操作系统课程试验
驱动程序问题——管道驱动程序开发

BobAnkh

June 2021

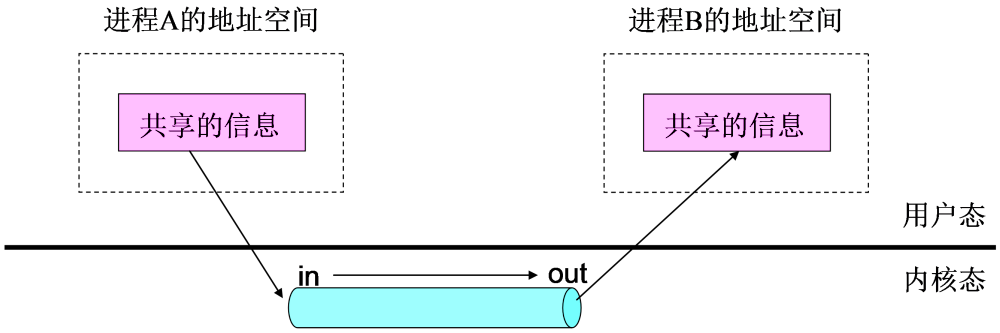
编程平台: Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-142-generic x86_64)
编程语言: C

一. 实验题目

1. 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。

二. 设计思路

考虑到 Linux 平台设备即文件的特性，所以本次实验中选择在 Linux 平台上进行设备驱动程序的编写，采用 C 语言编写实现，同时为了方便驱动程序的编译以及管道设备的挂载，编写了 `Makefile` 用于编译挂载。因为需要设计的是管道设备，所以我选择了编写字符设备。主要就是实现字符设备所需要实现的一些接口函数，初始化的时候注册设备并分配内核缓冲区，关闭的时候注销设备并释放内核缓冲区，采用互斥信号量的形式来实现读与写之间的互斥。具体的设计和细节见下节 §3中分析。

三. 程序结构

驱动程序主体在 `mypipe.c` 中进行了实现，主要是针对 Linux 提供的设备文件操作方法表 `file_operations` 的接口实现了必要和需要的几个函数，包括 `mypipe_init`, `mypipe_exit`, `mypipe_read`, `mypipe_write`, `mypipe_open`, `mypipe_release`。为了方便，对于设备号我采用了静态分配的方式，在宏定义中直接指定了主设备号。设备文件操作方法表 `file_operations` 是 Linux 中 `fs.h` 中定义的结构体，系统调用找到设备驱动程序时，正是通过这个结构体访问驱动程序的各个函数，我在本程序中对其定义如下：

```
// 设备文件操作方法表
static struct file_operations mypipe_flops = {
    owner : THIS_MODULE,
    open : mypipe_open,
    release : mypipe_release,
    write : mypipe_write,
    read : mypipe_read
};
```

在定义的 6 个函数中，`mypipe_init` 函数是用于模块初始化，`mypipe_exit` 是用于模块退出处理，`mypipe_read` 用于管道输出端从管道中读取数据，`mypipe_write` 用于管道输入端向管道中写入数据，`mypipe_open` 用于希望使用设备时打开设备，`mypipe_release` 用于结束时释放设备资源 (解除占用)。后两者 (`mypipe_open`, `mypipe_release`) 主要是驱动的必要接口，但是因为不需要做额外操作，所以仅做了空实现，下文不再介绍，仅介绍前四个主要的函数。

1. `mypipe_init` 模块初始化

该函数定义如下：

```
// 初始化函数
static int __init mypipe_init(void)
{
    // 注册字符设备
    int ret = register_chrdev(MYPIPE_MAJOR, DEVICE_NAME, &mypipe_flops);
    if (ret < 0)
    {
        printk(KERN_EMERG DEVICE_NAME " can't register major number. Please
            define another MYPIPE_MAJOR number.\n");
        goto fail;
```

```
    }
    printk(KERN_EMERG DEVICE_NAME " initialized successfully.\n");
    // 分配内核缓冲区
    buffer = kmalloc(BUF_SIZE, GFP_KERNEL);
    if (!buffer)
    {
        ret = -ENOMEM;
        goto unregister;
    }
    memset(buffer, 0, BUF_SIZE);
    // 初始化互斥信号量为1
    sema_init(&mutex, 1);

    return 0;
unregister:
    unregister_chrdev(MYPIPE_MAJOR, DEVICE_NAME);
fail:
    return ret;
}
```

可以看到，这个函数的函数名前加了初始化宏 `__init`，标记这是初始化函数，需要用下面这行代码，在驱动被加载到内核的时候，让初始化函数执行：

```
module_init(mypipe_init);
```

