成绩

华中科技大学

课程设计报告

课　　程：操作系统原理课程设计

课设名称：阻塞和非阻塞式设备驱动开发和内核缓冲区读写同步机制设计

院　　系：软件学院

专业班级：软件 2303 班

学　　号：U202317???

姓　　名：Qthrive

2025 年 06 月 13 日

目 录

1 课设目的

本次课程设计的目的是让学生深入**理解和应用“设备就是文件”这一核心操作系统概念** 。通过亲手实践，期望达到以下目标：

- **熟悉 Linux 设备驱动程序的完整开发过程** ，从编码、编译到加载和测试。
- 深刻**理解设备的阻塞和非阻塞两种工作机制** ，并能在驱动程序中实现这两种机制。
- 学习并**理解和应用内核同步机制**，特别是**等待队列 (wait_queue)** 的使用，以处理并发访问和资源等待问题。
- 提升综合**编程技能** ，具体包括：
  - **内核编程** ：了解内核模块的结构、内核 API 的使用以及内核编程的约束。
  - **驱动开发** ：掌握字符设备驱动的基本框架和实现方法。
  - **同步机制应用** ：如等待队列，用于协调生产者（写入者）和消费者（读取者）的行为。
  - **proc 接口文件系统** ：学习如何通过 proc 文件系统向用户空间暴露驱动程序的状态信息，便于调试和监控 。

2 课程设计内容

本次课程设计主要包含以下两大任务：

- **1. 编写一个设备驱动程序** ：
  - 该驱动程序内部设有一个**固定大小的缓冲区 BUFFER** 。课程建议使用 DEFINE_KFIFO(FIFO_BUFFER, char, 64)来定义一个 64 字节的字符型 FIFO 缓冲区 。
  - 核心要求是实现**读/写操作不遗漏、不重复** ，确保数据的完整性和一致性。
  - 必须**实现设备的阻塞和非阻塞两种工作方式** 。
    - 在阻塞方式下，当缓冲区满时，写操作应阻塞；当缓冲区空时，读操作应阻塞。
    - 在非阻塞方式下,这些操作应立即返回,即使无法立即完成。
  - 利用内核等待队列 (wait_queue_head_t WriteQueue/ReadQueue) 和相关 API (wait_event_interruptible(), wake_up_interruptible()) 来管理进程的阻塞和唤醒。
  - **实现 proc 接口** ，将驱动程序的工作情况（如缓冲区使用情况、内部状态等）记录在 proc 文件系统中，供用户查看 。
- **2. 编写若干具有读/写功能的测试程序** ：
  - 至少编写两个测试应用程序 。

- 通过这些测试程序与设备驱动进行交互，用于**观察缓冲区内容的变化**（通过读取 proc 文件）以及**读/写进程在不同情况下的阻塞与唤醒状态** 。
- 例如，可以设计测试用例：testA 写入 4 字节，testB 写入 2 字节，testC 写入 2 字节，testD 读取 4 字节，testE 读取 8 字节等 。

3 程序设计思路

整个程序设计围绕 Linux 字符设备驱动的开发，结合内核 FIFO 和等待队列机制，并通过 proc 文件系统提供状态观测。

模块划分：

1. **字符设备驱动模块 (my_char_driver.c)**:
   - **核心功能**: 实现字符设备的标准文件操作（open, release, read, write）。
   - **数据缓冲**: 内部集成一个固定大小的 kfifo 作为环形缓冲区，大小为 64 字节 。kfifo 自身保证了数据的先进先出特性，并简化了空/满状态的判断和数据拷贝。
   - **I/O 模式**: 支持阻塞和非阻塞 I/O。
     - **阻塞模式**:
       - 写操作时，若 kfifo 满，则写进程调用 wait_event_interruptible()在 WriteQueue 上等待，直到有空间可用（被读进程唤醒）。
       - 读操作时，若 kfifo 空，则读进程调用 wait_event_interruptible()在 ReadQueue 上等待，直到有数据可读（被写进程唤醒）。
     - **非阻塞模式**: 通过检查文件打开标志 O_NONBLOCK。若操作不能立即完成（如写满、读空），则立即返回-EAGAIN。
   - **同步机制**: 使用两个等待队列头 ReadQueue 和 WriteQueue 。写操作成功后，调用 wake_up_interruptible(&ReadQueue)唤醒可能等待数据的读进程；读操作成功后，调用 wake_up_interruptible(&WriteQueue)唤醒可能等待空间的写进程。
   - **Proc 接口模块**:
     - 创建一个名为 mydevice_status 的 proc 文件。
     - 当用户读取此 proc 文件时，驱动程序将 kfifo 的当前状态（如已用空间、剩余空间、是否为空/满）格式化输出。
2. **用户态测试程序 (如 test_writer.c, test_reader.c)**:
   - **核心功能**: 通过标准的 open(), write(), read(), close()系统调用与/dev/mydevice 设备文件交互。
   - **参数化**: 允许用户指定写入的数据、读取的字节数，以及是否以非阻塞方式打开设备。

