

# 操作系统大作业实验 6 报告

高艺轩 2022010639

## 1 实验题目

选择的题目为“管道驱动程序开发”。

## 2 问题描述

管道是现代操作系统中重要的进程间通信（IPC）机制之一，Linux 和 Windows 操作系统都支持管道。

管道在本质上就是在进程之间以字节流方式传送信息的通信通道，每个管道具有两个端，一端用于输入，一端用于输出，如下图所示。在相互通信的两个进程中，一个进程将信息送入管道的输入端，另一个进程就可以从管道的输出端读取该信息。显然，管道独立于用户进程，所以只能在内核态下实现。

在本实验中，请通过编写设备驱动程序 mypipe 实现自己的管道，并通过该管道实现进程间通信。

你需要编写一个设备驱动程序 mypipe 实现管道，该驱动程序创建两个设备实例，一个针对管道的输入端，另一个针对管道的输出端。另外，你还需要编写两个测试程序，一个程序向管道的输入端写入数据，另一个程序从管道的输出端读出数据，从而实现两个进程间通过你自己实现的管道进行数据通信。

## 3 思路与程序结构

实现 Linux 管道驱动的核心是编写一个 character device 驱动，我们将它通过模块的形式挂载到内核中。实验内核版本为 6.14.3-arch1-1。作为一个管道程序，我们需要向系统注册两个设备，一个作为输出，一个作为输入；同时，需要开辟一片缓冲区，用于暂时接收从管道入口发来的信息，在这里我选择使用了环形缓存（ring buffer）。

首先，定义一些变量用于存储状态：

```
10 | // BEGIN_LST_DEF
```

```

11 #define MYPIPE_BUFFER_SIZE 4096
12 #define MYPIPE_DEVICE_NAME "wendy"
13
14 static dev_t dev_major = 0;
15
16 enum MyPipeMinors {
17     MYPIPE_MINOR_IN = 0,
18     MYPIPE_MINOR_OUT,
19     MYPIPE_MINOR_COUNT,
20 };
21
22 struct mypipe_data {
23     struct cdev cdev;
24     // Whether opened by some writer
25     bool is_active;
26     u8 buffer[MYPIPE_BUFFER_SIZE];
27     size_t write_pos;
28     size_t read_pos;
29     size_t data_len;
30 };
31
32 // This could be an array if we want multiple cdev
33 static struct mypipe_data pdata;
34
35 static struct class *mypipedev_class = NULL;
36
37 // Protects the buffer
38 static DEFINE_MUTEX(mypipe_lock);
39 // Protects the is_active value
40 static DEFINE_MUTEX(isactive_lock);
41 // These are two wait_queue_head_t for storing tasks that are waiting for
42 // writing/ reading
43 static DECLARE_WAIT_QUEUE_HEAD(read_queue);
44 static DECLARE_WAIT_QUEUE_HEAD(write_queue);
45 // END_LST_DEF

```

作为一个设备驱动模块,我们要做的最主要的事情就是提供module\_init和module\_exit两个回调函数。我们的驱动程序将在这两个module\_init中初始化自身,向操作系统注册我们创建的设备以及注册另一个重要的回调函数结构体mypipe\_fops,同时在module\_exit中做相反的操作。模块初始化与销毁的代码如下:

```

225 // BEGIN_LST_INIT
226 static int __init mypipe_init(void) {
227     int ret;

