcntl failed\n");
4.   }
5.   else{
6.   printf("fcntl=%d\n",fcntl(fd,F_SETFL,FNDELAY));
7.   }
```

```
1.   if(fcntl(fd,F_SETFL,0) < 0){
2.      printf("fcntl failed\n");
3.      }
4.      else{
5.      printf("fcntl=%d\n",fcntl(fd,F_SETFL,0));
6.      }
```

2、串口配置

需要包含<termios.h>这个文件，该文件中定义了struct termios这个结构体类型。
struct termios结构至少包含以下成员：

```
        tcflag_t c_iflag;      /* input modes */
        tcflag_t c_oflag;      /* output modes */
        tcflag_t c_cflag;      /* control modes */
        tcflag_t c_lflag;      /* local modes */
        cc_t     c_cc[NCCS];   /* control chars */
```

1 c_cflag
c_cflag成员用于控制串口波特率、数据位、校验位、停止位以及硬件流控制等等，包成员有：

```
CBAUD                   波特率掩码位
        B0
        B50
        B75
        B110
        B134
        B150
        B200
        B300
        B600
        B1200
        B2400
        B4800
        B9600
        B19200
        B38400
        B57600
        B76800
        B115200
EXTA                    外部时钟
EXTB                    外部时钟
CSIZE                   数据位掩码位
        CS5
        CS6
        CS7
        CS8
CSTOPB                  2位停止位
CREAD                   接收使能
PARENB                  奇偶校验使能
PARODD                  使用奇校验
CLOCAL                  忽略调制解调点行
CRTSCTS                 硬件流控制使能位
```

通常情况下，CLOCAL和CREAD这两个选项应该应该总是被打开的。

1.1 设置波特率
波特率的存储位置依赖于操作系统，在比较老接口上波特率存储在c_cflag成员中，在后来的接口中提供了c_ispeed和c_ospeed这两个成员来存储实际的波特率值，所以在设置波特率时应该使用cfsetospeed和cfsetispeed这两个函数（而不是直接赋值的方式）。例如：

```
struct termios options;

/*
 * Get the current options for the port...
 */
tcgetattr(fd, &options);

/*
 * Set the baud rates to 19200...
 */
cfsetispeed(&options, B19200);
cfsetospeed(&options, B19200);

/*
 * Enable the receiver and set local mode...
 */
options.c_cflag |= (CLOCAL | CREAD);

/*
 * Set the new options for the port...
 */
tcsetattr(fd, TCSANOW, &options);
```

其中同时了tcgetattr和tcsetattr这两个函数用于获取和设置串口的属性。
tcgetattr函数原型如下：

```
int tcgetattr(int fd, struct termios *termios_p);
```

tcgetattr用于获取当前的串口设置到它的参数termios_p中，而要修改串口设置则使用tcsetattr函数，原型如下：

```
int tcsetattr(int fd, int optional_actions,
              const struct termios *termios_p);
```

其中optional_actions有几个选项值：

```
TCSANOW         立即修改设置
TCSADRAIN       等当所有数据传输完成后才修改设置
TCSAFLUSH       同样需要等待，但是它是立即刷新输入、输出缓冲区，然后才修改设置。
```

而cfsetispeed和cfsetospeed函数是专门用于设置串口波特率的，函数原型如下：

```
int cfsetispeed(struct termios *termios_p, speed_t speed);
int cfsetospeed(struct termios *termios_p, speed_t speed);
```

1.2 设置数据位

```
options.c_cflag &= ~CSIZE;      /* Mask the character size bits */
options.c_cflag |= CS8;         /* Select 8 data bits */
```

1.3 设置奇偶校验（通用数据位、停止位一起设置）
无校验（8N1）：

```
options.c_cflag &= ~PARENB;
options.c_cflag &= ~CSTOPB;
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;
```

1.4 设置硬件流控制
禁用硬件流控制：

```
options.c_cflag &= ~CRTSCTS;
```

```
2 c_lflag
ISIG            使能SIGINTR、SIGSUSP、SIGDSUSP和SIGQUIT信号
ICANON          使能规范输入模式
ECHO            使能输入字符回显功能
```

2.1 选择标准输入模式

