# “菜鸟玩转嵌入式”视频培训讲座
## — Linux驱动开发基础班

主办：上海申嵌信息科技有限公司
承办：嵌入式家园
协办：上海嵌入式家园-开发板商城
广州友善之臂计算机科技有限公司
主讲：贺光辉（嵌入式系统工程师）

嵌入式家园 www.embedclub.com

上海嵌入式家园-开发板商城 http://embedclub.taobao.com/

# 上章回顾


- 从并发的需要，引起的竞争状态，到解决竞争状态的方法机制:
  - 禁止中断
  - 信号量
  - 自旋锁
  - completion
  - 原子操作

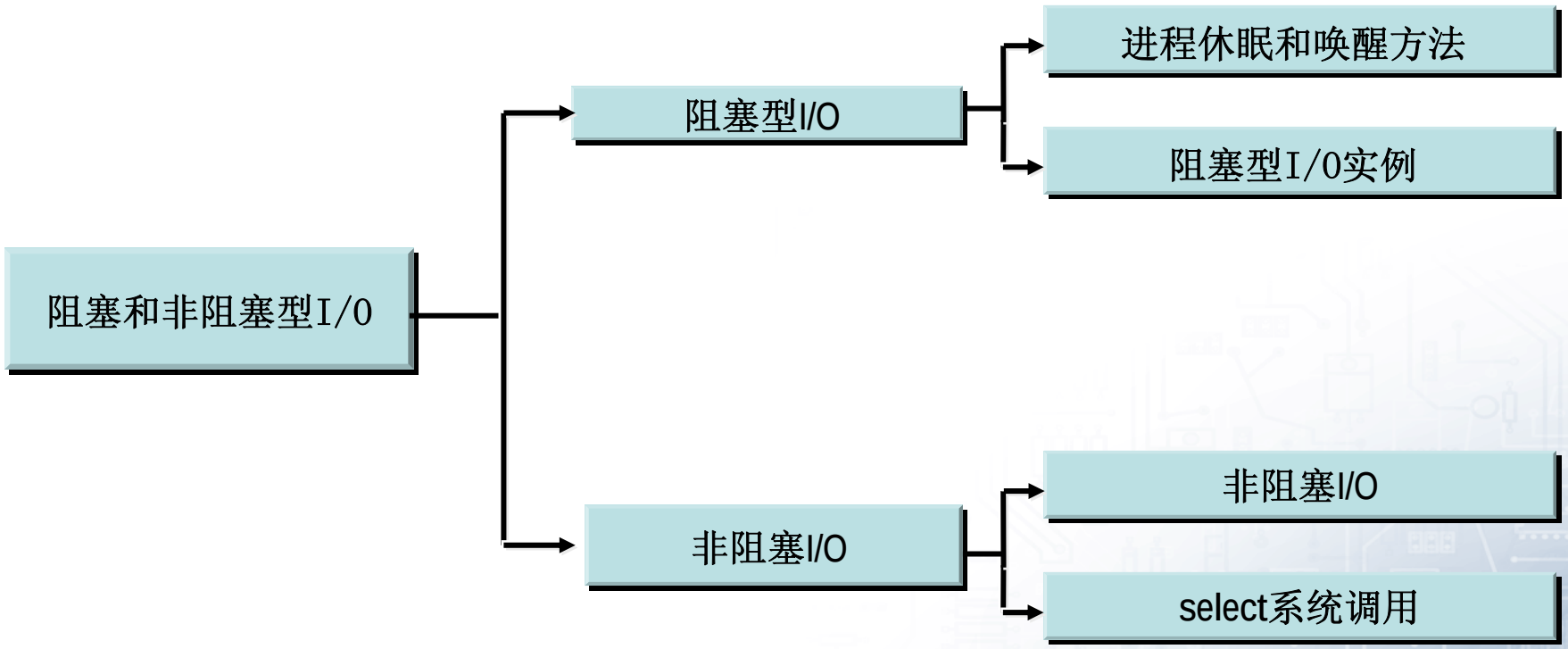

# 第4章

# 阻塞和非阻塞型I/O

# 本章目标

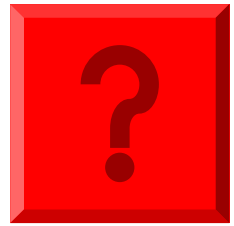

- 掌握进程睡眠和唤醒的方法
- 掌握阻塞型I/O的实现方法
- 掌握select系统调用的实现方法

# 本章结构

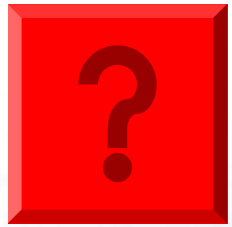


# 4-1 阻塞型I/O


当用户程序调用read函数时，驱动程序的read并没有准备好数据，怎么办？



当用户程序调用write时，驱动程序的write的缓冲区已满，此时怎么办？



1. 用户程序不会管理这些问题。
2. 驱动程序默认情况下，阻塞该进程。将其置入休眠状态直到请求可继续。
3. 或者直接返回，这是非阻塞I/O。

# 4-1-1进程休眠和唤醒方法


- 休眠的意义
  - 从调度器的运行队列→某个等待队列
  - 直到等到某个事件发生，再从等待队列返回到运行队列。

- 如何将进程安全的进入休眠状态？
  - 不能在原子上下文进行休眠
  - 休眠时，对外界一无所知，进程必须重新检测等待条件
  - 进程只有确保会被其他进程唤醒，才能进入休眠

- Linux的休眠机制
  - 进程经常由于某种条件没有得到满足而不得不进入睡眠状态，然后等待条件得到满足的时候再继续运行，进入运行状态。这种需求需要等待队列机制的支持。

等待队列就是一个进程链表，其中包含了等待某个特定事件的所有进程

# 4-1-1 等待队列


- 在Linux驱动程序设计中，可以使用等待队列来实现进程的阻塞，等待队列可看作保持进程的容器，在阻塞进程时，将进程放入等待队列，当唤醒进程时，从等待队列中取出进程。
- 每个等待任务都会抽象成一个wait_queue，并且挂载到wait_queue_head上。
