目　录

1 实验一：任务切换机制

1.1 实验概述

1.1.1 课设任务

理解并重现或实现一个基于极简页式内存机制的保护模式程序。

1.1.2 课设内容

1. 阅读和理解 X86 保护模式的系列程序（pmtest1 至 pmtest5）；
2. 阅读和理解 X86 保护模式的系列程序（pmtest6 至 pmtest7）；
3. 编程实现极简页式内存机制的保护模式程序：
   (1) CPU 进入保护模式；
   (2) 初始化 GDT、LDT、IDT 和 TSS 等数据结构；
   (3) 对内存采取极简方式页式化；
   (4) 在时钟驱动下支持 2 个任务切换：
      a. 每个任务使用各自对应的页表；
      b. 每个任务简单地输出 A 或 B。

1.2 实验设计思路

由于实验要求使用 LDT 配合 TSS 完成任务切换工作，同时要求任务的工作可以在不同页中执行，二者互不干扰，因此选择采用分页技术实现，来完成对相同线性地址到不同物理地址的映射工作。另外，实验要求根据时钟中断响应任务切换，因此还需要配置 IDT 并打开时钟中断。综上对程序进行如下编写实现的流程规划：

1、 完成对 GDT 表的初始化工作，主要是说明各个段的分配地址和属性情况，其中主要包括对页地址分配、LDT 表和 TSS 的构建等；
2、 完成对 LDT 段中内容的声明，包括对代码段和堆栈段的声明；
3、 完成对页目录表和页表的分配，同时指明线性地址到物理地址的映射关系。
4、 编写时钟中断相应程序调用的两个显示字符串的程序，并将其在程序运

行时（触发时钟中断前）利用 lib.inc 中的函数 MemCpy 复制到各自页表所在位置；

5、 填充 TSS，包括对页表切换所需的 CR3 的指定、允许中断的 EFLAGS 的设置、代码段 CS、数据段 DS、附加段 ES、自有堆栈段 ES 和 EDI 的指定，以及对 LDT 和位图的声明；

6、 为实现时钟中断，首先完成对 IDT 的初始化工作，程序执行过程中完成对 IRQ0 的设置以配置时钟中断；

7、 主程序执行流程中首先从实模式跳转进入保护模式，接着打开分页机制，再装载 LDT 并配置 TR，最后打开时钟中断并跳入任务 1 执行。

为更直观的描述程序执行流程，将上述编写流程转换为执行流程并使用下述流程图进行描述，具体如下：

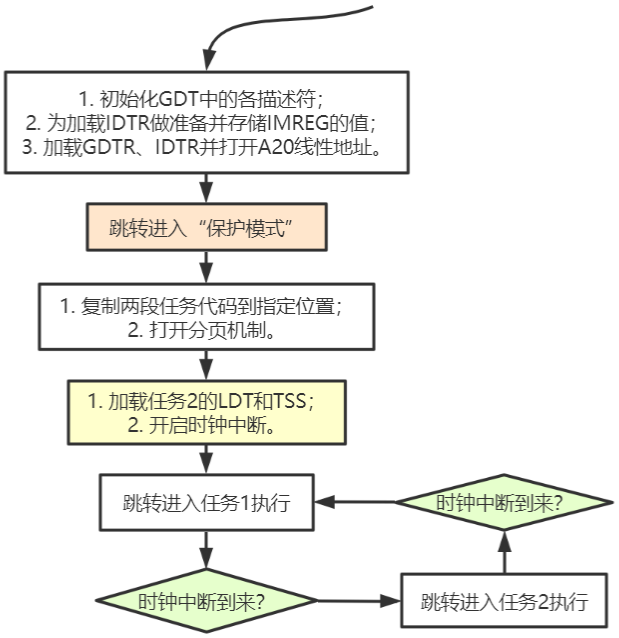

图 1-1 程序执行流程图

1.3 实验难点及核心技术分析

1.3.1 LDT 的配置

LDT 与 GDT 的功能类似，是 GDT 的局部描述的细化，因此对 LDT 的配置应首先完成其在 GDT 中的声明。接着由于希望两个程序独立开来，因此为每个程序都创建一个 LDT，其中用来声明他们的代码段和堆栈段。

以任务 1 的 LDT 举例如下图所示：

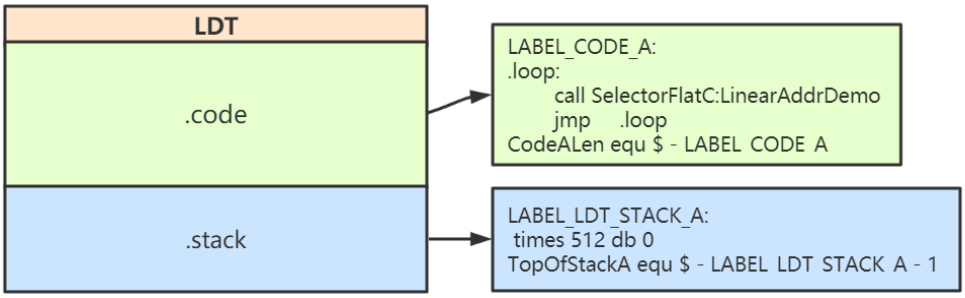

图 1-2 LDT 配置方案图

1.3.2 分页机制的实现

分页机制的实现依托于三点：一是初始化页目录和对应的页表，二是打开分页标志位 PG，三是切换 CR3。

首先解决初始化页目录和页表的问题。为更直观的展示页目录、页表以及执行程序代码之间的关系，笔者将三者放在同一个段中进行编写。该段的线性地址空间为 0~4GB，从低地址到高地址分别存放页目录表 A、页表 A、页目录表 B、页表 B、任务 1 代码和任务 2 代码。另外，由于不仅需要对该段进行读写操作，同时还需执行其中的代码，因此使用两个属性不同的描述符来对该段进行描述，这两个描述符均需要在 GDT 中声明并初始化，分别为 SelectorFlatC 和 SelectorFlatRW。