```
options.c_lflag |= (ICANON | ECHO | ECHOE);
```

2.2 选择原始输入模式

```
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
```

那么什么是标准输入模式（Canonical Input），什么又是原始输入模式（Raw Input）呢？
所谓标准输入模式是指输入是以行为单位的，可以这样理解，输入的数据是存放在缓冲一个缓冲区里面（但并未真正发送出去），可以使用Backspace或者Delete键来删除输入的字符，从而达到修改字符的目的，当按下回车键时，输入才真正的发送出去，这样终端程序才能接收到。
而原始输入模式是指输入是以字节为单位的，不以这样理解，输入的数据是存放在缓冲一个缓冲区里面（但并未真正发送出去），可以使用Backspace或者Delete键来删除输入的字符，从而达到修改字符的目的，当按下回车键时，输入才真正的发送出去，这样终端程序才能接收到。
通常情况下我们都是使用的是原始输入模式，也就是说输入的数据并不经缓行。在标准输入模式下，系统每次返回的是一行数据，在原始输入模式下，系统又是怎样返回数据的呢？如果读一次就返回一个字节，那么系统开销就会很大，但在读数据的时候，我们也并不知道一次要读多少字节的数据，解决办法是使用c_cc数组中的VMIN和VTIME，如果已经读到了VMIN个字节的数据或者已经超过VTIME时间，系统立即返回。关于VMIN和VTIME这两个选项后面还会详细说明。

```
3 c_iflag
INPCK           使能输入校验
IGNPAR          忽略校验错误
PARMRK          标记校验错误
IXON            使能输出软件流控制
IXOFF           使能输入软件流控制
```

3.1 使能软件流控制
例如：

```
options.c_iflag |= (IXON | IXOFF | IXANY);
```

3.2 禁用软件流控制
例如：

```
options.c_iflag &= ~(IXON | IXOFF | IXANY);
```

```
4 c_oflag
OPOST           启用输出处理
```

可以启用和禁止输出处理，例如：

```
options.c_oflag |= OPOST;       /* Choosing Processed Output */

options.c_oflag &= ~OPOST;      /* Choosing Raw Output */
```

```
5 c_cc
那么可能需要关注的是VMIN和VTIME这两个选项。
VMIN            最少读取字符数
VTIME           超时时间
```

这两个参数只有当设置为阻塞模式时才有效，有以下几种可能值：
5.1 MIN > 0 && TIME > 0
MIN为最少读取的字符数，当读取到一个字符后，会启动一个定时器，在定时器超时事前，如果已经读取到了MIN个字符，则read返回MIN个字符。如果在接收到MIN个字符之前，定时器已经超时，则read返回已读取到的字符，注意这个定时器会在每次读取到一个字符后重新启动，即重新开始计时，而且是读取到第一个字节后才启动，也就是说超时的情况下，至少读取到一个字节数据。

5.2 MIN > 0 && TIME == 0
在只有读取到MIN个字符时，read才返回，可能造成read被永久阻塞。

5.3 MIN == 0 && TIME > 0
和第一种情况稍有不同，在接收到一个字节时或者定时器超时时，read返回。如果是超时这种情况，read返回值是0。

5.4 MIN == 0 && TIME == 0
这种情况下read总是立即就返回，即不会被阻塞。

3、select编程

selcet函数：

```
int select(int maxfdp,fd_set *readfds,fd_set *writefds,fd_set *errorfds,struct timeval *timeout);
```

在说明参数之前，先说明2个结构体：

```
struct fd_set
```

可以理解为一个集合，这个集合中存放的是文件描述符(file descriptor)，即文件句柄，这可以是我们所说的普通意义的文件，当然Unix下任何设备、管道、FIFO等都是文件形式，全部包括在内，所以毫无疑问一个socket就是一个文件，socket句柄就是一个文件描述符。fd_set集合可以通过一些宏由人为来操作，比如：

? FD_ZERO(fd_set *set):清除一个文件描述符集；

? FD_SET(int fd, fd_set *set):将一个文件描述符加入文件描述符集中；

? FD_CLR(int fd, fd_set *set):将一个文件描述符从文件描述符集中清除；

? FD_ISSET(int fd, fd_set *set): 检查集合中指定的文件描述符是否可以读写。

struct timeval

struct timeval {

long tv_sec ;

lone tv_usec ;

}

设置超时时间，作为select的最后一个参数。

下面说明select的参数：

int maxfdp

是一个整数值，是指集合中所有文件描述符的范围，即所有文件描述符的最大值加1，不能错！在Windows中这个参数的值无所谓，可以设置不正确。

fd_set *readfds

是指向fd_set结构的指针，这个集合中应该包括文件描述符，我们是要监视这些文件描述符的读变化的，即我们关心是否可以从这些文件中读取数据了，如果这个集合中有一个文件可读，select就会返回一个大于0的值，表示有文件可读，如果没有可读的文件，则根据timeout参数再判断是否超时，若超出timeout的时间，select返回0，若发生错误返回负值。可以传入NULL值，表示不关心任何文件的读变化。

fd_set *writefds

是指向fd_set结构的指针，这个集合中应该包括文件描述符，我们是要监视这些文件描述符的写变化的，即我们关心是否可以向这些文件中写入数据了，如果这个集合中有一个文件可写，select就会返回一个大于0的值，表示有文件可写，如果没有可写的文件，则根据timeout参数再判断是否超时，若超出timeout的时间，select返回0，若发生错误返回负值。可以传入NULL值，表示不关心任何文件的写变化。

fd_set *errorfds

同上面两个参数的意图，用来监视文件错误异常。

struct timeval* timeout

是select的超时时间，这个参数至关重要，它可以使select处于三种状态，第一，若将NULL以形参传入，即不传入时间结构，就是将select置于阻塞状态，一定等到监视文件描述符集合中某个文件描述符发生变化为止；第二，若将时间值设为0秒0毫秒，就变成一个纯粹的非阻塞函数，不管文件描述符是否有变化，都立刻返回继续执行，文件无变化返回0，有变化返回一个正值；第三，timeout的值大于0，这就是等待的超时时间，即select在timeout时间内阻塞，超时时间之内有事件到来就返回了，否则在超时后不管怎样一定返回，返回值同上述。

返回值ret

负值：select错误；

正值：某些文件可读写或出错；

0：等待超时，没有可读写或错误的文件；

在select编程时，一般来说，首先使用FD_ZERO、FD_SET来初始化文件描述符集，在使用了select函数时，可循环使用FD_ISSET测试描述符集，在执行完对相关的文件描述符后，使用FD_CLR来清除描述符集。

使用FD_ISSET检测串口是否有读写动作时，每次循环都要清空，否则不会检测到有变化：

FD_ZERO(&rfds);// 清空串口接收端口集

FD_SET(fd,&rfds);// 设置串口接收端口集

4、read阻塞配置

除了在open函数或者fcntl函数中配置阻塞方式外，read操作还有额外的配置：

options.c_cc[VMIN] = xxx;

