10 非阻塞 IO

这章来讲另一种 IO 模型非阻塞 IO(NIO)，也就是同步非阻塞 IO。上一章说过，IO 操作的两个阶段先查询再读写，而非阻塞 IO 在查询阶段的处理和阻塞 IO 不同。应用程序需要进行 IO 操作前，先发起查询，驱动程序根据数据情况返回查询结果，如果返回查询结果 NG，应用程序就不执行读写操作了。如果应用程序非要读写的话，就继续去查询，直到驱动程序返回数据准备完成，才会做下一步的读写操作。

10.1 Linux 中的 NIO

非阻塞 IO 的处理方式是轮询。Linux 中提供了应用程序的轮询机制和相应的驱动程序系统调用。

10.1.1 应用程序中的轮询方法

应用程序中提供了三种轮询的方法：select、poll、epoll。实际上他们也是多路复用 IO 的解决方法，这里就不展开说了。

1) select

select 有良好的跨平台支持性，但是他的单个进程能够监视的文件描述符的数量存在最大限制(Linux 中一般为 1024)。

函数原型：

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

参数说明：

maxfdp：是集合中所有文件描述符的范围，即等于所有文件描述符的最大值加 1。

readfds：struct fd_set 结构体可以理解为是文件描述符(file descriptor)的集合，也就是文件句柄，他的每一位就代表一个描述符，readfds 用于监视指定描述符集的读变化，只要其中有一位可读，select 就会返回一个大于 0 的值。可以已使用这些宏来操作 fd_set 变量：

```
/* 把 fdset 所有位置 0, 清楚 fdset 和所有文件句柄的关系 */
FD_ZERO(fd_set *fdset)
/* 把 fdset 某个位置 1, 把 fdset 和文件句柄 fd 关联 */
FD_SET(int fd, fd_set *fdset)
/* 把 fdset 某个位置 0, 取消 fdset 和文件句柄的关联 */
FD_CLR(int fd, fd_set *fdset)
/* 判断某个文件句柄是否为 1, 即是否可操作 */
FD_ISSET(int fd, fdset *fdset)
```

writefds：用于见识文件是否可写。

errorfds：用于监视文件异常。

timeout：struct timeval 用来代表时间值，有两个成员，一个是秒数，另一个是毫秒数。定义如下：

```
struct timeval{
    long tv_sec;   /*秒 */
    long tv_usec;  /*微秒 */
}
```

这个参数的值有三种情况

如果传入 NULL，即不传入时间结构，就是一个阻塞函数，直到某个文件描述符发生变化才会返回。

如果把秒和微妙都设为 0，就是一个非阻塞函数，会立刻返，文件无变化返回 0，有变化返回正值。

如果值大于 0，则意为超时时间， select 若在 timeout 时间内没有检测到文件描述符变化，则会直接返回 0，有变化则返回正值。

使用示例：

```
void main(void)
{
    /* ret 获取返回值, fd 获取文件句柄 */
    int ret, fd;
    /* 定义一个监视文件读变化的描述符合集 */
    fd_set readfds;
    /* 定义一个超时时间结构体 */
    struct timeval timeout;

    /* 获取文件句柄, O_NONBLOCK 表示非阻塞访问 */
    fd = open("dev_xxx", O_RDWR | O_NONBLOCK);

    /* 初始化描述符合集 */
    FD_ZERO(&readfds);
    /* 把文件句柄 fd 指向的文件添加到描述符 */
    FD_SET(fd, &readfds);

    /* 超时时间初始化为 1.5 秒 */
    timeout.tv_sec = 1;
    timeout.tv_usec = 500000;

    /* 调用 select, 注意第一个参数为 fd+1 */
    ret = select(fd + 1, &readfds, NULL, NULL, &timeout);

    switch (ret)
    {
        case 0:
        {
            /* 超时 */
            break;
        }
```

```
32.         case -1:
33.         {
34.             /* 出错 */
35.             break;
36.         }
37.         default:
38.         {
39.             /* 监视的文件可操作 */
40.             /* 判断可操作的文件是不是文件句柄 fd 指向的文件 */
41.             if(FD_ISSET(fd, &readfds))
42.             {
43.                 /* 操作文件 */
44.             }
45.             break;
46.         }
47.     }
48. }
```