由于 Linux 的地址映射机制是逻辑地址转换为线性地址再转换为物理地址，因此我们需要给这些页目录、页表和程序代码指定物理地址进行实际存储，同时并规定一个线性地址用于程序操作时的访问。该段的划分和指定线性地址的映射方式如下图所示：

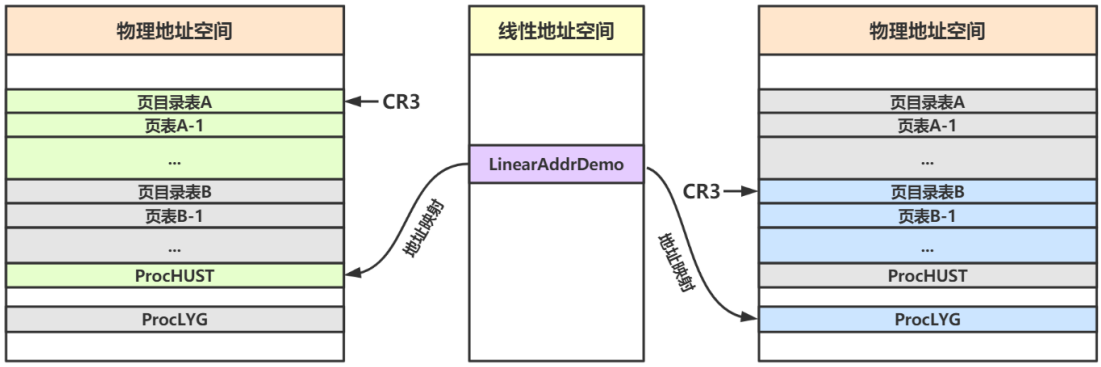

图 1-3 物理地址空间分配及地址映射关系图

接着详细描述地址分配和映射过程的实现。首先初始化页目录，基址为 SelectorFlatRW 描述符所指向的地址，页目录 A 和页目录 B 与基址的偏移分别用 PageDirBase0 和 PageDirBase1 来标记，对应的页表起始地址偏移分别用

PageTblBase0 和 PageTblBase1 来标记。根据以上地址空间的描述来初始化页目录表和其对应的页表。由于任务程序分别存储在 ProcHUST 和 ProcLYG 所指位置处，而分页机制的核心就是不使用物理地址而是经过线性地址映射后找寻物理地址，因此，笔者对两程序指定了相同的线性地址即 LinearAddrDemo。此外，由于当完成页面的切换时会导致 CR3 寄存器的更改，而进行寻址时首先会切换到 CR3 指定的页目录表，然后再根据线性地址的高 10 位查找该目录表找到对应页表地址，再从页表中根据线性地址的第 12~21 位查找实际的物理页首地址，将该物理页基址加上线性地址低 12 位即为其对应的物理地址，最后跳转至该物理地址。因此需要完成对 LinearAddrDemo 线性地址到物理地址的映射，来保证该线性地址转换到的物理地址中存放着指定任务的第一条指令。

至此完成了对所需页目录表、页表的初始化，接下来打开分页标志 PG 位，使得分页机制可以成功运行。

PG 位的开启较为简便，仅需将 CR0 的 PG 位（第 31 位）置 1 即可。另外，在对页目录表和页表进行初始化时也需要指定“页存在”属性以及读写执行属性等，以保证其在运行过程中可以被正常访问或执行。

1.3.3 TSS 的配置

由于需要进行两个任务的切换，因此使用 TSS 结构来实现对任务更改时各寄存器的管理。

首先来明确 TSS 的功能作用。当任务发生切换时，首先将当前任务的各类寄存器包括 EFLAGS 放入当前任务对应的 TSS 中，然后再从目标任务对应的 TSS 中加载信息执行目标任务。

因此，我们配置两个任务对应的 TSS。首先在 GDT 中进行 TSS 的声明，之后便可以使用其对应的选择子来完成任务切换。TSS 的具体内容中重点完成以下内容的设置（以任务 1 的 TSS 为例）：

表 1-1 TSS 重点内容对照表

名称	值	作用
CR3	PageDirBase0	指定页目录表地址
EFLAGS	0200h	允许中断
ESP	TopOfStackA	指定私有堆栈栈顶
EDI	PageDirBase0	指定页目录起始地址
ES	SelectorFlatRW	指定页目录所在段基址

CS	SelectorLDTCodeA	私有代码段基址
SS	SelectorLDTStackA	私有堆栈段基址
DS	SelectorData	私有数据段基址
LDT	SelectorLDTA	LDT
其他	0	--