- Linux中等待队列的实现思想：当一个任务需要在某个wait_queue_head上睡眠时，将自己的进程控制块信息封装到wait_queue中，然后挂载到wait_queue_head的链表中，执行调度睡眠。当某些事件发生后，另一个任务（进程）会唤醒wait_queue_head上的某个或者所有任务，唤醒工作也就是将等待队列中的任务设置为可调度的状态，并且从队列中删除。

# 4-1-1进程休眠和唤醒方法


- 等待队列通过”等待队列头”来管理
  - 类型：struct wait_queue_head_t
  - 定义在<linux/wait.h>中
  - 定义等待队列

```
wait_queue_head_t xxx_queue;
```

  - 初始化等待队列：

```
init_waitqueue_head(&xxx_queue);
```

  - 定义并初始化等待队列

```
DECLARE_WAIT_QUEUE_HEAD(xxx_queue);
```

# 4-1-1进程休眠和唤醒方法


- 等待事件
  - 当进程休眠时，等待被另外进程唤醒，同时检查进程等待的条件是否为真，若为真，则被调度执行；反之继续睡眠。
  - 等待事件，即为进程等待的条件（condition）。
- 休眠函数

等待队列头，通过"值传递"

休眠前后都要对该表达式求值；在条件为真之前，进程会保持休眠

```
wait_event(queue，condition)；
wait_event_interruptible(queue，condition)；
```

等待限定的jiffs时间达到后返回0值，无论condition为何值

```
wait_event_timeout(queue，condition，timeout)；
wait_event_interruptible_timeout(queue, condition，timeout)；
```

# 4-1-1进程休眠和唤醒方法


- 唤醒函数

唤醒所有的等待进程

```
wake_up(wait_queue_head_t *queue);
wake_up_interruptible(wait_queue_head_t *queue);
```

唤醒nr个独占等待进程，而不是一个，当nr=0时，唤醒所有的独占等待进程。

```
wake_up_nr(wait_queue_head_t *queue，int nr);
wake_up_interruptible_nr(wait_queue_head_t *queue，int nr);
```