这一段代码主要实现了以下 3 个功能：

(1) 注册设备

使用 `register_chrdev` 函数使用已经静态分配了的设备号来注册字符型设备，如果该设备号已经被分配则需要去宏定义中修改设备号来重新注册设备。

(2) 在内核空间申请内存

因为管道需要在内核空间存储一定量的数据，所以需要使用 `kmalloc` 函数在内核空间申请一定量的空间并且用 `memset` 函数进行初始化。申请的空间大小也定义在宏中。

(3) 初始化互斥信号量

因为内核空间的内存对于输入端和输出端两个设备实例而言属于共享资源，所以为了保证内容和顺序的正确性，需要用信号量对于这块缓冲区实现互斥访问。所以定义了信号量 `struct semaphore mutex` 并在此初始化为 1。

2. `mypipe_exit` 模块退出处理

该函数定义如下：

```
static void __exit mypipe_exit(void)
{
    // 注销字符设备
    unregister_chrdev(MYPIPE_MAJOR, DEVICE_NAME);
    down(&mutex);
    // 释放内存
    if (buffer)
```

```
        kfree(buffer);
    printk(KERN_EMERG DEVICE_NAME " removed successfully.\n");
    up(&mutex);
}
```

可以看到，这个函数的函数名前加了退出宏 `__exit`，标记这是退出函数，需要用下面这行代码，在驱动被从内核中卸载的时候，让退出函数执行：

```
module_exit(mypipe_exit);
```

该函数的主要功能就是注销字符设备和释放内存。

3. `mypipe_read` 输出端读取数据

该函数定义如下：

```
// 将内核中的缓冲区buffer中的数据输出到用户空间中
static ssize_t mypipe_read(struct file *kfile, char __user *buf, size_t count
    , loff_t *fpos)
{
    size_t len;
    size_t read_len = count;
    ssize_t ret = 0;
    if (down_interruptible(&mutex))
    {
        return -ERESTARTSYS;
    }

    if (pRead <= pWrite)
    {
        len = pWrite - pRead;
        if (len < count)
        {
            read_len = len;
        }
        if (copy_to_user(buf, buffer + pRead, read_len))
        {
            printk(KERN_EMERG DEVICE_NAME " Fail to read\n");
            ret = -EFAULT;
            goto out;
        }
        pRead = (pRead + read_len) % BUF_SIZE;
    }
    else
    {
        len = BUF_SIZE - pRead + pWrite;
        if (len < count)
        {
            read_len = len;
        }
        if (read_len <= (BUF_SIZE - pRead))
```

```
        {
            if (copy_to_user(buf, buffer + pRead, read_len))
            {
                printk(KERN_EMERG DEVICE_NAME " Fail to read\n");
                ret = -EFAULT;
                goto out;
            }
            pRead = (pRead + read_len) % BUF_SIZE;
        }
        else
        {
            if (copy_to_user(buf, buffer + pRead, BUF_SIZE - pRead))
            {
                printk(KERN_EMERG DEVICE_NAME " Fail to read\n");
                ret = -EFAULT;
                goto out;
            }
            if (copy_to_user(buf + BUF_SIZE - pRead, buffer, read_len - (
                BUF_SIZE - pRead)))
            {
                printk(KERN_EMERG DEVICE_NAME " Fail to read\n");
                ret = -EFAULT;
                goto out;
            }
            pRead = (pRead + read_len) % BUF_SIZE;
        }
    }
    printk(KERN_EMERG DEVICE_NAME " Read, pRead:%zu, pWrite:%zu\n", pRead,
        pWrite);
    ret = read_len;
out:
    up(&mutex);
    return ret;
}
```

在读取和写入上，因为只在内核开了一块内存缓冲区，为了充分利用空间，所以在这块缓冲区上进行循环的读和写，即如果写或者读到了一端，会从另一端接着。为了实现这一点，我选择使用 pRead 和 pWrite 分别来指示读到哪和写到哪了，两者的初值均设置为 0，表示初始无数据写入且无数据可读。

对于读的这一部分，如果指示写到哪里的 pWrite 在指示写到哪里的 pRead 右侧，如图 1所示，则它们的差值为可读的存储空间长度 (即黄色部分的空间)，读取是可以一直读空已存数据直到 pRead=pWrite 为止。此时采用 copy_to_user 函数将数据从内核空间读取到用户空间。如果指示写到哪里的 pWrite 在指示读到哪里的 pRead 左侧，如图 2所示，则存储空间长度减去它们的差值才是可读的存储空间长度，其余类似，只不过在某些情况下将分段使用 copy_to_user 函数将两段数据从内核空间读取到用户空间。