- **行为观察**：程序输出操作结果，结合 dmesg 内核日志和 cat /proc/mydevice_status 的输出来验证驱动行为。

设计原理与思路图示：

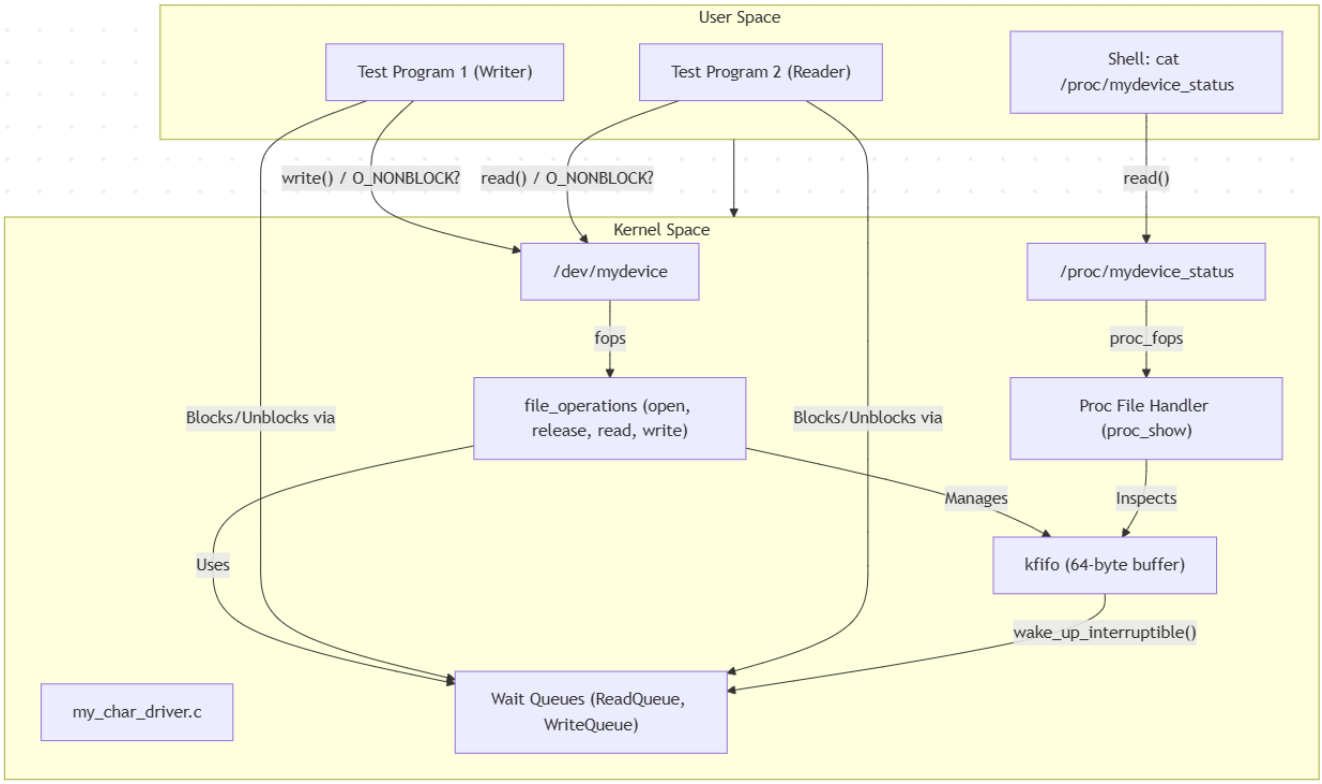

图 1：系统的整体结构图

详细实现思路：

1. **驱动初始化 (my_driver_init)**：
   - 注册字符设备，获取主设备号。
   - 创建设备类和设备节点 (/dev/mydevice)，方便用户访问。
   - 使用 kfifo_alloc()初始化 kfifo 缓冲区，大小为 64 字节 。
   - 使用 init_waitqueue_head()初始化 ReadQueue 和 WriteQueue 。
   - 使用 proc_create()创建/proc/mydevice_status 文件，并关联其读操作处理函数。
2. **驱动读操作 (dev_read)**：
   - 检查 kfifo 是否为空。
   - **阻塞模式**：如果为空且非 O_NONBLOCK，则调用 wait_event_interruptible(ReadQueue, !kfifo_is_empty(&fifo_buffer))使当前进程在 ReadQueue 上睡眠，直到 kfifo 不再为空或收到信号。
   - **非阻塞模式**：如果为空且 O_NONBLOCK，则立即返回-EAGAIN。
   - kfifo 非空时，使用 kfifo_to_user()或 (kfifo_out() + copy_to_user()) 从 kfifo 读取数据到用户提供的缓冲区。