根据上表信息，当任务首次发生转换时，便会读取上表中的信息以加载任务执行；当切出任务时会更新该 TSS 的内容以备下一次切进该任务时使用。

1.3.4 时钟中断的实现

要实现时钟中断首先要配置 IDT 来完成对中断的描述。为方便编写，将 255 个中断的对应调用程序均设置为时钟中断响应的代码。配置完 IDT 后需要将其加载，使用指令“lidt”即可。中断向量与对应中断处理过程的对应方式如下图所示：

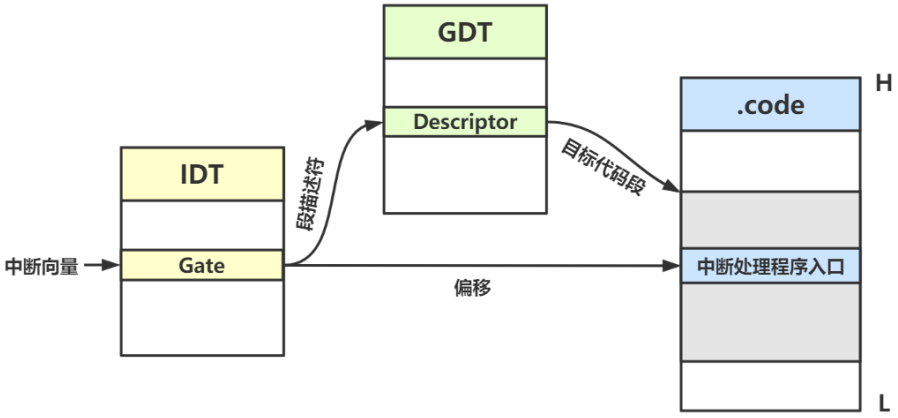

图 1-4 中断向量到中断处理程序对应过程图

其次，由于时钟中断的特殊性，还需要建立硬件中断与向量号之间的对应关系。时钟中断采用可屏蔽中断实现，通过可编程中断控制器 8259A 建立联系，因此需要对其主从引脚信息进行配置，具体如下图所示：

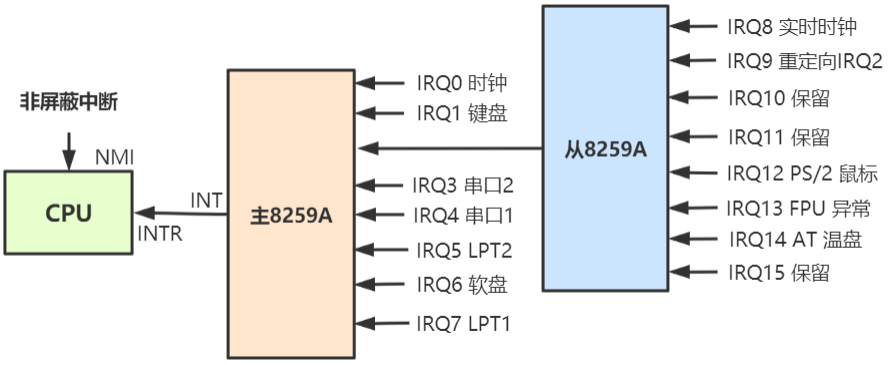

图 1-5 8259A 引脚示意图

在配置 8259A 后，当发生时钟中断时，会去其绑定的中断号即 20h 处找到中断向量，然后根据中断向量表所示转向该中断响应程序执行。

1.3.5 任务切换的实现

要实现任务切换达到屏幕上不断来回输出不同的字符串的效果，首先需要明确任务切换功能最初设计的目的。任务切换的初衷是为了解决单一进程长时间占用 CPU 导致其他进程无法执行的问题，因此，采用时间分片的基址配合任务切换使得各进程轮流占用 CPU。根据这一原则，在设计程序时，考虑到任务一直切换没有停止，因此程序代码采用死循环结构。当时钟中断到来时，首先调用任务 1 执行，当时钟中断再次到来时，首先将当前任务 1 的状态压入 TSS1 然后从 TSS2 中加载任务 2 的状态信息并执行任务 2；当时钟中断下次到来时，TSS2 记录当前执行状态，并从 TSS1 中加载任务 1 状态并执行；同时，由于两个任务均由死循环构成，因此会导致两任务都没有执行出口，故而达到了根据时钟中断持续切换显示不同字符串的效果。

此外，任务切换具体执行哪个任务是根据 current 变量作为判断依据，current 为 0 时执行任务 1，为 1 时执行任务 2，在每次切换任务前都将 current 值改为下次要执行的任务的代号，方便任务切换的实现。

时钟中断的任务切换代码如下图所示：

```
_ClockHandler:
ClockHandler    equ     _ClockHandler - $$
        push    eax
        mov     ax, [current]
        cmp     ax, 0
        jz      .hust

        mov     byte [current], 0               ; # 当current为1时输出'LYG '
        mov     al, 20h
        out     20h, al                         ; 发送 EOI
        jmp     SelectorTSSB:0
        pop     eax
        iretd
.hust:                                          ; # 当current为0时输出'HUST'
        mov     byte [current], 1
        mov     al, 20h
        out     20h, al                         ; 发送 EOI
        jmp     SelectorTSSA:0
        pop     eax
        iretd
```

图 1-6 时钟中断任务切换实现代码图

如上图所示，上述代码中仍有以下三个细节需要阐述：第一，在切换任务前，需要发送 EOI 来保证时钟中断下次可以被成功触发。第二，由于时钟中断触发时没有对寄存器进行保护，因此为了防止寄存器信息被更改，应首先将需要用到的