被唤醒后强制调度重新执行原休眠进程

```
wake_up_interruptible_sync(wait_queue_head_t *queue);
```

# 4-1-1进程休眠和唤醒方法

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;
ssize_t sleepy_read(struct file *filp, char __user *buf,
size_t count,loff t *pos)
{
    printk(KERN_DEBUG "process %i  (%s)  going to sleep\n",
           current->pid, current->comm);
    wait_event_interruptible(wq, flag != 0);
    flag = 0;
    printk(KERN_DEBUG "awoken %i  (%s)\n", current->pid, current->comm);
    return 0;
}
```

定义并初始化 wait_queue_head_t

打印出是哪个进程调用驱动程序。

在调用前后都要检查condition，如果满足条件，就不再休眠，否则进入休眠状态

# 4-1-1进程休眠和唤醒方法


```
ssize_t sleepy_write(struct file *filp, const char __user *buf,
size_t count,loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
           current->pid, current->comm);
    flag = 1;
    wake_up_interruptible(&wq);
    return count;
}
```

查看是哪个进程调用驱动的write函数

唤醒休眠在wq队列上的所有的进程。

# 4-1-1进程休眠和唤醒方法


- 上面实例中存在一个概率很小的竞态条件
  - A,B 进程都等在wq的队列上
  - C进程调用wake_up_interruptible
  - A进程被唤醒，检查条件flag !=0 成立
  - 此时，调度到B进程
  - B进程也检查到flag != 0成立
  - 这样，一个事件唤醒了两个进程，可能产生竞态
- 可以用原子操作防止这种情况

# 4-1-1进程休眠和唤醒方法


- 设置进程休眠的内部细节
  - 分配并初始化一个wait_queue_t结构
    - 包含休眠进程的信息，以及期望被唤醒的相关细节
  - 设置进程的状态，将其标记为休眠状态
    - TASK_INTERRUTIBLE // 可中断
    - TASK_UNINTERRUPTIBLE // 不可中断

```
void set_current_state(int new_state); //手动设置进程状态
```

```
current->state=TASK_INTERRUPTIBLE; //老内核版本设置方式
```

  - 让出处理器

```
if(!condition)
    schedule(); // 执行调度，让出处理器
```

# 4-1-1进程休眠和唤醒方法


- 手工休眠
  - 早期版本的方法
  - 读者可以自行了解
- 唤醒等待队列可能发生的情况
  - 当调用wake_up时，所有等待在该队列上的进程都被唤醒，并进入可运行状态
  - 如果只有一个进程可获得资源，此时，其他的进程又将再次进入休眠
  - 如果数量很大，被称为”疯狂兽群”

# 4-1-1进程休眠和唤醒方法


- 独占等待
  - **与普通休眠的不同**
    - 等待队列入口设置了WQ_FLAG_EXCLUSIVE标志时，则会被添加到等待队列的尾部。而没有这个标志的入口会被添加到头部。
    - 在某个等待队列上调用wake_up时，它会在唤醒第一个具有WQ_FLAG_EXCLUSIVE标志的进程之后停止唤醒其他独占进程。
  - **使进程进入独占等待函数：**

```
void prepare_to_wait_exclusive(wait_queue_head_t *queue
                               wait_queue_t *wait，int state)；
```

使用wait_event的变种
都无法使用独占等待

# 4-1-1进程休眠和唤醒方法


- 无条件休眠调用（老版本，建议不再使用）

void sleep_on(wait_queue_head_t *queue)；
让进程进入不可中断的睡眠，并把它放入等待队列queue。

void interrupt_sleep_on(wait_queue_head_t *queue)；
让进程进入可中断的睡眠，并把它放入等待队列queue。

# 4-1-2 阻塞方式


- 在阻塞型驱动程序中，read实现方式如下：如果进程调用read，但是设备没有数据或数据不足，进程阻塞。当新数据到达后，唤醒被阻塞进程。

# 4-1-2 阻塞方式


- 在阻塞型驱动程序中，write实现方式如下：如果进程调用write，但是设备没有足够的空间供其写入数据，进程阻塞。当设备中的数据被读走后，缓冲区中空出部分空间，则唤醒被阻塞进程。

# 4-1-2阻塞型I/O实例