```

```

228 dev_t dev;
229
230 // Acquire a region of major, minor for us to use. The name appears in
231 // /proc/devices (and potentially /sys ?).
232 ret = alloc_chrdev_region(&dev, 0, MYPIPE_MINOR_COUNT, MYPIPE_DEVICE_NAME)
    ;
233 if (ret) {
234     return ret;
235 }
236
237 dev_major = MAJOR(dev);
238
239 // Create the class in sysfs. The name appears in /sys/class/<name>
240 mypipdev_class = class_create(MYPIPE_DEVICE_NAME);
241 if (IS_ERR(mypipdev_class)) {
242     ret = PTR_ERR(mypipdev_class);
243     goto del_cdev;
244 }
245 mypipdev_class->dev_uevent = mypipe_uevent;
246
247 // Create a cdev in our pdata
248 cdev_init(&pdata.cdev, &mypipe_fops);
249 pdata.cdev.owner = THIS_MODULE;
250
251 // Register this newly created cdev to kernel. We only have one dev, so we
252 // can MKDEV(dev_major, 0).
253
254 // Also, we want to devices that one handles write and one read, so the
    last
255 // arg is not 1, but 2 (In this case, MYPIPE_MINOR_COUNT)
256 ret = cdev_add(&pdata.cdev, MKDEV(dev_major, 0), MYPIPE_MINOR_COUNT);
257 if (ret)
258     goto unregister;
259
260 // Create device node /dev/mypipe_in and /dev/mypipe_out
261 device_create(mypipdev_class, NULL, MKDEV(dev_major, MYPIPE_MINOR_IN),
262     NULL, "%s_in", MYPIPE_DEVICE_NAME);
263 device_create(mypipdev_class, NULL, MKDEV(dev_major, MYPIPE_MINOR_OUT),
264     NULL, "%s_out", MYPIPE_DEVICE_NAME);
265
266 pdata.is_active = false;
267
268 printk(KERN_INFO "%s: module loaded\n", MYPIPE_DEVICE_NAME);

```

```

269     return 0;
270
271 del_cdev:
272     cdev_del(&pdata.cdev);
273 unregister:
274     unregister_chrdev_region(MKDEV(dev_major, 0), MYPIPE_MINOR_COUNT);
275     return ret;
276 }
277
278 static void __exit mypipe_destroy(void) {
279     // This is basically the reverse operation of mypipe_init
280     device_destroy(mypipedev_class, MKDEV(dev_major, MYPIPE_MINOR_IN));
281     device_destroy(mypipedev_class, MKDEV(dev_major, MYPIPE_MINOR_OUT));
282     class_unregister(mypipedev_class);
283     class_destroy(mypipedev_class);
284     cdev_del(&pdata.cdev);
285     unregister_chrdev_region(MKDEV(dev_major, 0), MYPIPE_MINOR_COUNT);
286     printk(KERN_INFO "%s: module unloaded", MYPIPE_DEVICE_NAME);
287 }
288
289 module_init(mypipe_init);
290 module_exit(mypipe_destroy);
291 // END_LST_INIT

```

在module\_init中我们注册了回调函数结构体cdev\_init(&pdata.cdev, &mypipe\_fops);这其中的mypipe\_fops定义为:

```

210 // BEGIN_LST_FOP
211 const struct file_operations mypipe_fops = {.owner = THIS_MODULE,
212                                             .open = mypipe_open,
213                                             .read = mypipe_read,
214                                             .write = mypipe_write,
215                                             .release = mypipe_release};
216 // END_LST_FOP

```

open, release, read, write可以粗略地看作对应了用户态下的fopen, fclose, fread, fwrite四个操作。我们需要实现这四个回调函数才能正确地完成任务。对于open与release, 由于我们实现的是一个虚拟设备, 无需与物理硬件交互, 所以不需要进行特别的操作; 为了保证只有一个程序可以向管道写入, 同时保证在写入程序关闭管道时向读取端提示已经写入端已经关闭, 需要处理pdata.is\_active量来标识当前是否有程序正在占用管道的写入端。

对于read与write函数, 首先需要判断minor号是否匹配: 我们不允许在入口调用read, 也不允许在出口调用write。之后的操作是类似的, 以read为例。首先尝试获取缓冲区的锁, 如果当前的数据长度为 0, 那么要么是对面端已经关闭了管道而且我们已经读取了所有的数