- 如果成功读取数据，调用 wake_up_interruptible(&WriteQueue) 唤醒可能因缓冲区满而等待的写进程。
- 返回实际读取的字节数。

3. **驱动写操作 (dev_write)**:
   - 检查 kfifo 是否已满 (或剩余空间是否小于要写入的数据量)。
   - **阻塞模式**: 如果已满且非 O_NONBLOCK，则调用 wait_event_interruptible(WriteQueue, !kfifo_is_full(&fifo_buffer))使当前进程在 WriteQueue 上睡眠，直到 kfifo 有空间或收到信号。
   - **非阻塞模式**: 如果已满且 O_NONBLOCK，则立即返回-EAGAIN。
   - kfifo 有空间时，使用 kfifo_from_user()或 (copy_from_user() + kfifo_in()) 从用户缓冲区写入数据到 kfifo。
   - 如果成功写入数据，调用 wake_up_interruptible(&ReadQueue)唤醒可能因缓冲区空而等待的读进程。
   - 返回实际写入的字节数。
4. **Proc 文件读取 (proc_show)**:
   - 当用户读取/proc/mydevice_status 时被调用。
   - 使用 seq_printf()输出 kfifo 的当前状态：总大小、已用字节数、可用字节数、是否空/满等。
5. **驱动卸载 (my_driver_exit)**:
   - 移除 proc 文件。
   - 释放 kfifo 缓冲区。
   - 销毁设备节点、设备类。
   - 注销字符设备。

4 实验程序的难点或核心技术分析

重点原理与流程：

1. **字符设备驱动模型**:
   - **原理**: Linux 中设备被抽象为文件。字符设备提供字节流的无结构访问。驱动程序通过 struct file_operations 结构体将其功能函数（如 open, read, write, release）注册到内核，使得用户空间可以通过标准文件 API 与设备交互。
   - **流程**:
     - register_chrdev() (或 alloc_chrdev_region + cdev_add) 注册设备。
     - class_create() 和 device_create() 在 /sys/class 和 /dev 下创建条目。
   - **关键代码**: fops 结构体的填充，module_init 中的注册调用。

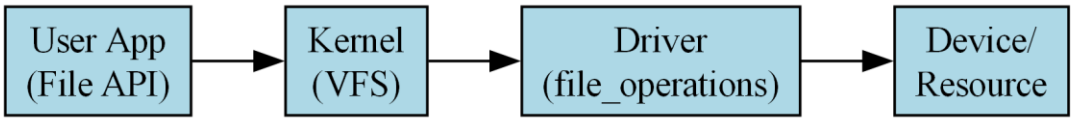

图 2：字符设备驱动模型图

2. KFIFO (Kernel FIFO):
   - **原理**: kfifo 是内核提供的一种优化的、类型安全的、固定大小的先进先出环形缓冲区。它内部处理了读写指针的回绕、空/满判断等逻辑，避免了手动管理这些指针的复杂性和出错风险 。
   - **流程**:
     - 初始化: kfifo_alloc() (动态分配) 或 DEFINE_KFIFO + kfifo_init (静态)。课程建议 DEFINE_KFIFO(FIFO_BUFFER, char, 64) 。
     - 写入: kfifo_in() 或 kfifo_from_user()。
     - 读取: kfifo_out() 或 kfifo_to_user()。
     - 状态检查: kfifo_is_empty(), kfifo_is_full(), kfifo_len(), kfifo_avail()。
   - **关键代码**: kfifo_alloc (或 DEFINE_KFIFO), kfifo_in/out (or kfifo_from/to_user), kfifo_len/avail.

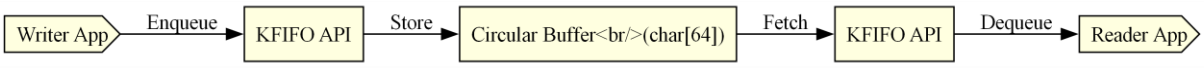

图 3：KFIFO 流程图

3. 等待队列 (Wait Queues) 与阻塞/非阻塞 I/O:
   - **原理**: 当设备无法立即满足请求时（如缓冲区满导致写阻塞，缓冲区空导致读阻塞），进程需要放弃 CPU 并等待条件满足。等待队列是实现这一机制的核心 。
     - **阻塞 I/O**: 进程调用 wait_event_interruptible(queue, condition)，进入睡眠状态，直到 condition 为真（被其他进程通过 wake_up_interruptible(&queue)唤醒）或收到信号。

- **非阻塞 I/O**: 通过检查 file->f_flags & O_NONBLOCK。若操作不能立即完成，驱动返回-EAGAIN，应用程序可以决定重试或做其他事情。

- **流程**:
  - 写操作：若 kfifo 满，阻塞模式下进程在 WriteQueue 上等待。
  - 读操作:若 kfifo 空,阻塞模式下进程在 ReadQueue 上等待。
  - 数据交换后：写操作唤醒 ReadQueue，读操作唤醒 WriteQueue。
- **关键代码**: init_waitqueue_head(), wait_event_interruptible(), wake_up_interruptible(), file->f_flags & O_NONBLOCK.

