实验 5：驱动程序问题

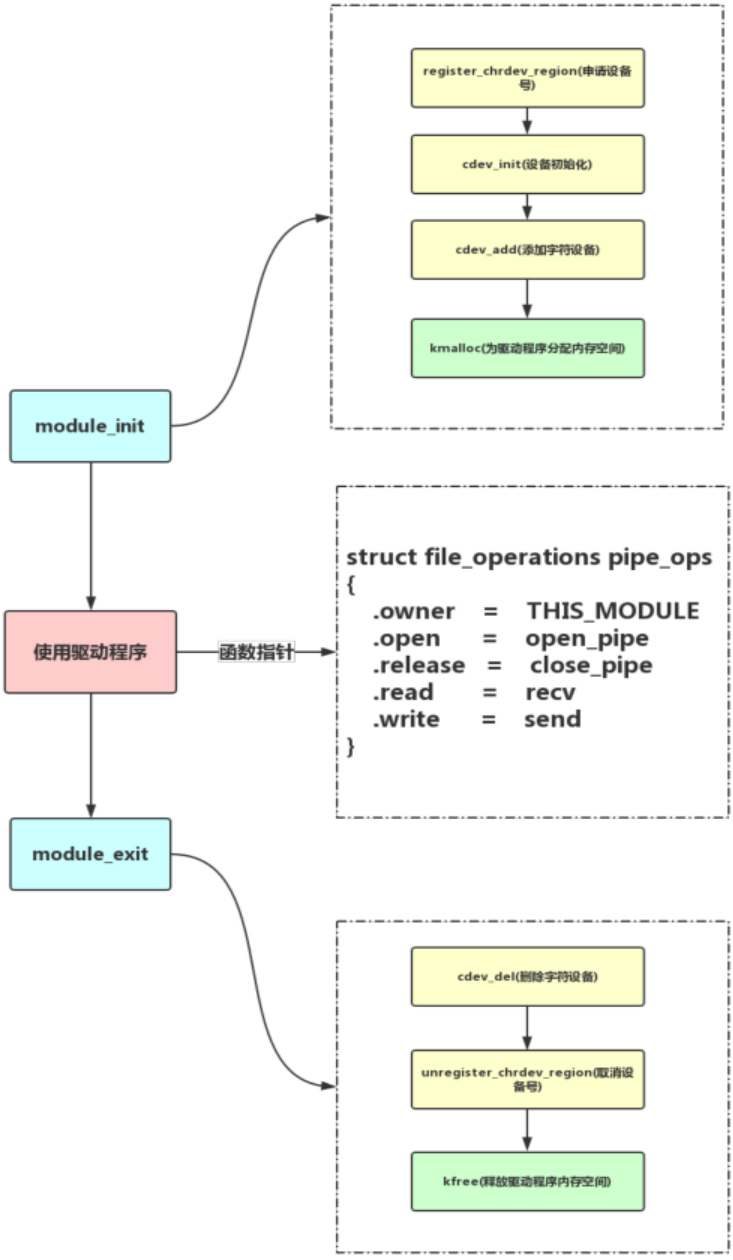
0. 实验环境

本次实验选择 **管道驱动程序开发**，操作系统为 **Linux**(Ubuntu 16.04)，编程语言为 C 语言。

文件列表：a) MyPipe.c, reader.c, writer.c: 分别为驱动程序，用于读出数据和写入数据程序的源代码； b)Makefile: 用于文件编译； c) 程序流程图.pdf：程序流程图图片

1. 设计思路与程序结构

程序流程图如下：



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。因此驱动程序中，创建两个设备实例。其中 0 号设备（次设备号为 0）用于读取数据，1 号设备（次设备号为 1）用于写入数据。

（a）驱动模块初始化

该驱动程序管理一个管道，驱动程序模块插入内核并初始化的过程中，会添加两个设备的设备节点。如流程图所示，分别进行 register_chrdev_region，cdev_init，cdev_add。本程序中令主设备号为 185，在使用本程序时请视情况更改代码中的 int MAJOR_NUMBER = 185;语句，谢谢~

同时，会为每个设备再分配各自的结构体（struct MyPipe），定义如下：

```
//pipe device struct
typedef struct MYPIPE {
    char* buffer;
    pipe_auth auth;
    dev_t dev_number;
    struct cdev cdev;
} Mypipe;
```