据，要么是管道对面还有等着写入的数据暂时没有写入。对于前者，我们可以直接返回0代表已经读取完毕，对于后者，我们可以将自己放入等待区，等待pdata.data\_len > 0的事件发生时重启此系统调用。如果当前的数据长度不为0，那么我们就可以从环形缓存中依次读取数据放入用户提供的缓冲区中，并且提醒可能的正在等待的写入端我们已经腾出了空间。write的操作是完全类似的。

```
83 // BEGIN_LST_READ_WRITE
84 static ssize_t mypipe_read(struct file *filep, char __user *buf, size_t size,
85                             loff_t *offset) {
86     ssize_t ret = 0;
87
88     if (iminor(file_inode(filep)) != MYPIPE_MINOR_OUT) {
89         return -EINVAL;
90     }
91
92     // Acquire the lock. If not acquired and interrupted by signal, let kernel
93     // to try restart the syscall
94     if (mutex_lock_interruptible(&mypipe_lock)) {
95         return -ERESTARTSYS;
96     }
97
98     // We currently have no data, but we can wait for possible new to write in
99     while (pdata.data_len == 0) {
100         mutex_unlock(&mypipe_lock);
101
102         // Check if the writer is still active
103         if (mutex_lock_interruptible(&isactive_lock)) {
104             return -ERESTARTSYS;
105         }
106         bool writer_is_active = pdata.is_active;
107         mutex_unlock(&isactive_lock);
108
109         if (!writer_is_active) {
110             return 0; // No data left and the writer is gone -> EOF for reader
111         }
112
113         // There is still writer, wait for data
114         if (filep->f_flags & O_NONBLOCK) {
115             // The caller says no waiting, so we don't wait
116             return -EAGAIN;
117         }
118         if (wait_event_interruptible(read_queue, pdata.data_len > 0)) {
119             return -ERESTARTSYS;
120         }
121     }
122 }
```

```

120     }
121     // Woken, relock for checking the condition
122     if (mutex_lock_interruptible(&mypipe_lock)) {
123         return -ERESTARTSYS;
124     }
125 }
126
127 // data_len > 0
128 size_t to_read = min(size, pdata.data_len);
129 size_t until_end = min(to_read, MYPIPE_BUFFER_SIZE - pdata.read_pos);
130 if (copy_to_user(buf, pdata.buffer + pdata.read_pos, until_end)) {
131     ret = -EFAULT;
132     goto out;
133 }
134
135 if (to_read > until_end) {
136     if (copy_to_user(buf + until_end, pdata.buffer, to_read - until_end)) {
137         ret = -EFAULT;
138         goto out;
139     }
140 }
141 pdata.read_pos = (pdata.read_pos + to_read) % MYPIPE_BUFFER_SIZE;
142 pdata.data_len -= to_read;
143 ret = to_read;
144
145 wake_up_interruptible(&write_queue);
146
147 out:
148     mutex_unlock(&mypipe_lock);
149     return ret;
150 }
151
152 static ssize_t mypipe_write(struct file *file, const char __user *buf,
153                             size_t size, loff_t *offset) {
154     // This is very similar to read(), just write instead of read
155
156     ssize_t ret = 0;
157
158     if (iminor(file_inode(file)) != MYPIPE_MINOR_IN) {
159         return -EINVAL;
160     }
161
162     if (mutex_lock_interruptible(&mypipe_lock)) {

```

```

163     return -ERESTARTSYS;
164 }
165
166 while (pdata.data_len == MYPIPE_BUFFER_SIZE) {
167     mutex_unlock(&mypipe_lock);
168     if (filep->f_flags & O_NONBLOCK) {
169         return -EAGAIN;
170     }
171     if (wait_event_interruptible(write_queue,
172                                pdata.data_len < MYPIPE_BUFFER_SIZE)) {
173         return -ERESTARTSYS;
174     }
175     if (mutex_lock_interruptible(&mypipe_lock)) {
176         return -ERESTARTSYS;
177     }
178 }
179
180 size_t space_left = MYPIPE_BUFFER_SIZE - pdata.data_len;
181 size_t to_write = min(size, space_left);
182 size_t until_end = min(to_write, MYPIPE_BUFFER_SIZE - pdata.write_pos);
183
184 if (copy_from_user(pdata.buffer + pdata.write_pos, buf, until_end)) {
185     ret = -EFAULT;
186     goto out;
187 }
188
189 if (to_write > until_end) {
190     if (copy_from_user(pdata.buffer, buf + until_end,
191                       to_write - until_end)) {
192         ret = -EFAULT;
193         goto out;
194     }
195 }
196
197 pdata.write_pos = (pdata.write_pos + to_write) % MYPIPE_BUFFER_SIZE;
198 pdata.data_len += to_write;
199 ret = to_write;
200
201 wake_up_interruptible(&read_queue);
202
203 out:
204 mutex_unlock(&mypipe_lock);
205 return ret;