```
struct xxx_pipe{
    wait_queue_head_t inq，outq；      /*读取和写入队列*/
    char *buffer，*end；               /*缓冲区的起始和结尾*/
    int buffersize；                   /*用于指针计算*/
    char *rp，*wp；                    /*读取和写入的位置*/
    int nreaders，nwriters；           /*用于读写打开的数量*/
    struct fasync_struct *async_queue；/*异步读取者*/
    struct semaphore sem；             /*互斥信号量*/
    struct cdev cdev；                 /*字符设备结构*/
};
```

# 4-1-2阻塞型IO实例


```
static ssize_t xxx_read(struct file *filp，char __user *buf，
                        size_t count,loff_t *f_pos)
{
    struct xxx_pipe *dev=filp->private_data；
    if(down_interruptible(&dev->sem))
        return -ERESTARTSYS；
     while(dev->rp == dev->wp) { /*无数据可读取*/
        up(&dev->sem)；/*释放锁*/
        if(filp->f_flags & O_NONBLOCK)
            return -EAGAIN；
        if(wait_event_interruptible(dev->inq，(dev->rp!=dev->wp))
            return -ERESTARTSYS；/*信号，通知fs层做相应处理*/
        /*否则循环，但首先获取锁*/
        if(down_interruptible(&dev->sem))
            return -ERESTARTSYS；
    }
```

# 4-1-2阻塞型IO实例


```
/*数据已就绪，返回*/
  if(dev->wp > dev->rp)
      count=min(count，(size_t)(dev->wp - dev->rp))；
  else/*写入指针回卷，返回数据直到dev->end*/
      count=min(count，(size_t)(dev->end – dev->rp))；
  if(copy_to_user(buf，dev->rp，count)){
    up(&dev->sem)；
    return -EFAULT；
  }
  dev->rp += count：
  if(dev->rp == dev->end)
    dev->rp = dev->buffer；/*回卷*/
  up(&dev->sem)；
  /*最后，唤醒所有写入者并返回*/
  wake_up_interruptible(&dev->outq)；
  return count；
}
```

# 阶段总结


- 介绍了阻塞型I/O的实现方法
- 介绍了进程休眠和唤醒的方法，重点介绍了wait_event和wake_up

# 4-2 非阻塞方式


- 阻塞方式是文件读写操作的默认方式，但应用程序员可通过使用O_NONBLOCK标志来人为的设置读写操作为非阻塞方式（该标志定义在<linux/fcntl.h>中，在打开文件时指定）。

# 4-2 非阻塞方式


- 如果设置了O_NONBLOCK标志，read和write的行为是不同的。如果进程在没有数据就绪时调用了read，或者在缓冲区没有空间时调用了write，系统只是简单地返回-EAGAIN，而不会阻塞进程。

# 4-2 非阻塞I/O的设置


- 调用进程显式的指明不想阻塞
- 设置filp->f_flag |= O_NONBLOCK

```
// 应用程序以非阻塞I/O的方式从串口驱动读取一个字符
int fd;
char buf;
fd = open("/dev/ttySAC0",O_RDWR | O_NONBLOCK);
…
while(read(fd, &buf, 1) != 1);/*串口上无输出也返回，所以要循环尝试读取串口*/
   printf("%c\n",buf);
```

# 4-2-1 poll和select


- poll、select都允许进程决定是否可以对一个或多个打开的文件做非阻塞的读取或写入
- 这些调用也会阻塞进程，直到给定的文件描述符集合中的任何一个可读取或写入
- 常用于那些要使用多个输入或输出流而又不会阻塞于其中任何一个流的应用程序中
- select – 在BSD Unix中引入
- poll – 由AT&T Unix System V引入

# 4-2-1 poll和select

- 都是调用驱动程序的poll来实现的

返回可以立即执行操作的位掩码

<linux/poll.h>中声明,驱动程序不需要了解该结构的细节

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

- poll函数负责完成两个步骤：
  - 在一个或多个可指示poll状态变化的等待队列上调用poll_wait
    - 如果当前没有文件描述符可用来执行I/0，则内核将使进程在传递到该系统调用的所有文件描述符对应的等待队列上等待。