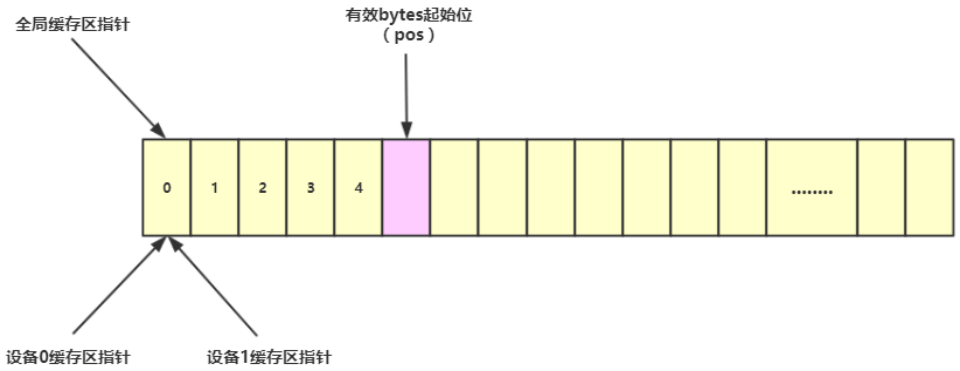
该结构体用于管理每个设备的信息，包括设备缓存区指针，设备读写权限，设备号与字符设备结构体。

之后为该管道分配内存空间(kmalloc)，同时初始化用于管理整个驱动程序的相应的数据结构（struct GLOBAL_PIPE）。struct GLOBAL_PIPE 定义如下：

```
//global pipe struct(for storing pipe info)
typedef struct GLOBAL_PIPE {
    char* buffer;
    size_t buf_size;      //buffer size
    size_t len;           //used size
    size_t pos;           //current reading position
    struct mutex lock;  //lock for read/write
} Gpipe;
```

该结构体用于管理整个管道的信息。如缓存区指针、大小，已写入 bytes 的数量，用于互斥的 mutex 量等。

值得注意的是，每个设备的缓存区指针都指向同一块内存空间，即 struct GLOBAL_PIPE 里的缓存区。但不同设备对该缓存区的操作有不同权限。图示如下：



当创建了两个对应的字符设备文件(如 sudo mknod in c 185 0)后，通过该文件可以对管道进行读或写。当打开文件(调用 open_pipe 函数)时，驱动程序会根据文件 inode 节点的信息(**inode->i_rdev**)判断次设备号，从而赋予字符设备文件对应的读写权限。读写权限枚举类型定义如下：

```
typedef enum PIPE_AUTHORITY {
    READONLY    = 0,
    WRITEONLY   = 1,
    READWRITE   = 2,
} pipe_auth;
```

在对文件进行读写(调用 recv 和 send 函数)时，会对设备的权限进行检查。如果该字符设备文件为只读权限（READONLY），则无法对管道进行写入，并返回 EPERM 错误码。

（b）使用驱动程序

当使用驱动程序时，驱动程序调用对应的处理函数。函数指针结构体定义如下：

```
static struct file_operations pipe_fops = {
    .owner          =    THIS_MODULE,
    .open           =    open_pipe,
    .release        =   close_pipe,
    .write          =   send,
    .read           =   recv,
};
```

本实验中，驱动程序实现了对管道的打开(open_pipe)，关闭(close_pipe)，写入数据(send)，接收数据(recv)这 4 个功能。

调用 open_pipe 时，会利用 struct file *filp 参数，将设备的地址存入 filp->private_data 中。当我们调用 send 或 recv 时，就可以通过 filp->private_data 获取到设备信息（如设备的读写权限）。调用 release，则会将 filp->private_data 置为 NULL。

当 send 或 recv 过程中，缓存区可以视为一个**循环队列**，每次写入或读出 bytes 时，会对当前位置进行模操作（如 wrt_pos = wrt_pos % buf_size）。

虽然可以指定一次写入多个 bytes，但程序设置了一个循环，每次只读或写 1 个 byte。读或写每个 byte 之前，会尝试获取 mutex，从而保证读和写的互斥。这种处理方式可以保证读和写几乎同时进行。具体代码如下：

```
//send data
static ssize_t send(struct file *filp, const char *buf, size_t count, loff_t *pos) {
	size_t i = 0;
	size_t wrt_pos = 0;
	Mypipe *dev = filp->private_data;

	printk("Mypipe: try to send\n");

	if(dev == NULL) {
		printk("Mypipe: device hasn't been open yet\n");
		return -1;
	}
	if(dev->auth != WRITEONLY) return -EPERM;

	while (i < count) {
		//buffer is full, return
		if(gpipe.len >= gpipe.buf_size) return -EAGAIN;

		//lock critical region
		mutex_lock(&(gpipe.lock));

		//round-robin char queue
		wrt_pos = (gpipe.pos + gpipe.len) % gpipe.buf_size;
		copy_from_user((dev->buffer + wrt_pos), (buf + i), 1);
		i += 1;
		gpipe.len += 1;

		mutex_unlock(&(gpipe.lock));
	}

	printk("Mypipe: send complete\n");

	return count;
}
```

```
//recieve data
static ssize_t recv(struct file *filp, char *buf, size_t count, loff_t *pos) {
	size_t i = 0;
	Mypipe *dev = filp->private_data;

	printk("Mypipe: try to recv\n");

	if(dev == NULL) {
		printk("Mypipe: device hasn't been open yet\n");
		return -1;
	}
	if(dev->auth != READONLY) return -EPERM;

	while (i < count) {
		//buffer is empty, return all bytes
		if(gpipe.len == 0) {
			printk("Mypipe: recv partially complete\n");
			return i;
		}

		//lock critical region
		mutex_lock(&(gpipe.lock));

		//round-robin char queue
		copy_to_user((buf + i), (dev->buffer + gpipe.pos), 1);
		gpipe.pos = (gpipe.pos + 1) % gpipe.buf_size;
		i += 1;
		gpipe.len -= 1;

		mutex_unlock(&(gpipe.lock));
	}

	printk("Mypipe: recv complete\n");

	return count;
}
```