```

```
206 | }
207
208 // END_LST_READ_WRITE
```

## 4 程序运行情况

首先，编写简单的写入（Listings 2）与读出程序（Listings 3）进行测试：



```
~/repos/os/lab6 main ?1 23:55:50  ~/repos/os/lab6 main ?1 23:55:44
> ./build/pread  •> ./build/pwrite
Reader received: Hello from the other side!  ~/repos/os/lab6 main ?1 23:56:06
~/repos/os/lab6 main ?1 7s 23:56:06  > []
```

图 1: 简单读入读出测试

之后，尝试利用管道传输超出缓冲区容量的数据（Listings 4），读取大约620 KB：



```
1 Writer started
2 Reader started
3 Writer ended
4 Reader read content: GNU 'make'
5 1 Overview of 'make'
6 2 An Introduction to Makefiles
7 3 Writing Makefiles
8 4 Writing Rules
9 5 Writing Recipes in Rules
10 6 How to Use Variables
11 7 Conditional Parts of Makefiles
12 8 Functions for Transforming Text
13 9 How to Run 'make'
14 10 Using Implicit Rules
```

图 2: 程序输出的开头

```

13700                                     (line 6589)
13701 * WEAVE:                               Implicit Variables.
13702                                     (line 7823)
13703 * wildcard:                             Wildcard Function. (line 1715)
13704 * wildcard <1>:                         File Name Functions.
13705                                     (line 5983)
13706 * word:                                 Text Functions.   (line 5805)
13707 * wordlist:                             Text Functions.   (line 5814)
13708 * words:                                Text Functions.   (line 5826)
13709 * YACC:                                 Implicit Variables.
13710                                     (line 7805)
13711 * YFLAGS:                               Implicit Variables.
13712                                     (line 7882)
13713 Reader ended
13714

```

图 3: 程序输出的结尾

## 5 实验总结

在本次实验中，我第一次实现了一个简单的 Linux 驱动。在阅读了一些教程后，对代码结构有了基本的认识，实际写起来也并不会觉得过于困难，这次时间很大地提升了我的编程水平。

## A 完整源代码

Listing 1: 内核驱动模块代码

```

1 #include <linux/cdev.h>
2 #include <linux/fs.h>
3 #include <linux/init.h>
4 #include <linux/module.h>
5
6 // Ref:
7 // https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.
  html
8 // https://olegkutkov.me/2018/03/14/simple-linux-character-device-driver/
9
10 // BEGIN_LST_DEF
11 #define MYPIPE_BUFFER_SIZE 4096
12 #define MYPIPE_DEVICE_NAME "wendy"
13
14 static dev_t dev_major = 0;

```

```

15
16 enum MyPipeMinors {
17     MYPIPE_MINOR_IN = 0,
18     MYPIPE_MINOR_OUT,
19     MYPIPE_MINOR_COUNT,
20 };
21
22 struct mypipe_data {
23     struct cdev cdev;
24     // Whether opened by some writer
25     bool is_active;
26     u8 buffer[MYPIPE_BUFFER_SIZE];
27     size_t write_pos;
28     size_t read_pos;
29     size_t data_len;
30 };
31
32 // This could be an array if we want multiple cdev
33 static struct mypipe_data pdata;
34
35 static struct class *mypipedev_class = NULL;
36
37 // Protects the buffer
38 static DEFINE_MUTEX(mypipe_lock);
39 // Protects the is_active value
40 static DEFINE_MUTEX(isactive_lock);
41 // These are two wait_queue_head_t for storing tasks that are waiting for
42 // writing/ reading
43 static DECLARE_WAIT_QUEUE_HEAD(read_queue);
44 static DECLARE_WAIT_QUEUE_HEAD(write_queue);
45 // END_LST_DEF
46
47 // BEGIN_LST_OPEN_RELEASE
48 static int mypipe_open(struct inode *inode, struct file *file) {
49     int minor = iminor(inode);
50     printk(KERN_INFO "%s opened, minor = %d", MYPIPE_DEVICE_NAME, minor);
51
52     // If its opening the writing end, we limit it to only one writer
53     if (minor == MYPIPE_MINOR_IN) {
54         if (mutex_lock_interruptible(&isactive_lock)) {
55             return -ERESTARTSYS;
56         }
57