寄存器压入栈中。第三，对任务切换的代码执行过程进行详述：以任务 2 执行时为例，当首次发生时钟中断时（称为第 n 次时钟中断），由于需要调用时钟中断处理程序，因此任务 2 会将任务 2 的下一条指令及对应寄存器状态压入栈中，然后再进入中断处理程序。此时，根据 current 判断将要执行“jmp SelectorTSSA:0”指令，TSS_B 中记录下一条指令即“pop eax”，随后从 TSS_A 中读取信息进入任务 1 中执行。当再次发生时钟中断时（称为第 n+1 次时钟中断），会将任务 1 的下一条指令压入栈中再调用中断处理程序，并根据 current 判断将要执行“jmp SelectorTSSB:0”指令，TSS_A 中记录下一条“pop eax”指令。此时由于执行“jmp SelectorTSSB:0”指令，因此会从 TSS_B 中取出寄存器状态并得知要执行“pop eax”指令，于是开始执行该指令及其后指令。当继续执行直到执行完“iret”指令后，由于任务 2 响应时间中断前以将任务 2 程序的下一条指令压入栈中，因此会根据栈中寄存器状态执行第 n 次时间中断触发前任务 2 执行到的下一条指令，保证了任务指令执行顺序得到正确的衔接。剩下的时钟中断响应过程与上述过程类似，不再赘述。

根据上述描述可知，当正在运行的任务切换后想要再切换回任务并继续执行的条件是，原任务必须已在运行状态，否则会导致“iret”执行后返回位置错误而导致任务切换失败。因此在开启 8259A 实现时钟调用之前，应该首先加载第 2 次中断才能执行的任务 2 的 LDT 和 TSS 并跳转入该任务执行。具体实现如下图所示：

```
    ; 加载LDT
    mov     ax, SelectorLDTB
    lldt    ax
    ; 配置TR
    mov     ax, SelectorTSSB
    ltr     ax
    ; 配置可编程控制中断器
    call    Init8259A
    sti
    jmp     SelectorLDTCodeB:0      ; # 进入任务1打印'LYG '
```

图 1-7 实现时钟中断前的准备代码图

另外还需补充说明的是，由于程序同时采用了分页机制，因此两任务的代码段均为一行指令，即循环调用指定线性地址的程序。具体以任务 1 为例，如下图所示：

```
LABEL_CODE_A:
.loop:
        call    SelectorFlatC:LinearAddrDemo
        jmp     .loop
```

图 1-8 任务调用代码图

综上，当使用 TSS 进行任务切换时，会执行上述代码，调用上述线性地址的

代码执行，虽然两任务调用的是同一线性地址的代码，但是由于两个 TSS 中指定的 CR3 存在不同，因此物理地址并不相同，故将执行不同的代码。页面切换与线性地址到物理地址的映射已在 1.3.2 节中详细阐述，在此不再赘述。

1.4 运行与测试

1.4.1 测试环境搭建

为解决本次实验的环境配置问题，笔者起初采用了 VMware 上搭建 Ubuntu 虚拟机的方式，但由于虚拟机环境运行响应速度较慢，因此最终选择了 WSL2 下配置图形化界面后搭建 bochs 的实验环境。

由于 bochs 的安装需要图形化界面的支持，因此首先完成对 WSL2 图形化界面的配置。图形化界面采用“VcXsrc+xfce4+xubuntu”的方式，下载并安装 VcXsrc，接着利用命令安装剩余二者，命令如下：

```
sudo apt-get install xfce4
sudo apt-get install xubuntu-desktop
```

安装完成后向“~/.bashrc”文件中添加 xfce4 与 VcXsrc 间通讯的配置信息，以保证二者在每次启动后都能正常通信，具体为：

```
export DISPLAY=`cat /etc/resolv.conf | grep nameserver | awk '{print $2}'`:0
```

至此打开 VcXsrc 的 XLaunch 程序和 WSL2 后即可使用图形化界面。

完成图形化界面的配置后即可安装 bochs，安装具体步骤与任务书中大致一致，因此不再赘述，仅将编译 bochs 的配置信息作如下展示：

```
./configure --enable-debugger --enable-disasm --enable-iodebug --enable-x86-debugger --
with-x --with-x11
```