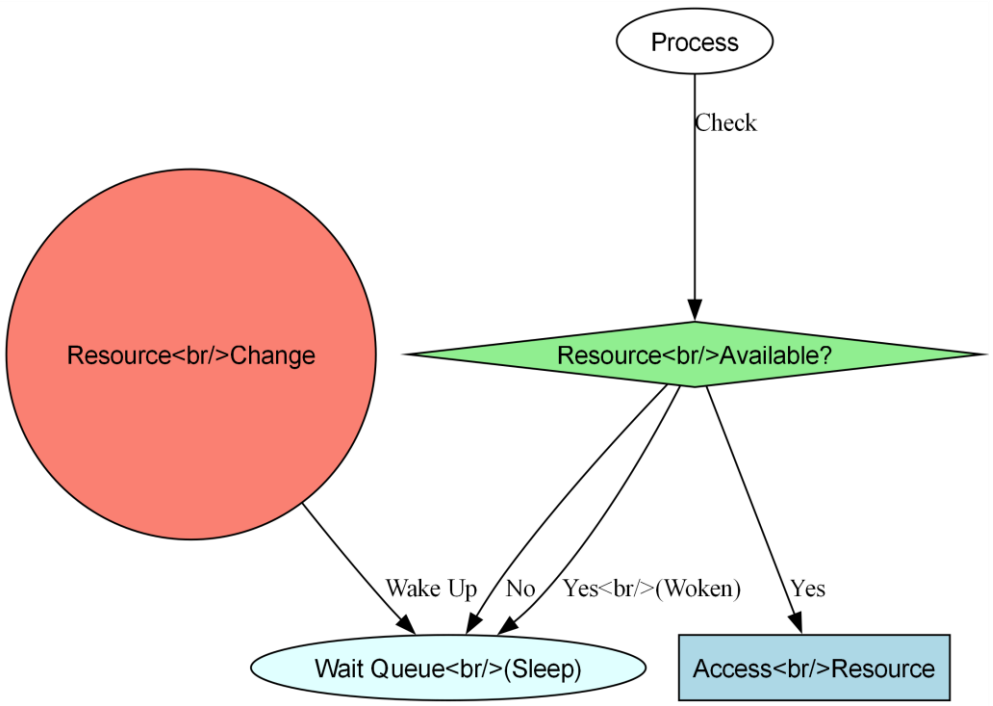

图 4：等待队列示意图

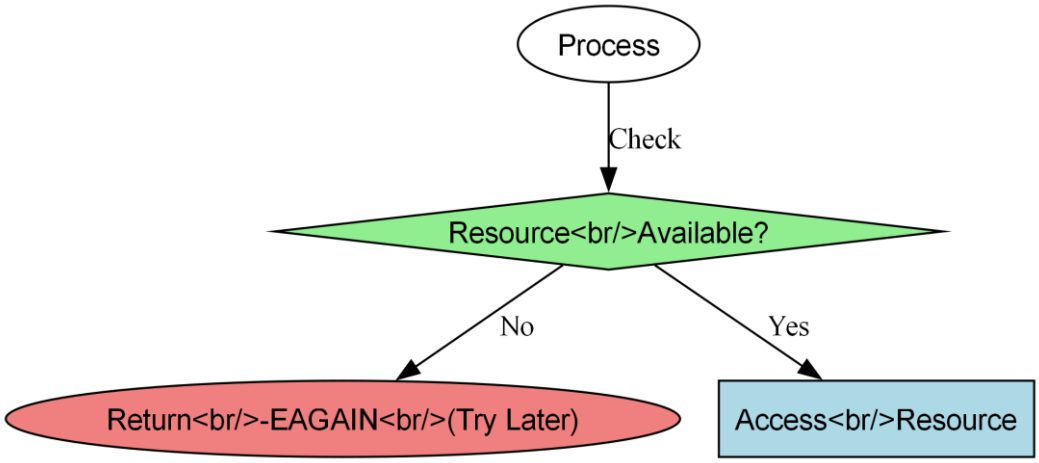

图 5：阻塞与非阻塞运行示意图

4. **Proc 文件系统接口**：
    - **原理**：Procfs 是一个虚拟文件系统，允许内核模块向用户空间暴露信息和提供可调参数。通过创建一个 proc 文件，可以方便地查看驱动内部状态。
    - **流程**：
        - proc_create() 创建 proc 条目，并指定 struct proc_ops（或 struct file_operations for older kernels）来处理文件操作。
        - 通常实现 proc_ops 中的 proc_show（配合 single_open）或直接的 read 函数，用于格式化输出信息。
    - **关键代码**：proc_create()，proc_remove()，proc_ops 结构体及 proc_show 函数实现，seq_file API（seq_printf）的使用。