此外，由函数参数中的 count 指示希望读取的长度，如果该长度超过可读空间长度则以可读空间长度为准。并且，涉及存储空间的操作之前，需要先获取互斥信号量，结束后再释放。在读取之后根据实际读取长度移动 pRead，如果其大于等于存储空间长度 BUF_SIZE，则保存其

对 BUF_SIZE 求模之后的结果。

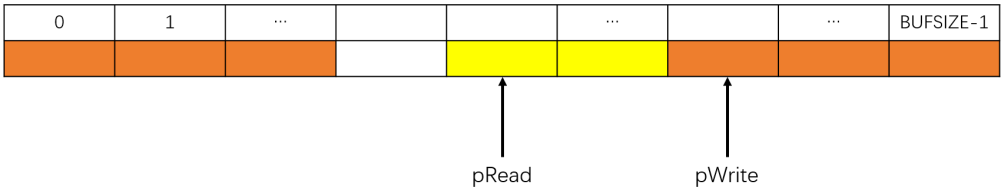


图 1: pRead ≤ pWrite 时，黄色表示已经存储了数据的空间，橙色表示可写的空间

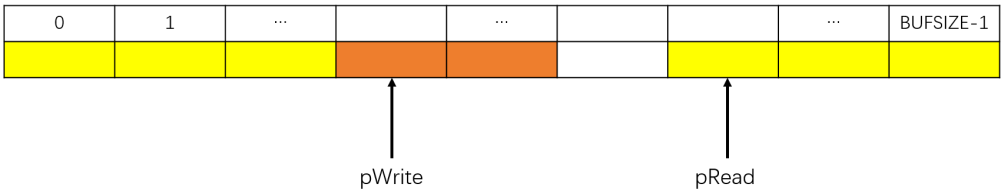


图 2: pRead > pWrite 时，黄色表示已经存储了数据的空间，橙色表示可写的空间

4. mypipe_write 输入端写入数据

该函数定义如下：

```
// 将用户程序写入内核的数据写入内核中的缓冲区buffer
static ssize_t mypipe_write(struct file *kfile, const char __user *buf,
    size_t count, loff_t *fpos)
{
    size_t len;
    size_t write_len = count;
    ssize_t ret = 0;
    if (down_interruptible(&mutex))
    {
        return -ERESTARTSYS;
    }

    if (pRead <= pWrite)
    {
        len = BUF_SIZE - pWrite + pRead;
        if (write_len >= len)
        {
            printk(KERN_EMERG DEVICE_NAME " Fail to write, not enough buffer,
                pRead:%zu, pWrite:%zu\n", pRead, pWrite);
            goto out;
        }
        if (write_len <= (BUF_SIZE - pWrite))
        {
            if (copy_from_user(buffer + pWrite, buf, write_len))
            {
                printk(KERN_EMERG DEVICE_NAME " Fail to write\n");
                ret = -EFAULT;
```

```
                goto out;
            }
            pWrite = (pWrite + write_len) % BUF_SIZE;
        }
        else
        {
            if (copy_from_user(buffer + pWrite, buf, BUF_SIZE - pWrite))
            {
                printk(KERN_EMERG DEVICE_NAME " Fail to write\n");
                ret = -EFAULT;
                goto out;
            }
            if (copy_from_user(buffer, buf + BUF_SIZE - pWrite, write_len - (
                BUF_SIZE - pWrite)))
            {
                printk(KERN_EMERG DEVICE_NAME " Fail to write\n");
                ret = -EFAULT;
                goto out;
            }
            pWrite = (pWrite + write_len) % BUF_SIZE;
        }
    }
    else
    {
        len = pRead - pWrite;
        if (write_len >= len)
        {
            printk(KERN_EMERG DEVICE_NAME " Fail to write, not enough buffer,
                 pRead:%zu, pWrite:%zu\n", pRead, pWrite);
            goto out;
        }
        if (copy_from_user(buffer + pWrite, buf, write_len))
        {
            printk(KERN_EMERG DEVICE_NAME " Fail to write\n");
            ret = -EFAULT;
            goto out;
        }
        pWrite = (pWrite + write_len) % BUF_SIZE;
    }
    printk(KERN_EMERG DEVICE_NAME " Write, pRead:%zu, pWrite:%zu\n", pRead,
        pWrite);
    ret = write_len;

out:
    up(&mutex);
    return ret;
}
```