至此，完成实验环境的搭建。

1.4.2 测试过程

对程序进行编译并写入软盘 B 中，如下：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/workspace$ make
sudo mount -o loop pm.img /mnt/floppy/
sudo cp mypmtest.com /mnt/floppy/ -v
'mypmtest.com' -> '/mnt/floppy/mypmtest.com'
sudo umount /mnt/floppy/
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/workspace$
```

图 1-9 编译装盘图

运行程序，结果如下：

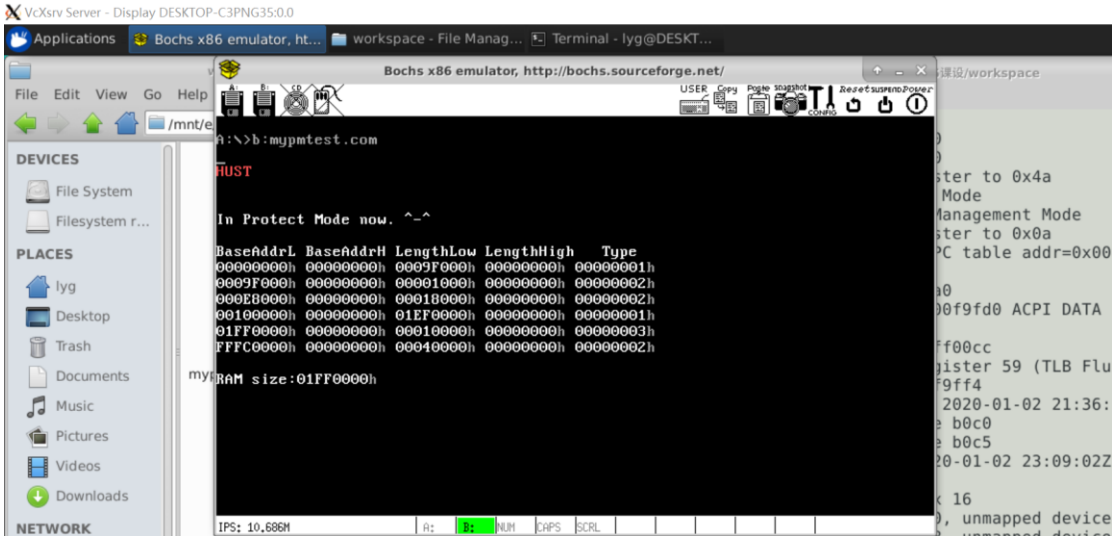

图 1-10 程序运行结果图 1

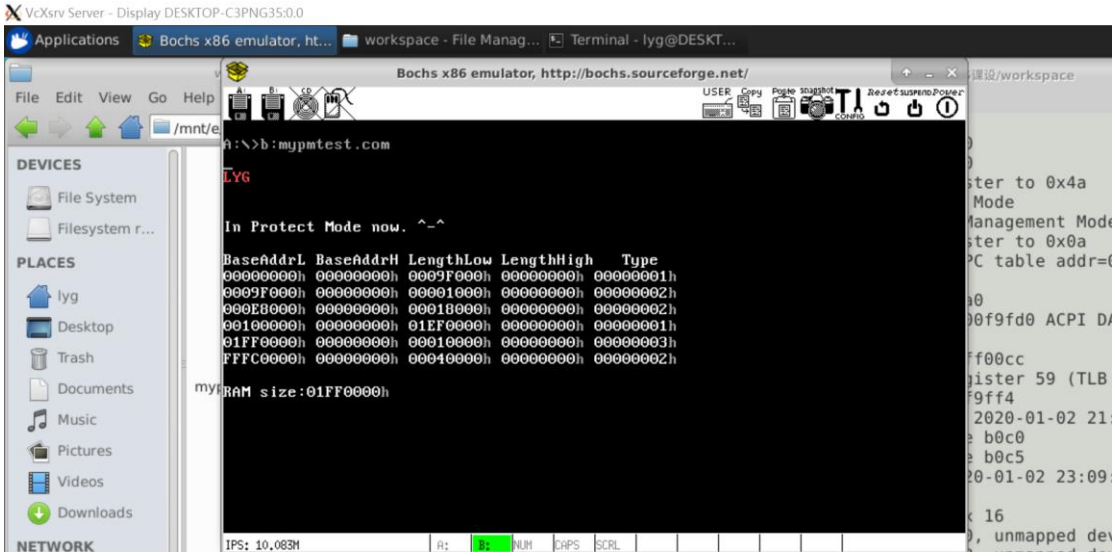

图 1-11 程序运行结果图 2

1.5 参考资料

[1]. 于渊.《Orange's 一个操作系统的实现》.北京:电子工业出版社,2009

[2]. https://github.com/wlmnzf/oranges

2 实验二：设备阻塞工作机制

2.1 实验概述

2.1.1 课设任务

1. 理解和应用“设备就是文件”的概念；
2. 熟悉 Linux 设备驱动程序开发过程；
3. 理解和应用内核等待队列同步机制

2.1.2 课设内容

1. 在 Linux 下编写设备驱动程序；
   (1) 内设固定大小缓冲区 BUFFER，保证读或写不遗漏不重复；
   (2) 实现设备的阻塞和非阻塞两种工作方式。
2. 编写不少于两个对设备进行读或写的测试应用程序，观察缓冲区变化与读或写进程的阻塞或被唤醒的同步情况。

2.2 实验设计思路

由于之前已经具备编写 Linux 驱动程序的经验，因此对 Linux 驱动框架不再详述，仅从本次实验目的出发对该同时满足阻塞与非阻塞工作方式的驱动编写流程进行阐述。

本实验代码的主体主要参考了实验指导书中的代码结构，但是由于上述结构在多进程测试时可能出现 kfifo 缓冲区溢出的情况（具体见 2.3 节的分析，在此不赘述），因此在上述结构的基础上对代码进行了修改。

使用内核互斥锁“struct mutex”结构体对 kfifo 缓冲区读写条件判断进行加锁，防止多个进程同时判断而导致结果一致从而使得同时读或写缓冲区而导致溢出。另外，为防止唤醒条件中仅判断缓冲区状态而导致进程被全部唤醒后均可获得 kfifo 资源导致的缓冲区溢出，对读写操作设置互斥锁。且由于采用的是循环队列存储结构，因此对读操作和写操作分别设置互斥锁，而非仅设置一个锁。

为实现实验要求中阻塞和非阻塞工作方式同时兼备，当以阻塞模式访问时，在未申请到锁之前进程将挂起；而当以非阻塞模式访问时，一旦没有申请到锁或条件不满足，则直接返回。

根据上述描述，以驱动“读”操作部分为例，展示其实现流程图如下：

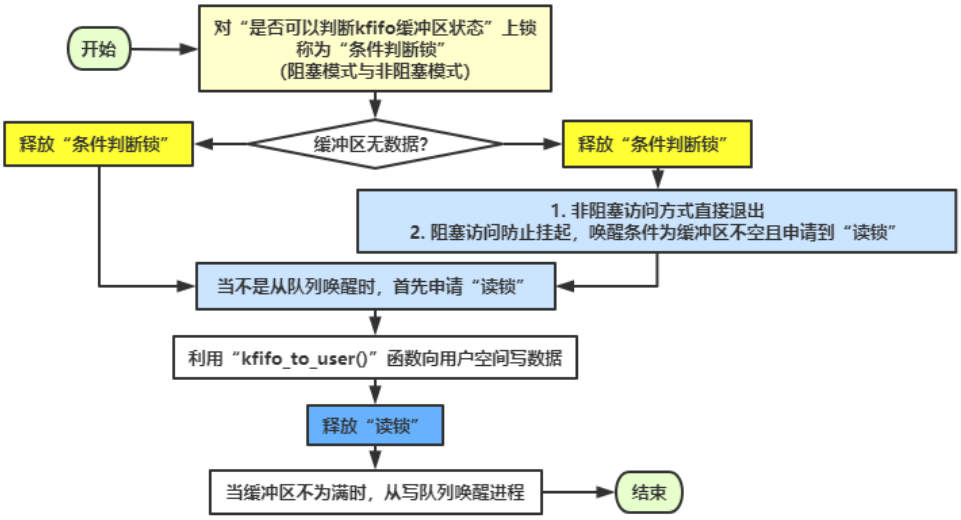

图 2-1 驱动读操作执行流程图

2.3 实验难点及核心技术分析

2.3.1 参考驱动程序在多进程下的缓冲区溢出问题的引出

为深度剖析该问题，首先对参考驱动程序进行相关测试，然后再根据测试结果进行详细分析，以下将根据上述流程对参考驱动程序导致的缓冲区溢出问题进行详细分析。

根据实验指导书中的参考驱动程序矿建对该驱动程序进行完善，以读操作为例，得到如下代码：