在 23 行调用 select 函数之前，做了很多准备工作，主要是 select 函数输入参数的初始化。

注意 11 行 open 函数输入参数中的 O_NONBLOCK 属性，如果需要非阻塞的访问文件，则需要添加这个属性。

41 行，在 ret 返回大于 0 时，使用宏定义 FD_ISSET 判断可操作的句柄是不是我们需要的句柄，在只等待一个文件的情况下，可以不做这个判断。

2) poll

poll 本质上和 sellect 没有区别，但是是他的最大连接数没有限制。

函数原型：

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

参数说明：

fds：struct pollfd 结构体是文件句柄和事件的组合，定义如下：

```
1.  struct pollfd {
2.      int fd;
3.      short events;
4.      short revents;
5.  };
```

fd 是文件句柄，events 是对于这个文件需要监视的事件类型，revents 是内核返回的事件类型。事件类型有：

POLLIN //有数据可读

POLLPRI //有紧急数据可读

POLLOUT //数据可写

POLLERR //指定文件描述符发生错误

POLLHUP //指定文件描述符挂起

POLLNVAL //无效请求

POLLRDNORM //有数据可读

nfds：poll 监视的文件句柄数量，也就是 fds 的数组长度。

timeout：超时时间，单位为毫秒。

使用示例：

```
1.  void main(void)
2.  {
3.      /* ret 获取返回值，fd 获取文件句柄 */
4.      int ret, fd;
5.      /* 定义 struct pollfd 结构体变量 */
6.      struct pollfd fds[1];
7.
8.      /* 非阻塞访问文件 */
9.      fd = open(filename, O_RDWR | O_NONBLOCK);
10.
11.     /* 初始化 struct pollfd 结构体变量 */
12.     fds[0].fd = fd;
13.     fds[0].events = POLLIN;
14.
15.     /* 调用 poll */
16.     ret = poll(fds, sizeof(fds), 1500);
17.     if(ret == 0)
18.     {
19.         /* 超时 */
20.     }
21.     else if (ret < 0)
22.     {
23.         /* 错误 */
24.     }
25.     else
26.     {
27.         /* 操作数据 */
28.     }
29. }
```

3)　epoll

可以理解为 event poll，设计用于大并发时的 IO 查询，常用于网络编程，暂不介绍。

10.1.2 驱动程序中的 poll 函数

应用程序中调用 select、poll、epoll 时，系统调用就会执行驱动程序中 file_operations 的 poll 函数。也就是我们需要实现的函数。圆原型如下：

```
unsigned int (*poll) (struct file *filp, struct poll_table_struct *wait)
```

参数说明：

filp：应用程序传递过来的值，应用程序 open 之后获得的目标文件句柄。

wait：应用程序传递过来的值，代表应用程序线程。我们需要在 poll 函数中调用 poll_wait 将应用程序添线程 wait 添加到 poll_table 等待队列中，poll_wait 函数原型如下：

```
void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
```