对于写的这一部分，如果指示写到哪里的 `pWrite` 在指示写到哪里的 `pRead` 左侧，如

图 2所示，则它们的差值减 1 为可写的存储空间长度 (即橙色部分的空间)。这样规定是为了保证 pWrite 不会追上 pRead 从而导致判断管道内为空，使数据遗失，即写入是可以一直写到 pWrite=pRead-1 为止。此时采用 copy_from_user 函数将数据从用户空间写入到内核空间。如果指示写到哪里的 pWrite 在指示读到哪里的 pRead 右侧，如图 1所示，则存储空间长度减去它们的差值再减 1 才是可写的存储空间长度，其余类似，只不过在某些情况下将分段使用 copy_from_user 函数将两段数据从用户空间写入到内核空间。

此外，由函数参数中的 count 指示希望读取的长度，如果该长度超过可写空间长度则直接拒绝写入并向上汇报错误。并且，涉及存储空间的操作之前，需要先获取互斥信号量，结束后再释放。在写入之后根据实际写入长度移动 pWrite，如果其大于等于存储空间长度 BUF_SIZE，则保存其对 BUF_SIZE 求模之后的结果。

四. 运行结果

本实验中，我在 Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-142-generic x86_64) 下对程序进行了编译测试运行。为了方便实验，内核缓存空间大小设置为 128 字节。

为了方便使用，编写了 Makefile，用于将一些编译和挂载/移出设备的指令放在一起。

通过如下操作进行编译安装：

1. 首先使用 make 指令将编写的 mypipe.c 文件在内核路径下编译成内核可用的 mypipe.ko 文件。显示如下：

```
❯ make
make -C /lib/modules/4.15.0-142-generic/build M=/home/shenyx/thuee-os/Project5-Driver modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-142-generic'
  CC [M]  /home/shenyx/thuee-os/Project5-Driver/mypipe.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/shenyx/thuee-os/Project5-Driver/mypipe.mod.o
  LD [M]  /home/shenyx/thuee-os/Project5-Driver/mypipe.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-142-generic'
```

2. 使用 make install 指令进行该设备的安装，这将会把该模块插入到内核中，同时挂载对应的设备至/dev/mypipe，并且使用 dmesg 打印内核信息用于检查时候正确插入和挂载。显示如下：

```
❯ make install
sudo insmod mypipe.ko
[sudo] password for shenyx:
dmesg | tail -n 1
[4615415.480796] MyPipe initialized successfully.
sudo mknod /dev/mypipe c 230 0
```

3. 编写读测试程序 readtest.c，主要部分的代码如下：

```
int main(void)
{
    int read_size = 0;
    ssize_t ret;
    char *buffer;
    int fd = open("/dev/mypipe", O_RDONLY | O_NONBLOCK);
    if (fd < 0)
    {
        perror("[ERROR] Fail to open pipe for reading data.\n");
```

```
        exit(1);
    }
    printf("Input a interger of how much bytes you want to read from the
        pipe> ");
    scanf("%d", &read_size);
    buffer = (char *)malloc(read_size * sizeof(char));
    ret = read(fd, buffer, read_size);
    if (ret > 0)
    {
        printf("Read %zd bytes from pipe, result is:\n%s\n", ret, buffer
            );
    }
    else
    {
        printf("Fail to read. Use command `dmesg` to debug.\n");
    }
    close(fd);
    free(buffer);
    return 0;
}
```

4. 编写写测试程序 `writetest.c`，主要部分的代码如下：

```
static int getLine(char *prmpt, char *buff, size_t sz)
{
    int ch, extra;
    if (prmpt != NULL)
    {
        printf("%s", prmpt);
        fflush(stdout);
    }
    if (fgets(buff, sz, stdin) == NULL)
        return NO_INPUT;

    if (buff[strlen(buff) - 1] != '\n')
    {
        extra = 0;
        while (((ch = getchar()) != '\n') && (ch != EOF))
            extra = 1;
        return (extra == 1) ? SIZE : OK;
    }

    buff[strlen(buff) - 1] = '\0';
    return OK;
}

int main(void)
{
    int ret;
    ssize_t res;
```

```
    char buffer[SIZE];

    int fd = open("/dev/mypipe", O_WRONLY | O_NONBLOCK);
    if (fd < 0)
    {
        perror("[ERROR] Fail to open pipe for writing data.\n");
        exit(1);
    }
    printf("Enter a string(should be less than limit, %d) you want to
        write to pipe", SIZE);
    ret = getLine("> ", buffer, sizeof(buffer));
    if (ret == NO_INPUT)
    {
        printf("\nNo input\n");
        return 1;
    }
    if (ret == SIZE)
    {
        printf("Input too long, exceeding the limit %d\n", SIZE);
        return 1;
    }
    res = write(fd, &buffer, strlen(buffer));
    if (res > 0)
    {
        printf("Write %zd bytes to pipe\n", res);
    }
    else
    {
        printf("Fail to write. Use command 'dmesg' to debug.\n");
    }
    close(fd);
    return 0;
}
```