```
29  static ssize_t BlockFIFODev_read(struct file *file, char __user *buf, size_t count, loff_t *ppos) {  //读取最大值
30      int readLen;
31      if (kfifo_is_empty(&FIFO_BUFFER)) {  //当缓冲区无数据
32          if (file->f_flags & O_NONBLOCK) {  //判断设备是阻塞还是非阻塞模式
33              return -EAGAIN;
34          }
35          //使读进程等待并设置唤醒条件
36          wait_event_interruptible(ReadQueue, !kfifo_is_empty(&FIFO_BUFFER));
37      }
38      //当缓冲区有数据
39      if (kfifo_to_user(&FIFO_BUFFER, buf, count, &readLen) == 0) {
40          printk(KERN_EMERG "Device Read %d bytes. Used buffer size: %d.\n", readLen, kfifo_len(&FIFO_BUFFER));  //调试输出信息
41      }
42      if (!kfifo_is_full(&FIFO_BUFFER)) {  //缓冲区有空间，唤醒写队列中进程
43          wake_up_interruptible(&WriteQueue);
44      }
45
46      return readLen;
47  }
```

图 2-2 参考驱动程序读操作代码图

对该参考驱动程序进行测试，首先使用两个进程进行读操作，然后再使用一个进程完成写操作。此时按照预期得到的结果应该是测试程序被挂起，一直等待缓冲区被填充数据或中断服务的到来。但在实际测试中，得到的结果却是程序正

常完成执行过程，根据 dmesg 命令得到的内核空间输出信息显示，kfifo 缓冲区发生溢出，导致数据不能被正常读写，可能造成读写空间的混乱。具体执行结果如下图所示：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver$ sudo ./test
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver$ sudo dmesg -c
[16182.092957] Device Open.
[16183.093620] Device Write 1 bytes. Used buffer size: 1.
[16183.093746] Device Read 1 bytes. Used buffer size: 0.
[16183.093747] Device Read 1 bytes. Used buffer size: -1.
[16183.093924] Device Release.
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver$
```

图 2-3 参考驱动程序测试结果图

观察到，输出信中包含 kfifo 缓冲区被占用的大小，当执行第二个读操作时，导致该缓冲区被占用大小溢出变为-1（实际是队尾超过队首）。

为完成对上述问题产生原因的剖析，需对以下两点进行分析：第一，kfifo 缓冲区结构是如何维护的；第二，wait_event_interruptible()与 wake_up_interruptible()函数是如何运作的。

2.3.2 kfifo 结构体与内存屏障

kfifo 缓冲区结构体如下，各结构体内的变量及作用已进行注释：

```
struct __kfifo {
    unsigned int    in;     /* 指向buffer中的队首(in % size)即(in & mask) */
    unsigned int    out;    /* 指向buffer中的队尾(out % size)即(out & mask) */
    unsigned int    mask;   /* 帮助完成取模操作，值为(size - 1) */
    unsigned int    esize;  /* 缓冲区一个单元的大小 */
    void            *data;  /* 用于存放数据的缓存 */
};
```