wait 作为参数 p 传递给 poll_wait 函数。

返回值：返回值和 struct pollfd 结构体中的事件类型相同。

10.2 实验

这章的实验目标和上一张相同，使用 ps_key1 控制 ps_led1，并使 cpu 占用率相较于第八章降低。

10.2.1 原理图

led 部分和章节 1.3.1 相同。

key 部分和章节 6.1 相同。

10.2.2 设备树

和章节 6.2 相同。

10.2.3 驱动代码

使用 petalinux 新建名为”ax-nio-drv”的驱励程序，并执行 petalinux-config -c rootfs 命令选上新增的驱动程序。

在 ax-nio-drv.c 文件中输入下面的代码：

```
1.  #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/init.h>
4.  #include <linux/ide.h>
5.  #include <linux/types.h>
6.  #include <linux/errno.h>
7.  #include <linux/cdev.h>
8.  #include <linux/of.h>
9.  #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
```

```c
#include <linux/timer.h>
#include <linux/of_irq.h>
#include <linux/irq.h>
#include <linux/wait.h>
#include <linux/poll.h>
#include <asm/uaccess.h>
#include <asm/mach/map.h>
#include <asm/io.h>

/* 设备节点名称 */
#define DEVICE_NAME       "nio_led"
/* 设备号个数 */
#define DEVID_COUNT       1
/* 驱动个数 */
#define DRIVE_COUNT       1
/* 主设备号 */
#define MAJOR_U
/* 次设备号 */
#define MINOR_U           0

/* 把驱动代码中会用到的数据打包进设备结构体 */
struct alinx_char_dev {
/** 字符设备框架 **/
    dev_t              devid;             //设备号
    struct cdev        cdev;              //字符设备
    struct class       *class;            //类
    struct device      *device;           //设备
    struct device_node *nd;               //设备树的设备节点
/** gpio **/
    int                alinx_key_gpio;    //gpio 号
/** 并发处理 **/
    atomic_t           key_sts;           //记录按键状态，为 1 时被按下
/** 中断 **/
    unsigned int       irq;               //中断号
/** 定时器 **/
    struct timer_list  timer;             //定时器
/** 等待队列 **/
    wait_queue_head_t  wait_q_h;          //等待队列头
};
/* 声明设备结构体 */
static struct alinx_char_dev alinx_char = {
    .cdev = {
        .owner = THIS_MODULE,
    },
```

```
60. };
61. 
62. /** 回掉 **/
63. /* 中断服务函数 */
64. static irqreturn_t key_handler(int irq, void *dev)
65. {
66.     /* 按键按下或抬起时会进入中断 */
67.     /* 开启 50 毫秒的定时器用作防抖动 */
68.     mod_timer(&alinx_char.timer, jiffies + msecs_to_jiffies(50));
69.     return IRQ_RETVAL(IRQ_HANDLED);
70. }
71. 
72. /* 定时器服务函数 */
73. void timer_function(unsigned long arg)
74. {
75.     /* value 用于获取按键值 */
76.     unsigned char value;
77.     /* 获取按键值 */
78.     value = gpio_get_value(alinx_char.alinx_key_gpio);
79.     if(value == 0)
80.     {
81.         /* 按键按下，状态置 1 */
82.         atomic_set(&alinx_char.key_sts, 1);
83. /** 等待队列 **/
84.         /* 唤醒进程 */
85.         wake_up_interruptible(&alinx_char.wait_q_h);
86.     }
87.     else
88.     {
89.         /* 按键抬起 */
90.     }
91. }
92. 
93. /** 系统调用实现 **/
94. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
95. static int char_drv_open(struct inode *inode_p, struct file *file_p)
96. {
97.     printk("gpio_test module open\n");
98.     return 0;
99. }
100. 
101. /* read 函数实现，对应到 Linux 系统调用函数的 write 函数 */
102. static ssize_t char_drv_read(struct file *file_p, char __user *buf, size_t len, loff_t *lof
    f_t_p)
```

```
{
    unsigned int keysts = 0;
    int ret;

    /* 读取 key 的状态 */
    keysts = atomic_read(&alinx_char.key_sts);
    /* 判断文件打开方式 */
    if(file_p->f_flags & O_NONBLOCK)
    {
        /* 如果是非阻塞访问，说明以满足读取条件 */
    }
    /* 判断当前按键状态 */
    else if(!keysts)
    {
        /* 按键未被按下(数据未准备好) */
        /* 以当前进程创建并初始化为队列项 */
        DECLARE_WAITQUEUE(queue_mem, current);
        /* 把当前进程的队列项添加到队列头 */
        add_wait_queue(&alinx_char.wait_q_h, &queue_mem);
        /* 设置当前进成为可被信号打断的状态 */
        __set_current_state(TASK_INTERRUPTIBLE);
        /* 切换进程，是当前进程休眠 */
        schedule();

        /* 被唤醒，修改当前进程状态为 RUNNING */
        set_current_state(TASK_RUNNING);
        /* 把当前进程的队列项从队列头中删除 */
        remove_wait_queue(&alinx_char.wait_q_h, &queue_mem);

        /* 判断是否是被信号唤醒 */
        if(signal_pending(current))
        {
            /* 如果是直接返回错误 */
            return -ERESTARTSYS;
        }
        else
        {
            /* 被按键唤醒 */
        }
    }
    else
    {
        /* 按键被按下(数据准备好了) */
    }
```

```
147.
148.     /* 读取 key 的状态 */
149.     keysts = atomic_read(&alinx_char.key_sts);
150.     /* 返回按键状态值 */
151.     ret = copy_to_user(buf, &keysts, sizeof(keysts));
152.     /* 清除按键状态 */
153.     atomic_set(&alinx_char.key_sts, 0);
154.     return 0;
155. }
156.
157. /* poll 函数实现 */
158. unsigned int char_drv_poll(struct file *filp, struct poll_table_struct *wait)
159. {
160.     unsigned int ret = 0;
161.
162.     /* 将应用程序添添加到等待队列中 */
163.     poll_wait(filp, &alinx_char.wait_q_h, wait);
164.
165.     /* 判断 key 的状态 */
166.     if(atomic_read(&alinx_char.key_sts))
167.     {
168.         /* key 准备好了，返回数据可读 */