```

```

58     if (pdata.is_active) {
59         return -EBUSY;
60     }
61     pdata.is_active = true;
62     mutex_unlock(&isactive_lock);
63 }
64 return 0;
65 }
66
67 static int mypipe_release(struct inode *inode, struct file *file) {
68     int minor = iminor(inode);
69     printk(KERN_INFO "%s released, minor = %d", MYPIPE_DEVICE_NAME, minor);
70
71     if (minor == MYPIPE_MINOR_IN) {
72         mutex_lock(&isactive_lock);
73         pdata.is_active = false;
74         mutex_unlock(&isactive_lock);
75         wake_up_interruptible(&read_queue);
76     }
77     return 0;
78 }
79
80
81 // END_LST_OPEN_RELEASE
82
83 // BEGIN_LST_READ_WRITE
84 static ssize_t mypipe_read(struct file *filep, char __user *buf, size_t size,
85                          loff_t *offset) {
86     ssize_t ret = 0;
87
88     if (iminor(file_inode(filep)) != MYPIPE_MINOR_OUT) {
89         return -EINVAL;
90     }
91
92     // Acquire the lock. If not acquired and interrupted by signal, let kernel
93     // to try restart the syscall
94     if (mutex_lock_interruptible(&mypipe_lock)) {
95         return -ERESTARTSYS;
96     }
97
98     // We currently have no data, but we can wait for possible new to write in
99     while (pdata.data_len == 0) {
100         mutex_unlock(&mypipe_lock);

```

```

101
102     // Check if the writer is still active
103     if (mutex_lock_interruptible(&isactive_lock)) {
104         return -ERESTARTSYS;
105     }
106     bool writer_is_active = pdata.is_active;
107     mutex_unlock(&isactive_lock);
108
109     if (!writer_is_active) {
110         return 0; // No data left and the writer is gone -> EOF for reader
111     }
112
113     // There is still writer, wait for data
114     if (filep->f_flags & O_NONBLOCK) {
115         // The caller says no waiting, so we don't wait
116         return -EAGAIN;
117     }
118     if (wait_event_interruptible(read_queue, pdata.data_len > 0)) {
119         return -ERESTARTSYS;
120     }
121     // Woken, relock for checking the condition
122     if (mutex_lock_interruptible(&mypipe_lock)) {
123         return -ERESTARTSYS;
124     }
125 }
126
127 // data_len > 0
128 size_t to_read = min(size, pdata.data_len);
129 size_t until_end = min(to_read, MYPIPE_BUFFER_SIZE - pdata.read_pos);
130 if (copy_to_user(buf, pdata.buffer + pdata.read_pos, until_end)) {
131     ret = -EFAULT;
132     goto out;
133 }
134
135 if (to_read > until_end) {
136     if (copy_to_user(buf + until_end, pdata.buffer, to_read - until_end)) {
137         ret = -EFAULT;
138         goto out;
139     }
140 }
141 pdata.read_pos = (pdata.read_pos + to_read) % MYPIPE_BUFFER_SIZE;
142 pdata.data_len -= to_read;
143 ret = to_read;