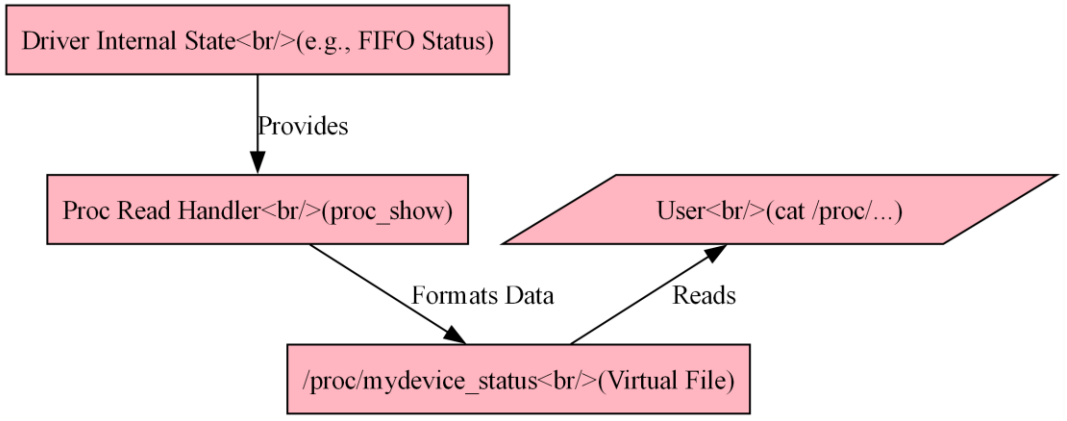

图 6：Proc 接口示意图

遇到的难点及解决方法 / 核心技术细节分析：

1. **并发与同步的正确处理**:
    - **难点**: 多个进程可能同时读写设备。kfifo 函数本身是设计为可以在某些并发场景下安全使用的（如一个单独的 reader 和一个单独的 writer），但当有多个 reader 或多个 writer 时，或者当 kfifo 操作与状态检查之间不是原子操作时，需要额外的同步。
    - **分析/解决方法**:
        - 在我的实现中，kfifo 的读写操作 (kfifo_to_user, kfifo_from_user) 和状态检查 (kfifo_is_empty, kfifo_is_full) 是在 wait_event_interruptible 的循环条件和唤醒后的逻辑中进行的。
        - **关键点**: wait_event_interruptible 在检查条件为假时会自动释放 CPU 并允许中断。当被唤醒时，它会重新检查条件。
        - **关于 FIFO 顺序问题**: 当多个写者因缓冲区满而阻塞，读操作释放空间并唤醒所有等待的写者时，这些写者**会竞争执行**。Linux **调度器决定**哪个写者先运行，从而决定其数据先进入 kfifo。这可能导致数据进入 kfifo 的顺序与写者*尝试*写入的顺序不一致，但 kfifo 本身仍按其接收顺序输出。这并非 kfifo 的错误，而是并发唤醒和调度的自然结果。要严格保证请求顺序，需要更复杂的队列管理和锁机制（如在 dev_write 的入口处加锁，确保一次只有一个写者能尝试放入数据或进入等待队列，但这会降低并发性并可能引入其他问题）。
2. **阻塞与唤醒的精确控制**:
    - **难点**: 确保进程在正确的条件下阻塞，并在正确的时机被精确唤醒，避免死锁或竞争。
    - **解决方法**:
        - 始终在循环中使用 wait_event_interruptible(queue, condition)，因为进程可能被伪唤醒（spurious wakeup）或因信号唤醒但条件仍不满足。
        - 唤醒时机：写操作成功后（数据已入队），唤醒等待数据的读进程 (wake_up_interruptible(&ReadQueue))。读操作成功后（空间已释放），唤醒等待空间的写进程 (wake_up_interruptible(&WriteQueue))。
        - wake_up_interruptible() 只唤醒可中断睡眠的进程。
3. **用户空间与内核空间数据拷贝**:
    - **难点**: 内核不能直接访问用户空间指针。必须使用 copy_from_user()和 copy_to_user()。kfifo_from_user()和 kfifo_to_user()封装了这些操作。
    - **解决方法**: 正确使用这些 API，并检查其返回值以处理可能的错误（如无效的用户地址导致-EFAULT）。

4. **Proc 接口的实现**:
    - **难点**: procfs 的 API 接口多样。尽量使用 seq_file 接口，它能更好地处理 proc 文件内容超过一页的情况，并简化了实现。
    - **解决方法**: 使用 proc_create()创建 proc 文件，并提供一个 proc_ops 结构，其中的 proc_open 函数通常调用 single_open()，它需要一个 show 回调函数（如我们实现的 proc_show）。proc_show 函数使用 seq_printf()来输出信息。

5 开发和运行环境的配置

（一）. 开发环境配置:

- **操作系统**: 优麒麟（内核版本 5.15.0-25-generic）
- **编译器**: GCC (GNU Compiler Collection)
- **内核头文件**:

  Bash

  ```
  sudo apt update
  sudo apt install linux-headers-$(uname -r) build-essential
  ```

- **文本编辑器/IDE**: Vim

（二）. 编写源代码:

- 创建项目目录, /os_project
- 在该目录下创建 my_char_driver.c, Makefile, test_writer.c, test_reader.c.