169.         ret = POLLIN;
170.     }
171.     else
172.     {
173.
174.     }
175.
176.     return ret;
177. }
178.
179. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
180. static int char_drv_release(struct inode *inode_p, struct file *file_p)
181. {
182.     printk("gpio_test module release\n");
183.     return 0;
184. }
185.
186. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
187. static struct file_operations ax_char_fops = {
188.     .owner    = THIS_MODULE,
189.     .open     = char_drv_open,
190.     .read     = char_drv_read,
```

```
191.     .poll    = char_drv_poll,
192.     .release = char_drv_release,
193. };
194. 
195. /* 模块加载时会调用的函数 */
196. static int __init char_drv_init(void)
197. {
198.     /* 用于接受返回值 */
199.     u32 ret = 0;
200. 
201. /** 并发处理 **/
202.     /* 初始化原子变量 */
203.     atomic_set(&alinx_char.key_sts, 0);
204. 
205. /** gpio 框架 **/
206.     /* 获取设备节点 */
207.     alinx_char.nd = of_find_node_by_path("/alinxkey");
208.     if(alinx_char.nd == NULL)
209.     {
210.         printk("alinx_char node not find\r\n");
211.         return -EINVAL;
212.     }
213.     else
214.     {
215.         printk("alinx_char node find\r\n");
216.     }
217. 
218.     /* 获取节点中 gpio 标号 */
219.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinxkey-gpios", 0);
220.     if(alinx_char.alinx_key_gpio < 0)
221.     {
222.         printk("can not get alinxkey-gpios");
223.         return -EINVAL;
224.     }
225.     printk("alinxkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
226. 
227.     /* 申请 gpio 标号对应的引脚 */
228.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinxkey");
229.     if(ret != 0)
230.     {
231.         printk("can not request gpio\r\n");
232.         return -EINVAL;
233.     }
234. 
```

```
235.     /* 把这个 io 设置为输入 */
236.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
237.     if(ret < 0)
238.     {
239.         printk("can not set gpio\r\n");
240.         return -EINVAL;
241.     }
242.
243. /** 中断 **/
244.     /* 获取中断号 */
245.     alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
246.     /* 申请中断 */
247.     ret = request_irq(alinx_char.irq,
248.                       key_handler,
249.                       IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
250.                       "alinxkey",
251.                       NULL);
252.     if(ret < 0)
253.     {
254.         printk("irq %d request failed\r\n", alinx_char.irq);
255.         return -EFAULT;
256.     }
257.
258. /** 定时器 **/
259.     alinx_char.timer.function = timer_function;
260.     init_timer(&alinx_char.timer);
261.
262. /** 等待队列 **/
263.     init_waitqueue_head(&alinx_char.wait_q_h);
264.
265. /** 字符设备框架 **/
266.     /* 注册设备号 */
267.     alloc_chrdev_region(&alinx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
268.
269.     /* 初始化字符设备结构体 */
270.     cdev_init(&alinx_char.cdev, &ax_char_fops);
271.
272.     /* 注册字符设备 */
273.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
274.
275.     /* 创建类 */
276.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
277.     if(IS_ERR(alinx_char.class))
278.     {
```

```
279.         return PTR_ERR(alinx_char.class);
280.     }
281.
282.     /* 创建设备节点 */
283.     alinx_char.device = device_create(alinx_char.class, NULL,
284.                                       alinx_char.devid, NULL,
285.                                       DEVICE_NAME);
286.     if (IS_ERR(alinx_char.device))
287.     {
288.         return PTR_ERR(alinx_char.device);
289.     }
290.
291.     return 0;
292. }
293.
294. /* 卸载模块 */
295. static void __exit char_drv_exit(void)
296. {
297. /** gpio **/
298.     /* 释放 gpio */
299.     gpio_free(alinx_char.alinx_key_gpio);
300.
301. /** 中断 **/
302.     /* 释放中断 */
303.     free_irq(alinx_char.irq, NULL);
304.
305. /** 定时器 **/
306.     /* 删除定时器 */
307.     del_timer_sync(&alinx_char.timer);
308.
309. /** 字符设备框架 **/
310.     /* 注销字符设备 */
311.     cdev_del(&alinx_char.cdev);
312.
313.     /* 注销设备号 */
314.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
315.
316.     /* 删除设备节点 */
317.     device_destroy(alinx_char.class, alinx_char.devid);
318.
319.     /* 删除类 */
320.     class_destroy(alinx_char.class);
321.
322.     printk("timer_led_dev_exit_ok\n");
```

```
323. }
324.
325. /* 标记加载、卸载函数 */
326. module_init(char_drv_init);
327. module_exit(char_drv_exit);
328.
329. /* 驱动描述信息 */
330. MODULE_AUTHOR("Alinx");
331. MODULE_ALIAS("alinx char");
332. MODULE_DESCRIPTION("NIO LED driver");
333. MODULE_VERSION("v1.0");
334. MODULE_LICENSE("GPL");
```

