可能是全网最全的select讲解

原创 陈同学在搬砖 陈同学在搬砖 2020-04-12 02:53

用途

相比与普通的阻塞IO模型
select相当于是一名监管员
把多个要处理的文件描述符纳入自己的监管
在设定的时间内阻塞查询 看哪些套接字是就绪的
如果是就绪的则对这些套接字进行IO处理

用法

看一下下面这段简单的代码
实现的功能就是把标准输入(即文件描述符为0)
那入select的监管
然后select在5s内阻塞的轮询
看是否有读就绪事件
如果有的话就返回 然后对其进行处理
如果超时或者出错的或也返回

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include<strings.h>
#include<sys/socket.h>
#include<iostream>
#include<arpa/inet.h>
using namespace std;

int main(void) {

    /**step1 :  select工作之前,需要知道要监管哪些套接字**/
   int listen_fd=0;

    fd_set  read_set;
    FD_ZERO(&read_set);
    FD_SET(listen_fd,&read_set);


    /*step2 : select开始工作,设定时间内阻塞轮询套接字是否就绪*/
    struct timeval tv;
       tv.tv_sec = 5;
        tv.tv_usec = 0;
    int ret=select(listen_fd+1,&read_set,NULL,NULL,&tv);

    /*step3 : select完成工作,即如果出现就绪或者超时 ,则返回*/
    if(ret==-1){
        cout<<"errno!"<<endl;
    }
    else if(ret==0){
        cout<<"time out"<<endl;
    }
    else if(ret>0){
             if(FD_ISSET(listen_fd,&read_set));
             {
                 char *buffer=new char[10];
                 read(listen_fd,buffer,sizeof(buffer));
                 cout<<"Input String : "<<buffer<<endl;
             }

    }

}
```

使用方法总结如下

接口

上面的使用涉及到了下面几个接口

fd_set

1.fd_set是一种位数组类型，
也就是说数组中的数组元素值只能是0或1

2.因为由上面小实例
可以看出 select要监听三种就绪事件(可读 可写 出错)
是通过先建立三个事件对应的位数组
然后三个位数组初始化
 然后把要监听的套接字
 在该套接字数组中置为1进行的

FD_SET

FD_SET(int fd, fd_set *set)
把文件描述符fd加入到
对应的监听列表(fd_set类型的位数组),
就是把数组中该文件描述符位的元素置1,

FD_CLR

FD_CLRint fd, fd_set *set)
把文件描述符fd踢出
出对应的监听列表(fd_set类型的位数组),
就是把数组中该文件描述符位的元素置0,

FD_ISSET

int FD_ISSETint fd, fd_set *set)
判断文件描述符fd是否
在set对应的事件中就绪(fd_set类型的位数组),
就是判断该位数组是否为1
是的话 返回1 否则返回0

FD_ZERO

FD_ISSET(fd_set *set)
对监听列表ser进行置0 相当于对其进行初始化

select

接口：
int select(int nfds, fd_set *readfds, fd_set *writefds,fd_set *exceptfds, struct timeval *timeout);

参数：
nfds 表示总共有几个要监管查询的套接字
通常被设定为所监听的最大文件描述符值+1
因为文件描述附是从0开始的
fd_set是一种位数组类型，
也就是说数组中的数组元素值只能是0或1
readfd表示要进行监管的读操作的套接字的数组，
writefds表示要进行监管的写操作套接字的数组
exceptfds表示要进行监管的异常事件套接字的数组
参数timeout表示每次查询停留的时间，
其中timeval结构体的格式如下
struct timeval {
long tv_sec; /* 秒数 */
long tv_usec; /* 毫秒数 */
}

阻塞情况：
3种情况：
设置为NULL,永远等下去(阻塞)，
设置timeval,等待固定时间(固定时间阻塞)，
设置timeval为0,检查描述字后立即返回,(非阻塞)

返回值：

3种情况：
在此期间只要有一个套接字就就绪了，
select就会返回停止阻塞

成功时返回就绪套接字数目
去判断自己感兴趣的套接字是否在其中
然后调用read /accept同步读写或者建立连接等IO操作进行响应
 如果就绪就执行相应的处理

 超时时返回0

失败时返回-1
同时设置errno
失败时候的errno可能有以下情况
EBADF 监听集合中传入了无效的文件描述符
EINTR select在工作过程中被信号中断了
EINVAL nfds 参数是负数或者超出了最大的数量限制
EINVAL The value contained within timeout is invalid.
 ENOMEM 无法开辟内存

源码

源码框架

这里简单的总结了select源码的一个框架
点击查看大图

源码细节

将注释都写好了 细节可以看注释

select对应的系统调用如下

```
SYSCALL_DEFINE5(select, int, n, fd_set __user *, inp, fd_set __user *, outp,
  fd_set __user *, exp, struct timeval __user *, tvp)
```

将其展开后得到如下函数

```
long sys_select(int n, fd_set __user * inp, fd_set __user * outp,
                    fd_set __user * exp, struct timeval __user * tvp)
```