5. 使用 `make test` 指令编译相关的读和写测试程序，显示如下：

```
❯ make test
gcc readtest.c -o readtest.exe
gcc writetest.c -o writetest.exe
```

接下来使用编译好的 `readtest.exe` 和 `writetest.exe` 进行一些管道读写操作。同时将使用 `dmesg` 命令查看设备驱动的输出的内核调试信息，结果如图 3所示

1. 先向管道中写入了 60 字节的数据如下：

```
❯ sudo ./writetest.exe
Enter a string(should be less than limit, 128) you want to write to pipe> 123456789012345678901234567890123456789012345678901234567890
Write 60 bytes to pipe
```

2. 再从管道中读取 35 字节的数据如下：

```
❯ sudo ./readtest.exe
Input a interger of how much bytes you want to read from the pipe> 35
Read 35 bytes from pipe, result is:
12345678901234567890123456789012345
```

3. 再向管道中写入了 72 字节的数据如下 (此时写入指示会从一端循环到另一端，可以在图 3中看到对应变化)：

```
❯ sudo ./writetest.exe
Enter a string(should be less than limit, 128) you want to write to pipe> abcdefghijklmnopqrstuvwxyz1234567890abcdefghijklmnopqrstuvwxyz1234567890
Write 72 bytes to pipe
```

4. 再从管道中读取 80 字节的数据如下：

```
❯ sudo ./readtest.exe
Input a interger of how much bytes you want to read from the pipe> 80
Read 80 bytes from pipe, result is:
67890123456789012345678 90abcdefghijklmnopqrstuvwxyz1234567890abcdefghijklmnopqrs
```

5. 这时候管道中仅剩下 17 字节的数据了，这个时候如果我们再读 25 字节的数据，将只能够读出 17 字节的数据如下所示：

```
❯ sudo ./readtest.exe
Input a interger of how much bytes you want to read from the pipe> 25
Read 17 bytes from pipe, result is:
tuvwxyz1234567890
```

由上述测试实验，可以看到非常良好的完成了管道通信的任务，实现了相应功能。

```
❯ dmesg | tail -n 17
[4615415.480796] MyPipe initialized successfully.
[4615873.084571]  Open pipe.
[4615901.248033] MyPipe Write, pRead:0, pWrite:60
[4615901.248062]  Release pipe.
[4615980.473137]  Open pipe.
[4615984.092380] MyPipe Read, pRead:35, pWrite:60
[4615984.092408]  Release pipe.
[4616016.994111]  Open pipe.
[4616077.697103] MyPipe Write, pRead:35, pWrite:4
[4616077.697131]  Release pipe.
[4616190.879771]  Open pipe.
[4616193.138079] MyPipe Read, pRead:115, pWrite:4
[4616193.138113]  Release pipe.
[4616280.139195]  Open pipe.
[4616282.753795] MyPipe Read, pRead:4, pWrite:4
[4616282.753838]  Release pipe.
[4616462.079252] MyPipe removed successfully.
```

图 3: dmesg 命令输出的设备驱动内核调试信息

使用完毕后可以使用 `make uninstall` 卸载设备并将模块从内核中移除，显示如下：

```
❯ make uninstall
sudo rmmod mypipe
[sudo] password for shenyx:
dmesg | tail -n 1
[4616462.079252] MyPipe removed successfully.
sudo rm -f /dev/mypipe
```

五. 实验感受和总结收获

通过本次实验，我复习了设备驱动程序的相关知识，并且更加清楚地了解了设备驱动程序需要实现哪些功能和接口，更加明晰了用户进程触发系统调用从而切换到内核态的相关机制，也是第一次从零开始学习了如何在 Linux 系统上编译一个驱动设备，同时也学习了包括 Makefile 书写、内核模块插入在内等一系列新的知识，可以说让我获益匪浅。这一些经验和经历，不光让我对于操作系统的相关知识点和概念有了更加深刻具体的理解，更是对于我日后的科研与开发有着很大的帮助。

六. 附录

随报告附设备驱动程序 `mypipe.c`，读测试程序 `readtest.c` 和写测试程序 `write.c`，以及用来进行编译的 Makefile。