```

```

144
145     wake_up_interruptible(&write_queue);
146
147 out:
148     mutex_unlock(&mypipe_lock);
149     return ret;
150 }
151
152 static ssize_t mypipe_write(struct file *filep, const char __user *buf,
153                             size_t size, loff_t *offset) {
154     // This is very similar to read(), just write instead of read
155
156     ssize_t ret = 0;
157
158     if (iminor(file_inode(filep)) != MYPIPE_MINOR_IN) {
159         return -EINVAL;
160     }
161
162     if (mutex_lock_interruptible(&mypipe_lock)) {
163         return -ERESTARTSYS;
164     }
165
166     while (pdata.data_len == MYPIPE_BUFFER_SIZE) {
167         mutex_unlock(&mypipe_lock);
168         if (filep->f_flags & O_NONBLOCK) {
169             return -EAGAIN;
170         }
171         if (wait_event_interruptible(write_queue,
172                                     pdata.data_len < MYPIPE_BUFFER_SIZE)) {
173             return -ERESTARTSYS;
174         }
175         if (mutex_lock_interruptible(&mypipe_lock)) {
176             return -ERESTARTSYS;
177         }
178     }
179
180     size_t space_left = MYPIPE_BUFFER_SIZE - pdata.data_len;
181     size_t to_write = min(size, space_left);
182     size_t until_end = min(to_write, MYPIPE_BUFFER_SIZE - pdata.write_pos);
183
184     if (copy_from_user(pdata.buffer + pdata.write_pos, buf, until_end)) {
185         ret = -EFAULT;
186         goto out;

```

```

187     }
188
189     if (to_write > until_end) {
190         if (copy_from_user(pdata.buffer, buf + until_end,
191                             to_write - until_end)) {
192             ret = -EFAULT;
193             goto out;
194         }
195     }
196
197     pdata.write_pos = (pdata.write_pos + to_write) % MYPIPE_BUFFER_SIZE;
198     pdata.data_len += to_write;
199     ret = to_write;
200
201     wake_up_interruptible(&read_queue);
202
203 out:
204     mutex_unlock(&mypipe_lock);
205     return ret;
206 }
207
208 // END_LST_READ_WRITE
209
210 // BEGIN_LST_FOP
211 const struct file_operations mypipe_fops = {.owner = THIS_MODULE,
212                                             .open = mypipe_open,
213                                             .read = mypipe_read,
214                                             .write = mypipe_write,
215                                             .release = mypipe_release};
216 // END_LST_FOP
217
218 // Used to set the permission to allow any user to r/w
219 static int mypipe_uevent(const struct device *dev,
220                          struct kobj_uevent_env *env) {
221     add_uevent_var(env, "DEVMODE=%#o", 0666);
222     return 0;
223 }
224
225 // BEGIN_LST_INIT
226 static int __init mypipe_init(void) {
227     int ret;
228     dev_t dev;
229

```

```

230 // Acquire a region of major, minor for us to use. The name appears in
231 // /proc/devices (and potentially /sys ?).
232 ret = alloc_chrdev_region(&dev, 0, MYPIPE_MINOR_COUNT, MYPIPE_DEVICE_NAME)
    ;
233 if (ret) {
234     return ret;
235 }
236
237 dev_major = MAJOR(dev);
238
239 // Create the class in sysfs. The name appears in /sys/class/<name>
240 mypipdev_class = class_create(MYPIPE_DEVICE_NAME);
241 if (IS_ERR(mypipdev_class)) {
242     ret = PTR_ERR(mypipdev_class);
243     goto del_cdev;
244 }
245 mypipdev_class->dev_uevent = mypipe_uevent;
246
247 // Create a cdev in our pdata
248 cdev_init(&pdata.cdev, &mypipe_fops);
249 pdata.cdev.owner = THIS_MODULE;
250
251 // Register this newly created cdev to kernel. We only have one dev, so we
252 // can MKDEV(dev_major, 0).
253
254 // Also, we want to devices that one handles write and one read, so the
    last
255 // arg is not 1, but 2 (In this case, MYPIPE_MINOR_COUNT)
256 ret = cdev_add(&pdata.cdev, MKDEV(dev_major, 0), MYPIPE_MINOR_COUNT);
257 if (ret)
258     goto unregister;
259
260 // Create device node /dev/mypipe_in and /dev/mypipe_out
261 device_create(mypipdev_class, NULL, MKDEV(dev_major, MYPIPE_MINOR_IN),
262     NULL, "%s_in", MYPIPE_DEVICE_NAME);
263 device_create(mypipdev_class, NULL, MKDEV(dev_major, MYPIPE_MINOR_OUT),
264     NULL, "%s_out", MYPIPE_DEVICE_NAME);
265
266 pdata.is_active = false;
267
268 printk(KERN_INFO "%s: module loaded\n", MYPIPE_DEVICE_NAME);
269 return 0;
270