```
options.c_cc[VTIME] = xxx;
这两个配置只有当设置为阻塞方式（blocking IO）时才有效，否则是无效的，这两个参数的默认值为0。
其中VMIN表示read操作时最小读取的字节数。
VTIME表示read操作时没有读到数据时等待的时间，单位为10毫秒。例如：
options.c_cc[VMIN] = 8; /* 表示最少读取8个字节 */
options.c_cc[VTIM] = 5; /* 表示超时时间为50毫秒 */
```

5、ioctl

那么对于读来说，还可以使用ioctl函数在read之前获取可读的字节数，这样也就不用关心read是阻塞与非阻塞了，例如：

```
#include <unistd.h>
#include <termios.h>
int fd;
int bytes;
ioctl(fd, FIONREAD, &bytes);
```

附录：串口打开和初始化部分代码

```
#define DEVNAME "/dev/ttyUSB0"

int serial_init(void)
{
        struct termios options;

        /* 以非阻塞方式打开串口 */
        fd = open(DEVNAME, O_RDWR | O_NOCTTY | O_NDELAY);
        if (fd < 0) {
                printf("Open the serial port error!\n");
                return -1;
        }
        fcntl(fd, F_SETFL, 0);
        tcgetattr(fd, &options);
        /*
         * Set the baud rates to 9600
         */
        cfsetispeed(&options, B9600);
        cfsetospeed(&options, B9600);

        /*
         * Enable the receiver and set local mode
         */
        options.c_cflag |= (CLOCAL | CREAD);

        /*
         * Select 8 data bits, 1 stop bit and no parity bit
         */
        options.c_cflag &= ~PARENB;
        options.c_cflag &= ~CSTOPB;
        options.c_cflag &= ~CSIZE;
        options.c_cflag |= CS8;

        /*
         * Disable hardware flow control
         */
        options.c_cflag &= ~CRTSCTS;

        /*
         * Choosing raw input
         */
        options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);

        /*
         * Disable software flow control
         */
        options.c_iflag &= ~(IXON | IXOFF | IXANY);

        /*
         * Choosing raw output
         */
        options.c_oflag &= ~OPOST;

        /*
         * Set read timeouts
         */
        options.c_cc[VMIN] = 8;
        options.c_cc[VTIME] = 10;
        //options.c_cc[VMIN] = 0;
        //options.c_cc[VTIME] = 0;

        tcsetattr(fd, TCSANOW, &options);
        return 0;
}
```

select方式读取数据代码：

```
int main(void)
{
  int fd;
  int nread,nwrite,i;
  char buff[8];
  fd_set rd;
  fd = 0;
  /*打开串口*/
  if((fd = open_port(fd,1)) < 0)
  {
    perror("open_port error!\n");
    return ;
  }
  /*设置串口*/
  if((i= set_opt(fd,115200,8,'N',1)) < 0)
  {
    perror("set_opt error!\n");
    return (-1);
  }
  /*利用select函数来实现多个串口的读写*/
while(1)
{
  FD_ZERO(&rd);
  FD_SET(fd,&rd);
  while(FD_ISSET(fd,&rd))
  {
    if(select(fd+1,&rd,NULL,NULL,NULL) < 0)
      perror("select error!\n");
    else
    {
      while((nread = read(fd,buff,8))>0)
      {
        printf("nread = %d,%s\n",nread,buff);
      }
    }
  }
}
close(fd);
    return ;
}
```