驱动代码在上一章的代码基础上，增加了 poll 实现，并稍微修改了 read 函数。

191 行在 file_operations 结构体中添加 poll 函数实现。

158 行实现 poll 函数，调用一下 poll_wait 函数，之后哦按段数据状态，如果数据准备好，就返回 POLLIN 状态标识。

110 行在 read 函数中稍作修改，先判断文件打开方式，如果是非阻塞的方式访问，就不去做队列相关的操作了，直接返回数据给用户，否则按照阻塞访问处理。

10.2.4 测试代码

测试代码在 6.4 节的基础上修改，新建 QT 工程名为”ax_nioled_test”，新建 main.c，输入下列代码：

```
1.  #include "stdio.h"
2.  #include "unistd.h"
3.  #include "sys/types.h"
4.  #include "sys/stat.h"
5.  #include "fcntl.h"
6.  #include "stdlib.h"
7.  #include "string.h"
8.  #include "poll.h"
9.  #include "sys/select.h"
10. #include "sys/time.h"
11. #include "linux/ioctl.h"
12.
13. int main(int argc, char *argv[])
14. {
15.
16.     /* ret 获取返回值, fd 获取文件句柄 */
17.     int ret, fd, fd_1;
18.     /* 定义一个监视文件读变化的描述符合集 */
19.     fd_set readfds;
```

```
    /* 定义一个超时时间结构体 */
    struct timeval timeout;

    char *filename, led_value = 0;
    unsigned int key_value;

    if(argc != 2)
    {
        printf("Error Usage\r\n");
        return -1;
    }

    filename = argv[1];
    /* 获取文件句柄，O_NONBLOCK 表示非阻塞访问 */
    fd = open(filename, O_RDWR | O_NONBLOCK);
    if(fd < 0)
    {
        printf("can not open file %s\r\n", filename);
        return -1;
    }

    while(1)
    {
        /* 初始化描述符合集 */
        FD_ZERO(&readfds);
        /* 把文件句柄 fd 指向的文件添加到描述符 */
        FD_SET(fd, &readfds);

        /* 超时时间初始化为 1.5 秒 */
        timeout.tv_sec = 1;
        timeout.tv_usec = 500000;

        /* 调用 select，注意第一个参数为 fd+1 */
        ret = select(fd + 1, &readfds, NULL, NULL, &timeout);
        switch (ret)
        {
            case 0:
            {
                /* 超时 */
                break;
            }
            case -1:
            {
                /* 出错 */
```

```
64.                 break;
65.             }
66.             default:
67.             {
68.                 /* 监视的文件可操作 */
69.                 /* 判断可操作的文件是不是文件句柄 fd 指向的文件 */
70.                 if(FD_ISSET(fd, &readfds))
71.                 {
72.                     /* 操作文件 */
73.                     ret = read(fd, &key_value, sizeof(key_value));
74.                     if(ret < 0)
75.                     {
76.                         printf("read failed\r\n");
77.                         break;
78.                     }
79.                     printf("key_value = %d\r\n", key_value);
80.                     if(1 == key_value)
81.                     {
82.                         printf("ps_key1 press\r\n");
83.                         led_value = !led_value;
84.
85.                         fd_l = open("/dev/gpio_leds", O_RDWR);
86.                         if(fd_l < 0)
87.                         {
88.                             printf("file /dev/gpio_leds open failed\r\n");
89.                             break;
90.                         }
91.
92.                         ret = write(fd_l, &led_value, sizeof(led_value));
93.                         if(ret < 0)
94.                         {
95.                             printf("write failed\r\n");
96.                             break;
97.                         }
98.
99.                         ret = close(fd_l);
100.                        if(ret < 0)
101.                        {
102.                            printf("file /dev/gpio_leds close failed\r\n");
103.                            break;
104.                        }
105.                    }
106.                }
107.                break;
```

```
108.             }
109.         }
110.     }
111.     close(fd);
112.     return ret;
113. }
```

73 行的 read 函数开始，之后的代码与 6.4 节是一样的，通过判断 key 的状态，来改变 led 的状态。

在调用 read 之前，先调用 select 函数来检测数据状态，用法和 10.1.1 节中的用法相同，就不重复说明了。

10.2.5 运行测试

测试方式和现象和上一章一样，步骤如下：

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-concled-drv.ko
insmod ./ax-nio-drv.ko
cd ./build-ax_nioled_test-ZYNQ-Debug
./ax-key-test /dev/nio_led&
top
```

此外，可以尝试一下，把测试程中的超时时间改成 0 或者 NULL 来贯彻现象。