当对该结构体进行初始化时，会首先判断用户申请的大小是否为 2 的幂次，当不是时对该数进行上取 2 的幂次操作，即申请的 kfifo 缓冲区大小永远为 2 的幂次。使用 2 的幂次的理由为，由于 kfifo 维护的是一个内核循环队列空间，因此存在“取模运算”，而日常的取模运算时间开销较大，因此应该避免该操作。而当缓冲区大小为 2 的幂次时，对数 a 的取模操作可以转换为对 a 的按位与操作，即将“kfifo->in % kfifo->size”操作转换为“kfifo->in & (kfifo->size - 1)”操作即“kfifo->in & kfifo->mask”操作，以此提高运行效率。

测试过程中发现，kfifo 结构体缓冲区在并发访问时并没有加锁行为，这是由于采用了“内存屏障”。由于编译器在对代码进行编译的过程中会对部分代码进行优化，因此可能导致指令顺序的重排。然而，内核同步必须避免指令的重排，因此使用内存屏障来保证指令操作的原顺序性。kfifo 结构体缓冲区相应的读写操作函数便使用了该方法来实现“并发无锁”操作。

首先对内存屏蔽所需的函数进行说明，其具体功能如下表所示：

表 2-1 内存屏蔽函数功能表

函数	功能
smp_rmb()	读访问内存屏障：它保证在屏障（调用）之后的任何读操作在执行之前，屏障之前的所有读操作都已经完成
smp_wmb()	写访问内存屏障：它保证在屏障之后的任何写操作在执行之前，屏障之前的所有写操作都已经完成
smp_mb()	读写访问内存屏障：它保证在屏障（调用）之后的任何读写操作在执行之前，屏障之前的所有读写操作都已经完成

接着对 Linux 内核中 kfifo.c 文件中 kfifo 结构体读写有关的操作函数进行分析，来说明内存屏障在实现 kfifo 并发无锁中的作用。以读操作为例进行说明，内核代码如下图所示：

```
static void kfifo_copy_out (struct __kfifo *fifo, void *dst, unsigned int len, unsigned int off) {
    unsigned int size = fifo->mask + 1, esize = fifo->esize, l;

    off &= fifo->mask;
    if (esize != 1) {
        off *= esize; size *= esize; len *= esize;
    }
    l = min(len, size - off);

    memcpy(dst, fifo->data + off, l);
    memcpy(dst + l, fifo->data, len - l);
    /*
     * make sure that the data is copied before incrementing the fifo->out index counter
     */
    smp_wmb();
}
```

图 2-4 kfifo 读操作内核代码图

当使用 kfifo 的读操作函数时，会最终调用执行函数 kfifo_copy_out()完成对数据的读取。由于不同类型的缓冲区一个单元的大小各不相同，因此该函数首先根据 esize 单元大小值计算各参数对应的真实字节值，以备在 memcpy()函数中直接使用。接着计算 l 为距 buffer 尾值与 buffer 长度间的较小值，来方便执行复制操作，避免溢出的发生。两个 memcpy()函数先从 kfifo->out 到 buffer 尾复制数据，然后再从 buffer 首复制剩余的数据。最后，最关键的是调用写访问内存屏蔽函数 smp_wmb()来确保在此之前已经完成了所有写操作，即完成了数据从 kfifo 写入指定 dst 的操作。若没有该内存屏障，则可能先执行增加 out 的操作而导致数据没有读完却开始写操作最终发生数据损坏；同时当两个读进程同时发起调用时，由于内存屏障的存在，只有一个进程可以完成对数据的读取，否则若另一进程也开始读取数据则会导致屏障后仍存在写操作（从 kfifo 到 dst），不满足屏障条件，因此内存屏障的使用最终达到了并发无锁的目的。

2.3.3 wait_event_interruptible()与 wake_up_interruptible()函数

wait_event_interruptible()与 wake_up_interruptible()函数是 wait.h 中的宏定义函数。当调用函数 wait_event_interruptible()时，会将当前进程加入指定的队列中等待，同时设置 condition 唤醒条件。当调用函数 wake_up_interruptible()时，会遍历等待队列中的进程并对每个进程的唤醒条件进行判断，当唤醒条件满足时则将该进程唤醒。

现在，回头分析参考驱动程序发生溢出的原因，主要是由于当缓冲区中不存在数据时，对读进程进行挂起并设置唤醒条件为缓冲区存在数据。而当写进程向缓冲区中写入数据后，由于调用了 wake_up_interruptible()函数，此时遍历所有挂起的读进程，其判断条件均为缓冲区存在数据，而由于 kfifo 结构没有对同时判断缓冲区状态进行限制，因此导致此时队列中所有读进程都被唤醒。尽管 kfifo 结构体对读写进行了并发控制，但是没有对缓冲区状态判断进行并发控制（一般情况下也不应对此进行限制），从而导致了 out 值被溢出更改，最终超过了 in 值而导致数据区的溢出。

2.3.4 修正的阻塞与非阻塞内核驱动程序

根据上述对参考驱动程序出现的缓冲区访问溢出问题的分析，以及对内核循环队列缓冲区和内核队列管理函数的深入理解，在原有程序的基础上对其进行更改，以避免上述问题的产生。