（三）. 编译程序:

- **编译内核模块**: 在项目目录下打开终端，执行：

  Bash

  ```
  make
  ```

- **编译测试程序**:

  Bash

  ```
  gcc test_writer.c -o test_writer
  gcc test_reader.c -o test_reader
  ```

（四）. 更新与运行程序:

- **加载内核模块（驱动)**:

  Bash

  ```
  sudo insmod my_char_driver.ko
  ```

- **检查模块加载状态**:

  Bash

  ```
  lsmod | grep my_char_driver
  dmesg | tail
  ```

- **查看设备节点**: 设备节点/dev/mydevice 应该已由 device_create 自动创建。

  Bash

  ```
  ls -l /dev/mydevice
  ```

  查看主设备号：

  Bash

  ```
  cat /proc/devices | grep mydevice
  ```

- **运行测试程序**: 打开一个或多个终端：

  Bash

  ```
  # 终端 1：写数据
  sudo ./test_writer "123456789"

  # 终端 2：读数据
  sudo ./test_reader 9

  # 测试非阻塞情况
  sudo ./test_writer "nonblock" nonblock
  sudo ./test_reader 8 nonblock
  ```

- **观察 proc 文件**:

  Bash

cat /proc/mydevice_status

重复此命令以观察缓冲区状态变化。

- **卸载内核模块**:

  Bash

  sudo rmmod my_char_driver

  再次用 dmesg | tail 查看卸载信息。

- **更新程序**:
  1. 修改 my_char_driver.c 或测试程序源码。
  2. 如果内核模块已加载，先卸载: sudo rmmod my_char_driver
  3. 重新编译:

     make clean && make,

     gcc test_writer.c -o test_writer

     gcc test_reader.c -o test_reader

  4. 重新加载模块: sudo insmod my_char_driver.ko
  5. 重新运行测试程序。

（五）. 调试程序:

- **pr_info/pr_debug**: 在驱动代码中加入打印语句,通过 dmesg 查看执行流程和变量值。pr_debug 需要特定配置才能显示。
- **Proc 文件**: 利用/proc/mydevice_status 输出内部状态信息。
- **GDB (for user-space)**: 测试程序可以用 GDB 调试。
- **分析 dmesg 输出**: 内核的警告和错误信息有助于定位问题。

6 运行和测试过程

（一）安装驱动模块:

执行 sudo insmod my_char_driver 命令后，在 dmesg | tail 中应当能看到设备初始化信息，如下图所示：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo insmod my_char_drive
r.ko
[sudo] ukylin 的密码：
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ dmesg | tail
dmesg: 读取内核缓冲区失败：不允许的操作
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo dmesg | tail
[ 7529.858287] audit: type=1400 audit(1747643726.773:88): apparmor="DENIED" oper
ation="capable" profile="/usr/sbin/cupsd" pid=780581 comm="cupsd" capability=12
 capname="net_admin"
[ 7529.893730] audit: type=1400 audit(1747643726.809:89): apparmor="DENIED" oper
ation="capable" profile="/usr/sbin/cups-browsed" pid=780622 comm="cups-browsed"
capability=23  capname="sys_nice"
[ 7571.664758] 我的设备：正在初始化MyDevice LKM
[ 7571.664765] 我的设备：主设备号235注册成功
[ 7571.664923] 我的设备：设备类注册成功
[ 7571.666037] 我的设备：设备节点创建成功
[ 7571.666039] 我的设备：KFIFO初始化成功
[ 7571.666067] 我的设备：等待队列初始化完成
[ 7571.666093] 我的设备：/proc/mydevice_status创建成功
[ 7571.666094] 我的设备：LKM初始化成功
```

图 7：模块安装示意图

（二） 测试 proc 接口：

在终端运行 sudo cat /proc/mydevice_status 命令后可以观察到设备当前状态以及缓冲区使用情况，如下图所示：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo cat /proc/mydevice_s
tatus
--- 我的设备状态 ---
缓冲区大小    : 64 字节
缓冲区中数据  : 0 字节
剩余空间      : 64 字节
缓冲区状态  : 空
```

图 8：proc 接口测试示意图

（三） 读写测试：

1. 空读取测试（阻塞）：

在缓冲区还没有任何内容时，对设备进行读操作，输入命令 sudo ./test_reader 30，此时会发生阻塞：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_reader 30
测试读取程序：打开 /dev/mydevice...
测试读取程序：尝试读取30字节数据...
```

图 9：空读取（阻塞）示意图

随后用 sudo ps aux | grep test_命令查看进程状态，状态为 S 代表进

程处于可中断睡眠状态，即正在等待某项 I/O 操作的完成：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ps aux | grep test_
root     1047388  0.0  0.1  19916  6088 pts/1    S+   17:18   0:00 sudo ./test_r
eader 8
root     1047389  0.0  0.0  19916   904 pts/2    Ss   17:18   0:00 sudo ./test_r
eader 8
root     1047390  0.0  0.0   2776   956 pts/2    S+   17:18   0:00 ./test_reader
 8
ukylin   1049776  0.0  0.0  14780  2344 pts/0    S+   17:18   0:00 grep --color=
auto test_
```

