# Linux阻塞型程序设计

## 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，等待队列可以看做保存进程的容器，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。

Linux2.6内核提供了如下关于等待队列的操作

1. 定义等待队列

wait\_queue\_head\_t my\_queue

1. 初始化等待队列

init\_waitqueue\_head(&my\_queue)

1. 定义并初始化等待队列

DECLARE\_WAIT\_QUEUE\_HEAD(my\_queue)

1. 有条件睡眠

wait\_event(queue,condition)

当condition为真时，立即返回；否则让进程进入TASK\_UNINTERRUPTIBLE模式的睡眠

wait\_event\_interruptible(queue,condition)

当condition为真时，立即返回；否则让进程进入TASK\_INTERRUPTIBLE模式的睡眠

int wait\_event\_killable(wait\_queue\_t queue, condition)

当condition为真时，立即返回；否则让进程进入TASK\_KILLABLE模式的睡眠

1. 无条件睡眠

sleep\_on(wait\_queue\_head\_t \*q)

让进程进入不可中断的睡眠

interruptible\_sleep\_on(wait\_queue\_head\_t \*q)

让进程进入可中断的睡眠

1. 从等待队列中唤醒进程

wake\_up(wait\_queue\_t \*q)

可以唤醒所有模式的等待队列

wake\_up\_interruptible(wait\_queue\_t \*q)

唤醒TASK\_INTERRUPTIBLE模式的进程

## 代码示例

#include <linux/module.h>

#include <linux/types.h>

#include <linux/fs.h>

#include <linux/errno.h>

#include <linux/mm.h>

#include <linux/sched.h>

#include <linux/init.h>

#include <linux/cdev.h>

#include <asm/io.h>

#include <asm/system.h>

#include <asm/uaccess.h>

#include "memdev.h"

static mem\_major = MEMDEV\_MAJOR;

bool have\_data = false; /\*表明设备有足够数据可供读\*/

module\_param(mem\_major, int, S\_IRUGO);

struct mem\_dev \*mem\_devp; /\*设备结构体指针\*/

struct cdev cdev;

/\*文件打开函数\*/

int mem\_open(struct inode \*inode, struct file \*filp)

{

struct mem\_dev \*dev;

/\*获取次设备号\*/

int num = MINOR(inode->i\_rdev);

if (num >= MEMDEV\_NR\_DEVS)

return -ENODEV;

dev = &mem\_devp[num];

/\*将设备描述结构指针赋值给文件私有数据指针\*/

filp->private\_data = dev;

return 0;

}

/\*文件释放函数\*/

int mem\_release(struct inode \*inode, struct file \*filp)

{

return 0;

}

/\*读函数\*/

static ssize\_t mem\_read(struct file \*filp, char \_\_user \*buf, size\_t size, loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct mem\_dev \*dev = filp->private\_data; /\*获得设备结构体指针\*/

/\*判断读位置是否有效\*/

if (p >= MEMDEV\_SIZE)

return 0;

if (count > MEMDEV\_SIZE - p)

count = MEMDEV\_SIZE - p;

while (!have\_data) /\* 没有数据可读，考虑为什么不用if，而用while,中断信号唤醒 \*/

{

if (filp->f\_flags & O\_NONBLOCK)

return -EAGAIN;

wait\_event\_interruptible(dev->inq,have\_data);

}

/\*读数据到用户空间\*/

if (copy\_to\_user(buf, (void\*)(dev->data + p), count))

{

ret = - EFAULT;

}

else

{

\*ppos += count;

ret = count;

printk(KERN\_INFO "read %d bytes(s) from %d\n", count, p);

}

have\_data = false; /\* 表明不再有数据可读 \*/

return ret;

}

/\*写函数\*/

static ssize\_t mem\_write(struct file \*filp, const char \_\_user \*buf, size\_t size, loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct mem\_dev \*dev = filp->private\_data; /\*获得设备结构体指针\*/

/\*分析和获取有效的写长度\*/

if (p >= MEMDEV\_SIZE)

return 0;

if (count > MEMDEV\_SIZE - p)

count = MEMDEV\_SIZE - p;

/\*从用户空间写入数据\*/

if (copy\_from\_user(dev->data + p, buf, count))

ret = - EFAULT;

else

{

\*ppos += count;

ret = count;

printk(KERN\_INFO "written %d bytes(s) from %d\n", count, p);

}

have\_data = true; /\* 有新的数据可读 \*/

/\* 唤醒读进程 \*/

wake\_up(&(dev->inq));

return ret;

}

/\* seek文件定位函数 \*/

static loff\_t mem\_llseek(struct file \*filp, loff\_t offset, int whence)

{

loff\_t newpos;

switch(whence) {

case 0: /\* SEEK\_SET \*/

newpos = offset;

break;

case 1: /\* SEEK\_CUR \*/

newpos = filp->f\_pos + offset;

break;

case 2: /\* SEEK\_END \*/

newpos = MEMDEV\_SIZE -1 + offset;

break;

default: /\* can't happen \*/

return -EINVAL;

}

if ((newpos<0) || (newpos>MEMDEV\_SIZE))

return -EINVAL;

filp->f\_pos = newpos;

return newpos;

}

/\*文件操作结构体\*/

static const struct file\_operations mem\_fops =

{

.owner = THIS\_MODULE,

.llseek = mem\_llseek,

.read = mem\_read,

.write = mem\_write,

.open = mem\_open,

.release = mem\_release,

};

/\*设备驱动模块加载函数\*/

static int memdev\_init(void)

{

int result;

int i;

dev\_t devno = MKDEV(mem\_major, 0);

/\* 静态申请设备号\*/

if (mem\_major)

result = register\_chrdev\_region(devno, 2, "memdev");

else /\* 动态分配设备号 \*/

{

result = alloc\_chrdev\_region(&devno, 0, 2, "memdev");

mem\_major = MAJOR(devno);

}

if (result < 0)

return result;

/\*初始化cdev结构\*/

cdev\_init(&cdev, &mem\_fops);

cdev.owner = THIS\_MODULE;

cdev.ops = &mem\_fops;

/\* 注册字符设备 \*/

cdev\_add(&cdev, MKDEV(mem\_major, 0), MEMDEV\_NR\_DEVS);

/\* 为设备描述结构分配内存\*/

mem\_devp = kmalloc(MEMDEV\_NR\_DEVS \* sizeof(struct mem\_dev), GFP\_KERNEL);

if (!mem\_devp) /\*申请失败\*/

{

result = - ENOMEM;

goto fail\_malloc;

}

memset(mem\_devp, 0, sizeof(struct mem\_dev));

/\*为设备分配内存\*/

for (i=0; i < MEMDEV\_NR\_DEVS; i++)

{

mem\_devp[i].size = MEMDEV\_SIZE;

mem\_devp[i].data = kmalloc(MEMDEV\_SIZE, GFP\_KERNEL);

memset(mem\_devp[i].data, 0, MEMDEV\_SIZE);

/\*初始化等待队列\*/

init\_waitqueue\_head(&(mem\_devp[i].inq));

}

return 0;

fail\_malloc:

unregister\_chrdev\_region(devno, 1);

return result;

}

/\*模块卸载函数\*/

static void memdev\_exit(void)

{

cdev\_del(&cdev); /\*注销设备\*/

kfree(mem\_devp); /\*释放设备结构体内存\*/

unregister\_chrdev\_region(MKDEV(mem\_major, 0), 2); /\*释放设备号\*/

}

MODULE\_AUTHOR("David Xie");

MODULE\_LICENSE("GPL");

module\_init(memdev\_init);

module\_exit(memdev\_exit);