```
void poll_wait(struct file *，wait_queue_head_t *，poll_table *);
```

  - 返回一个用来描述操作是否可以立即无阻塞执行的位掩码

即：使用pool_wait将等待队列添加到poll_table中

# 4-2-1 位掩码


- <linux/poll.h>定义poll位掩码：
- POLLIN
  - 设备可读
- POLLRDNORM
  - 数据可读（ "normal" data is available for reading ）
- POLLOUT
  - 设备可写
- POLLWRNORM
  - 数据可写（ "normal" data is available for writing ）
- 一般设备可读返回： POLLIN | POLLRDNORM
- 一般设备可写返回： POLLOUT | POLLWRNORM

# 4-2-1 范例


```
static unsigned int xxx_poll(struct file *filp，poll_table *wait)
{
        struct xxx_pipe *dev = filp->private_data；
        unsigned int mask=0；
        down(&dev->sem)；

        poll_wait(filp，&dev->inq，wait)；
        poll_wait(filp，&dev->outq，wait)；
        if(read_buffer_not_empty) //如果接收buffer不为空，可读
                mask |= POLLIN | POLLRDNORM；     /*可读取*/
        if(write_buffer_not_full)     //如果写buffer不满，可写
                mask |= POLLOUT | POLLWRNORM；     /*可写入*/

        up(&dev->sem)；
        return mask；//返回位掩码
}
```

增加两个等待队列到poll_table中

# 4-2-1 select系统调用


- select系统调用用于多路监控，当没有一个文件满足要求时，select将阻塞调用进程。

```
int select( int maxfd, fd_set  *readfds, fd_set   *writefds,
                fd_set *exceptfds, struct timeval *timeout);
```

```
struct timeval
{
    int tv_sec; /*秒*/
    int tv_usec; /*微妙*/
};
```

# 4-2-1 select系统调用


- maxfd
  - 文件描述符的范围，比待检测的最大文件描述符大1
- readfds
  - 被读监控的文件描述符集
- writefds
  - 被写监控的文件描述符集
- exceptfds
  - 被异常监控的文件描述符集
- timeout
  - 定时器

# 4-2-1 select系统调用（参数）


- timeout取不同的值，该调用有不同的表现：
  - timeout值为0，不管是否有文件满足要求，都立刻返回，无文件满足要求返回0，有文件满足要求返回一个正值。
  - timeout为NULL，select将阻塞进程，直到某个文件满足要求。
  - timeout值为正整数，就是等待的最长时间，即select在timeout时间内阻塞进程。

# 4-2-1 select系统调用（返回值）


- select调用返回时，返回值有如下情况：
  - 正常情况下返回满足要求的文件描述符个数；
  - 经过了timeout等待后仍无文件满足要求，返回值为0；
  - 如果select被某个信号中断，它将返回-1并设置errno为EINTR；
  - 如果出错，返回-1并设置相应的errno。

# 4-2-1 select系统调用（使用方法）


- 将要监控的文件添加到文件描述符集
- 调用select开始监控
- 判断文件是否发生变化

# 4-2-1 select系统调用（使用方法）


- 系统提供4个宏对描述符集进行操作：