图 10：空读取进程状态示意图

此时在另一个终端写入，内容会被立即读出：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo cat /proc/mydevice_s
tatus
--- 我的设备状态 ---
缓冲区大小    : 64 字节
缓冲区中数据 : 0 字节
剩余空间      : 64 字节
缓冲区状态 : 空
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "测试
空读取"
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："测试空读取"（15 字节）
测试写入程序：消息成功写入(15 字节)。
测试写入程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_reader 30
测试读取程序：打开 /dev/mydevice...
测试读取程序：尝试读取30字节数据...
测试读取程序：接收到15字节数据："测试空读取"
测试读取程序：关闭设备...
```

图 11：空读取写入运行示意图

2. 空读取测试（非阻塞）：

在非阻塞模式下，使用命令 sudo ./test_reader 30 nonblock：空缓冲区被读取时会直接返回错误代码，然后直接关闭设备：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo cat /proc/mydevice_s
tatus
--- 我的设备状态 ---
缓冲区大小    : 64 字节
缓冲区中数据 : 0 字节
剩余空间      : 64 字节
缓冲区状态 : 空
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_reader 30 non
block
测试读取程序：打开 /dev/mydevice...
测试读取程序：尝试读取30字节数据...
测试读取程序：读取会阻塞(已设置非阻塞模式)或缓冲区为空，读取0字节。
测试读取程序：关闭设备...
```

图 12：空读取（非阻塞）运行示意图

3. 满写入测试（阻塞）：

当缓冲区满时继续进行写入，此时进程会阻塞，直到缓冲区中有数据被读出，留出空间之后才能写入：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo cat /proc/mydevice_s
tatus
[sudo] ukylin 的密码：
--- 我的设备状态 ---
缓冲区大小    : 64 字节
缓冲区中数据 : 64 字节
剩余空间      : 0 字节
缓冲区状态 : 满
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "1234"
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："1234"（4 字节）
```

图 13：满写入（阻塞）运行示意图

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ps aux | grep test_
root     1364745  0.0  0.1  19912  6080 pts/0    S+   20:15   0:00 sudo ./test_w
riter 1234
root     1364750  0.0  0.0  19912   904 pts/2    Ss   20:15   0:00 sudo ./test_w
riter 1234
root     1364751  0.0  0.0   2776   988 pts/2    S+   20:15   0:00 ./test_writer
 1234
ukylin   1366088  0.0  0.0  14780  2400 pts/1    S+   20:15   0:00 grep --color=
auto test_
```

图 14：满写入（阻塞）程序状态图

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_reader 4
[sudo] ukylin 的密码：
测试读取程序：打开 /dev/mydevice...
测试读取程序：尝试读取4字节数据...
测试读取程序：接收到4字节数据："1234"
测试读取程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "1234"
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："1234"（4 字节）
测试写入程序：消息成功写入(4 字节)。
测试写入程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
```

图 15：满写入解除阻塞运行示意图

4. 满写入测试（非阻塞）：

当缓冲区已满时，在终端输入命令：sudo ./test_writer “1234” nonblock，由于是非阻塞模式，所以会直接返回错误信息并退出设备：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo cat /proc/mydevice_s
tatus
--- 我的设备状态 ---
缓冲区大小    : 64 字节
缓冲区中数据  : 64 字节
剩余空间      : 0 字节
缓冲区状态  : 满
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "1234"
 nonblock
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："1234"（4 字节）
测试写入程序：写入会阻塞(已设置非阻塞模式)或缓冲区已满，写入 0 字节。
测试写入程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
```

图 16：满写入（非阻塞）示意图

4. 进程竞争及数据完整性测试：

当缓冲区已满时，在外部继续开启两个进程，一个写入 1234，另一个写入 5678，开始时，先开启写入 1234 的进程，再开启写入 5678 的进程，然后开启一个读进程，一次性读取 64 字节的数据，此时两个写进程同时解除阻塞，分别写入成功，此时读取 8 个字节，观察输出内容：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_reader 64
测试读取程序：打开 /dev/mydevice...
测试读取程序：尝试读取64字节数据...
测试读取程序：接收到8字节数据："56781234"
测试读取程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "1234"
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："1234"(4 字节)
测试写入程序：消息成功写入(4 字节)。
测试写入程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$

测试写入程序：写入消息："5678"(4 字节)
测试写入程序：消息成功写入(4 字节)。
测试写入程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
```

图 17：进程竞争示意图 1

我发现输出内容是 56781234，这与进程的启动顺序相反，我感到疑惑，于是我再次进行实验，仍然先启动写入 1234 的进程，再次观察输出：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_reader 8
测试读取程序：打开 /dev/mydevice...
测试读取程序：尝试读取8字节数据...
测试读取程序：接收到8字节数据："12345678"
测试读取程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "1234"
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："1234"(4 字节)
测试写入程序：消息成功写入(4 字节)。
测试写入程序：关闭设备...
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "5678"
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："5678"(4 字节)
测试写入程序：消息成功写入(4 字节)。
测试写入程序：关闭设备...
```