（c）注销驱动模块：

首先删除两个设备实例，然后将设备号注销，最后将驱动中的内存空间释放，销毁 mutex 变量。

2. 程序运行状况

（a）安装驱动程序与设备：

Shell 命令如下：

```
make
sudo insmod MyPipe.ko
cd /dev
sudo mkdir MyPipe
cd MyPipe
sudo mknod in c 185 0
sudo mknod out c 185 1
```

可以通过 dmesg 命令 查看驱动与设备加载情况，结果如下：

```
[  865.244016] Mypipe: alloc device number complete, major number: 185
[  865.244017] Mypipe: register pipe device 0 complete, minor number: 0
[  865.244018] Mypipe: register pipe device 1 complete, minor number: 1
[  865.244018] Mypipe: allocate memory for global pipe buffer complete
[  865.244019] Mypipe: Inserting pipe module complete
```

可以看到：分配设备号，注册两个设备，分配驱动内存空间均成功。

（b）在 shell 中测试驱动程序:

首先我在 shell 将 shell 命令结果重定向到设备中，从而实现管道的写入。并在 shell 中读取管道中被写入的内容。具体 shell 命令如下：

```
cd /dev/MyPipe
sudo su
echo Hello_World > out
cat in
```

结果如下：

```
blake@blake-GP62-6QG:/dev/MyPipe$ cd /dev/MyPipe
blake@blake-GP62-6QG:/dev/MyPipe$ sudo su
root@blake-GP62-6QG:/dev/MyPipe# echo Hello_World > out
root@blake-GP62-6QG:/dev/MyPipe# cat in
Hello_World
root@blake-GP62-6QG:/dev/MyPipe#
```

可以看到：cat 命令成功读出了写入的字符串。

(c) 编写两个读写程序测试驱动程序：

```c
#include <stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>

int main() {
    int fd;
    char s[20];
    fd = open("/dev/MyPipe/in", O_RDONLY, S_IREAD);

    if(fd != -1) {
        read(fd, &s, 12);
        printf("%s\n", s);
        return 0;
    }
    else {
        printf("Can't open device file\n");
        return -1;
    }
}
```

```c
#include <stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>

int main() {
    int fd;
    char s[] = "Hello World!";
    fd = open("/dev/MyPipe/out", O_WRONLY, S_IWRITE);

    if(fd != -1) {
        write(fd, &s, 12);
        return 0;
    }
    else {
        printf("Can't open device file\n");
        return -1;
    }
}
```

利用 gcc 编译两个程序：

gcc reader.c –o reader

gcc writer.c –o writer

在 root 下，运行可执行程序 reader, writer，结果如下：

```
root@blake-GP62-6QG:/home/blake/application/大三上/操作系统/大作业# ./writer
root@blake-GP62-6QG:/home/blake/application/大三上/操作系统/大作业# ./reader
Hello World!
root@blake-GP62-6QG:/home/blake/application/大三上/操作系统/大作业#
```

执行正确。因此驱动程序正确的完成了管道的任务。

3. 体会和遇到的问题：

1.此次实验，让我充分了解了 linux 系统下，字符设备与驱动的安装与工作过程。在将驱动模块插入内核时，设备首先需要可用的设备号，对于字符设备而言，需要由主设备号和次设备号共同标识。然后初始化与添加设备实例。成功后，需要再创建字符设备文件，通过字符设备文件就可以对驱动程序进行调用了。

同时，对 linux 对驱动程序要求的统一的接口也有了一定的了解。通过 file_operations

结构体，我们可以定义一系列所需要的操作函数。而对于调用这些函数的操作系统和用户而言，接口是统一的。

因此，这次实验让我对驱动程序的本质有了更深刻的了解。

2.在实验的一开始，我对设备与驱动的关系的理解有一定偏差，导致驱动程序无法工作：我最初以为驱动程序插入内核时，无法指定设备数量。可以通过 mknod 等方式加载任意数量的设备，因此在注册设备号时使用 register_chrdev，相当于只注册了一个设备。这使得创建字符设备文件后，发生文件打不开，并返回“此文件不存在”的错误，因为实际的设备其实并不存在。

使用 register_chrdev_region 后，驱动程序就可以工作了。