```

```

271 del_cdev:
272     cdev_del(&pdata.cdev);
273 unregister:
274     unregister_chrdev_region(MKDEV(dev_major, 0), MYPIPE_MINOR_COUNT);
275     return ret;
276 }
277
278 static void __exit mypipe_destroy(void) {
279     // This is basically the reverse operation of mypipe_init
280     device_destroy(mypipedev_class, MKDEV(dev_major, MYPIPE_MINOR_IN));
281     device_destroy(mypipedev_class, MKDEV(dev_major, MYPIPE_MINOR_OUT));
282     class_unregister(mypipedev_class);
283     class_destroy(mypipedev_class);
284     cdev_del(&pdata.cdev);
285     unregister_chrdev_region(MKDEV(dev_major, 0), MYPIPE_MINOR_COUNT);
286     printk(KERN_INFO "%s: module unloaded", MYPIPE_DEVICE_NAME);
287 }
288
289 module_init(mypipe_init);
290 module_exit(mypipe_destroy);
291 // END_LST_INIT
292
293 MODULE_LICENSE("GPL");

```

Listing 2: 简单写入程序代码

```

1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h>
5
6  int main() {
7      int fd = open("/dev/wendy_in", O_WRONLY);
8      if (fd < 0) {
9          printf("Error open pipe");
10         return 1;
11     }
12
13     const char *msg = "Hello from the other side!\n";
14     write(fd, msg, strlen(msg));
15
16     close(fd);
17     return 0;
18 }

```

Listing 3: 简单读出程序代码

```

1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int main() {
6      int fd = open("/dev/wendy_out", O_RDONLY);
7      if (fd < 0) {
8          printf("Error open pipe");
9          return 1;
10     }
11
12     char buf[128];
13     ssize_t n = read(fd, buf, sizeof(buf) - 1);
14     if (n < 0) {
15         printf("Error read pipe");
16         return 1;
17     }
18
19     buf[n] = '\0';
20     printf("Reader received: %s", buf);
21
22     close(fd);
23     return 0;
24 }

```

Listing 4: 测试程序代码

```

1  #include <iostream>
2  #include <thread>
3  #include <fstream>
4  #include <chrono>
5  #include <syncstream>
6
7  void writer() {
8      std::osyncstream(std::cout) << "Writer started" << std::endl;
9
10     std::ofstream pipe_write("/dev/wendy_in");
11     if (!pipe_write.is_open()) {
12         std::cerr << "Cannot open pipe in" << std::endl;
13     }
14
15     std::ifstream txt("./make.txt");
16     if (!txt.is_open()) {

```

```

17     std::cerr << "Canno open make.txt" << std::endl;
18 }
19 // pipe_write << "Hello from the other side\n";
20 pipe_write << txt.rdbuf();
21 std::osyncstream(std::cout) << "Writer ended" << std::endl;
22 }
23
24 void reader() {
25     std::osyncstream(std::cout) << "Reader started" << std::endl;
26     std::ifstream pipe_read("/dev/wendy_out");
27     std::osyncstream(std::cout) << "Reader read content: " << pipe_read.rdbuf
        ();
28     std::osyncstream(std::cout) << "Reader ended" << std::endl;
29 }
30
31 int main() {
32     std::thread writer_thread(writer);
33     std::this_thread::sleep_for(std::chrono::milliseconds(100));
34     std::thread reader_thread(reader);
35     if (reader_thread.joinable()) {
36         reader_thread.join();
37     }
38     if (writer_thread.joinable()) {
39         writer_thread.join();
40     }
41     return 0;
42 }

```