图 18：进程竞争示意图 2

此时发现运行结果改变了，我带着疑惑查询资料，给出的解释如下：

在没有显式同步机制（如互斥锁）的情况下，写入顺序的不确定性是并发系统的

固有特性，并非驱动的 “错误”。它恰恰体现了内核调度器与等待队列的设计逻辑：

等待队列负责管理阻塞的进程；

调度器负责在资源可用时决定进程的执行顺序（这是一个 **“抢占式”** 的动态过程）。

7 实验心得和建议

一、问题和挑战：

在实验中，我遇到了一些挑战和问题，总结如下：

（一）命令操作问题：

在实验中，我开始会用./test_writer “1234”来进行写数据，但是会报错，程序提示 Permission denied，后来我知道了用 sudo 管理员权限进行操作，问题得到解决，如图所示：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ ./test_writer "1234"
测试写入程序：打开 /dev/mydevice...
测试写入程序：打开设备失败：Permission denied
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ sudo ./test_writer "1234"
[sudo] ukylin 的密码：
测试写入程序：打开 /dev/mydevice...
测试写入程序：写入消息："1234" (4 字节)
测试写入程序：消息成功写入(4 字节)。
测试写入程序：关闭设备...
```

图 19：命令操作问题

（二）编译问题：

在实验中，我在主机上编写好 Makefile 文件，然后通过微信的文件传输助手传输到虚拟机上，然后直接用 make 命令编译，然后程序报错，提示缺失分隔符，如图：

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ make
Makefile:7: *** 缺失分隔符。 停止。
```

图 20：缺失分隔符报错

进入 Makefile 文件后发现，Windows 系统的 tab 和 Linux 的 tab 有些区别，导致编译失败

```
obj-m += my_char_driver.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

图 21：缺失分隔符代码图

纠正后能够正常编译：

```
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

```
ukylin@ukylin-vmwarevirtualplatform:~/桌面/os_project$ make
make -C /lib/modules/6.2.0/build M=/home/ukylin/桌面/os_project modules
make[1]: 进入目录“/usr/src/linux-headers-6.2.0”
  CC [M]  /home/ukylin/桌面/os_project/my_char_driver.o
/home/ukylin/桌面/os_project/my_char_driver.c:26:20: warning: ‘my_cdev’ defined
but not used [-Wunused-variable]
   26 | static struct cdev my_cdev;              // 字符设备结构体
      |                    ^~~~~~~
  MODPOST /home/ukylin/桌面/os_project/Module.symvers
  CC [M]  /home/ukylin/桌面/os_project/my_char_driver.mod.o
  LD [M]  /home/ukylin/桌面/os_project/my_char_driver.ko
  BTF [M] /home/ukylin/桌面/os_project/my_char_driver.ko
Skipping BTF generation for /home/ukylin/桌面/os_project/my_char_driver.ko due t
o unavailability of vmlinux
make[1]: 离开目录“/usr/src/linux-headers-6.2.0”
```

图 22：缺失分隔符修复图

（三）进程竞争的疑惑：

如上文所示，在进行进程竞争的测试时，发现两个写入进程同时被唤醒时的顺序不同，导致写入的内容的顺序不同，但是两个进程写入的内容都是

完整的，重复实验也发现唤醒顺序不一定固定，通过查阅资料和网站，我得知了原因：

由于我没有**显式实现同步机制（互斥锁）**，在这种情况下，写入顺序的**不确定性**是并发系统的**固有特性**，并非驱动的 “错误”。它恰恰体现了内核调度器与等待队列的设计逻辑：

1. **等待队列**负责管理阻塞的进程；
2. **调度器**负责在资源可用时决定进程的执行顺序（这是一个 **“抢占式”** 的动态过程）。

二、心得：

在实验中，我遇到了很多困难，但是都通过自己查阅资料，借助他人的智慧成功解决了难题，也从中学到了很多与驱动开发相关的知识，我也知道了学习知识一直都是站在前人的肩膀上的，遇到困难不能自己一个人死磕，理清问题后向他人寻求帮助才是最智慧的选择。

8 学习和编程实现参考网址

(1) **《操作系统原理》教材**及**课程课件** ：提供了本次课设的基础理论知识。

(2) **www.kernel.org/doc/html/latest/**： Linux 内核的官方文档，我主要参考了字符驱动设备、kfifo 以及 procfs 的内容。

(3)**www.lwn.net**:这是包含大量关于 Linux 内核开发的文章和教程,对我写 kfifo 和 procfs 的内容起到了很多帮助。

(5) **Stack Overflow**:这个论坛对于各种错误有很完整的收录和回答，我写程序的过程中多次用到，比如 sudo 命令的使用，进程竞争的问题等等。

(6) **https://blog.csdn.net/**: 在 CSDN 中我找到了很多具体的驱动开发教程，包含详细的代码和步骤，为我带来了很多方便。