```
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);
```
判断文件描述符是否被置位

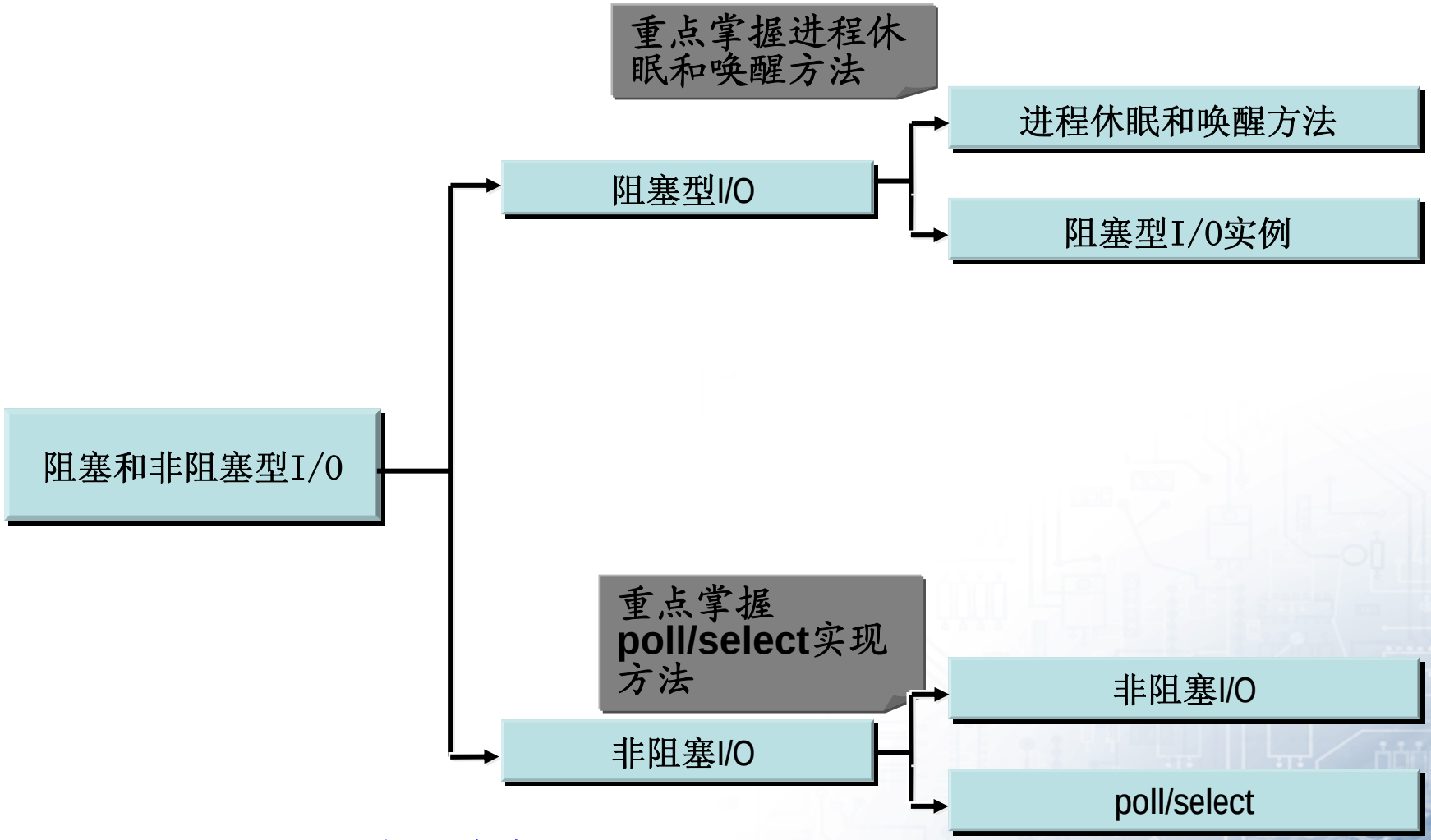
# 4-2 非阻塞I/O方式轮询设备范例


```
int fd;
fd_set rfds,wfds;//读/写文件描述符集
/*以非阻塞方式打开/dev/xxx设备文件*/
fd = open("/dev/xxx", O_RDWR | O_NONBLOCK);
FD_ZERO(&rfds);
FD_ZERO(&wfds);
FD_SET(fd, &rfds);
FD_SET(fd, &wfds);
select(fd + 1, &rfds, &wfds, NULL, NULL);
/*数据可获得*/
if(FD_ISSET(fd, &rfds))
{
        //读数据
}
if(FD_ISSET(fd, &wfds))
{
        //写数据
}
```

# 阶段总结

- poll/select在驱动程序中的实现
- 应用层如何调用驱动中的poll/select

# 本章总结



- **任务一、阻塞型字符设备驱动实验**
  - 修改Beep驱动，在设备驱动读函数中实现阻塞，在设备驱动写函数中实现唤醒。读进程执行时，蜂鸣器鸣叫，写进程执行时，蜂鸣器关闭。
- **任务二、在memdev驱动中，引入阻塞型机制**
  - 修改read和wirte函数，实现阻塞型I/O访问
- **任务三、在memdev驱动中，引入非阻塞型机制**
  - 在驱动中实现poll函数，实现非阻塞型I/O访问