```
SYSCALL_DEFINE5(select, int, n, fd_set __user *, inp, fd_set __user *, outp,
  fd_set __user *, exp, struct timeval __user *, tvp)
{
    /* 从应用层会传递过来三个需要监听的集合，可读，可写，异常 */
    ret = core_sys_select(n, inp, outp, exp, to);

    return ret;
}
```

接下来看core_sys_select

```
int core_sys_select(int n, fd_set __user *inp, fd_set __user *outp,
      fd_set __user *exp, struct timespec *end_time)
{
    /* 在栈上分配一段内存 */
    long stack_fds[SELECT_STACK_ALLOC/sizeof(long)];

    size = FDS_BYTES(n); //n个文件描述符需要多少个字节

    /*
     * 如果栈上的内存太小，那么就重新分配内存
     * 为什么是除以6呢?
     * 因为每个文件描述符要占6个bit（输入：可读，可写，异常；输出结果：可读，可写，异常）
     */
    if (size > sizeof(stack_fds) / 6)
  bits = kmalloc(6 * size, GFP_KERNEL);

    /* 设置好bitmap对应的内存空间 */
    fds.in      = bits; //可读
 fds.out     = bits +   size; //可写
 fds.ex      = bits + 2*size; //异常
 fds.res_in  = bits + 3*size; //返回结果，可读
 fds.res_out = bits + 4*size; //返回结果，可写
 fds.res_ex  = bits + 5*size; //返回结果，异常

    /* 将应用层的监听集合拷贝到内核空间 */
    get_fd_set(n, inp, fds.in);
    get_fd_set(n, outp, fds.out);
    get_fd_set(n, exp, fds.ex);

    /* 清空三个输出结果的集合 */
 zero_fd_set(n, fds.res_in);
 zero_fd_set(n, fds.res_out);
 zero_fd_set(n, fds.res_ex);

    /* 调用do_select阻塞，满足条件时返回 */
    ret = do_select(n, &fds, end_time);

    /* 将结果拷贝回应用层 */
    set_fd_set(n, inp, fds.res_in);
    set_fd_set(n, outp, fds.res_out);
    set_fd_set(n, exp, fds.res_ex);

    return ret;
}
```

下面来看一看do_select函数

```
int do_select(int n, fd_set_bits *fds, struct timespec *end_time)
{
    for (;;) {
        /* 遍历所有监听的文件描述符 */
     for (i = 0; i < n; ++rinp, ++routp, ++rexp)
        {
            for (j = 0; j < __NFDBITS; ++j, ++i, bit <<= 1)
            {
                /* 调用每一个文件描述符对应驱动的poll函数，得到一个掩码 */
                mask = (*f_op->poll)(file, wait);

                /* 根据掩码设置相应的bit */
                if ((mask & POLLIN_SET) && (in & bit)) {
                    res_in |= bit;
                    retval++;
                }

                if ((mask & POLLOUT_SET) && (out & bit)) {
                    res_out |= bit;
                    retval++;
                }

                if ((mask & POLLEX_SET) && (ex & bit)) {
                    res_ex |= bit;
                    retval++;
                }
            }
        }

        /* 如果条件满足，则退出 */
        if (retval || timed_out || signal_pending(current))
            break;

        /* 调度，进程睡眠 */
        poll_schedule_timeout(&table, TASK_INTERRUPTIBLE, to, slack);
    }
}
```

do_select会遍历所有要监听的文件描述符，调用对应驱动程序的poll函数，驱动程序的poll一般实现如下

```
static unsigned int button_poll(struct file *fp, poll_table * wait)
{
 unsigned int mask = 0;

    /* 调用poll_wait */
 poll_wait(fp, &wq, wait); //wq为自己定义的一个等待队列头

 /* 如果条件满足，返回相应的掩码 */
 if(condition)
  mask |= POLLIN;

 return mask;
}
```

看看poll_wait做了什么

```
static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
 if (p && wait_address)
  p->qproc(filp, wait_address, p);
}
```

p->qproc在之前又被初始化为__pollwait

```
static void __pollwait(struct file *filp, wait_queue_head_t *wait_address,
    poll_table *p)
{
    /* 分配一个结构体 */
    struct poll_table_entry *entry = poll_get_entry(pwq);

    /* 将等待队列元素加入驱动程序的等待队列头中 */
 add_wait_queue(wait_address, &entry->wait);
}
```

至此 select源码分析完毕

特性

select特点分析

```
select缺点:
(1)select能监听的文件描述符个数受限于FD_SETSIZE,
一般为1024

(2)源码中的do_select部分是采用for循环的形式来遍历的,
也就是select采用轮询的方式扫描文件描述符,
文件描述符数量越多，性能越差;

(3)源码中的code_sys_select在每次在轮询期间
都需要将用户态的监听位数组拷贝到内核态的fds对象中,
select需要复制大量的句柄数据结构到内核空间,
产生巨大的开销;

(4)select返回的是含有整个句柄的数组,
应用程序需要遍历整个数组才能发现哪些句柄发生了事件 比较繁琐;

(5)select的触发方式是水平触发,
应用程序如果没有完成
对一个已经就绪的文件描述符进行IO操作,
那么之后每次select调用还是会
将这些文件描述符通知进程。
```

```
select优点:
  (1)用户可以在一个线程内
  同时处理多个socket的IO请求。
  同时没有多线程多进程那样耗费系统资源


   (2)目前几乎在所有的平台上支持,
   其良好跨平台支持也是它的一个优点
```

我是陈同学

让技术 有温度

你的支持是我搬砖的动力

往期精彩回顾

听说点击在看的今年都会暴富脱单，升职加薪