第一，为解决判断缓冲区空与满状态时仅有单一进程对其进行访问，而避免由于判断条件一致但实际操作空间不够而导致的溢出问题，对缓冲区状态条件判断加锁，记为“条件判断锁”。同时，由于阻塞和非阻塞访问方式在申请锁时存在差异，因此需要分别加锁。

第二，为解决唤醒时由于所有挂起进程均满足缓冲区状态条件而导致进程全部被唤醒但实际操作空间不满足而导致的溢出问题，对唤醒条件进行更改，添加一个对缓冲区读或写资源的占用锁，记为“读锁”或“写锁”。

第三，为保证非挂起进程在执行完缓冲区状态判断后对缓冲区读独占或写独占（但读和写可同时进行），需要对其进行加锁。

第四，调用 kfifo 读写函数执行读写操作。

第五，当缓冲区符和读或写操作条件时，唤醒读或写进程。

根据以上五点内容对原参考驱动程序进行修改，得到修正的阻塞与非阻塞内核驱动程序代码如下图所示，以读操作为例：

```c
static ssize_t FIFODev_read(struct file *file, char __user *buf, size_t count, loff_t *ppos) {  //读取最大值
    int readLen;
    //保证读进程只有一个可以判断kfifo缓冲区状态，防止多个进程同时判断非空而导致不被挂起以致kfifo溢出
    if (file->f_flags & O_NONBLOCK) {  //阻塞方式与非阻塞方式的上锁方式不同
        if (mutex_trylock(&mutex_r) != 1) {  //非阻塞方式上锁失败，直接返回
            printk(KERN_EMERG "[INFO]: Nonblock try lock failed.\n");
            return -EAGAIN;
        }
    } else mutex_lock(&mutex_r);  //阻塞方式以阻塞方式上锁
    //单一的读进程判断是否要挂起
    if (kfifo_is_empty(&FIFO_BUFFER)) {  //当缓冲区无数据
        mutex_unlock(&mutex_r);  //释放判断读条件的锁
        if (file->f_flags & O_NONBLOCK) {  //非阻塞访问模式由于无数据不能读而直接返回
            printk(KERN_EMERG "[INFO]: Read, but buffer is empty.\n");
            return -EAGAIN;
        }
        //阻塞模式挂起，唤醒条件为kfifo有数据且获取到访问该缓冲区的锁，这样避免了仅判断kfifo状态而导致的过多进程被唤醒一致读溢出的发生
        wait_event_interruptible(ReadQueue, ((!kfifo_is_empty(&FIFO_BUFFER)) && (mutex_trylock(&can_read) == 1)));
    } else mutex_unlock(&mutex_r);  //释放判断读条件的锁
    //当缓冲区有数据时进行读操作
    if (!mutex_is_locked(&can_read)) mutex_lock(&can_read);  //当不是从队列唤醒的时，首先申请缓冲区访问锁
    if (kfifo_to_user(&FIFO_BUFFER, buf, count, &readLen) == 0) {
        printk(KERN_EMERG "[INFO]: Device Read %d bytes. Used buffer size: %d.\n", readLen, kfifo_len(&FIFO_BUFFER));  //调试输出信息
    }
    mutex_unlock(&can_read);  //释放占用的缓冲区
    if (!kfifo_is_full(&FIFO_BUFFER)) {  //缓冲区有空间，唤醒写队列中进程
        wake_up_interruptible(&WriteQueue);
    }

    return readLen;
}
```

图 2-5 修正的驱动程序读操作代码图

2.4 运行与测试

2.4.1 驱动程序的编译安装与卸载

使用 make 完成对驱动的编译，编写 Makefile 文件如下，由于笔者的实验环境为 WSL2 Ubuntu 18.04，因此 Makefile 执行的跳转路径与 VMware 虚拟机上的路径存在差异：

```
ifneq ($(KERNELRELEASE),)
obj-m := BlockFIFODev.o
else
KDIR := /usr/src/linux-headers-$(shell uname -r)
all:
    make -w -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
endif
```

在驱动文件所在目录下执行该 Makefile，此时会跳转进入指定目录执行该目录下的 Makefile 完成对驱动程序的构建，如下图所示：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ make
make -w -C /usr/src/linux-headers-4.19.128-microsoft-standard M=/mnt/e/OS课设/driver3 modules
make[1]: Entering directory '/home/lyg/WSL2-Linux-Kernel-4.19.128-microsoft-standard'
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /mnt/e/OS课设/driver3/BlockFIFODev.mod.o
  LD [M]  /mnt/e/OS课设/driver3/BlockFIFODev.ko
make[1]: Leaving directory '/home/lyg/WSL2-Linux-Kernel-4.19.128-microsoft-standard'
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-6 驱动程序编译结果图

此时该目录下已产生驱动装载相关文件，如下图：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ ls
BlockFIFODev.c   BlockFIFODev.mod.c  BlockFIFODev.o  Module.symvers  test    test1
BlockFIFODev.ko  BlockFIFODev.mod.o  Makefile        modules.order   test.c  test1.c
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-7 驱动相关文件图

为该驱动创建一个与该驱动相关联的设备文件。使用 mknod 命令在/dev 目录下创建一个名为 BFDev 的文件，其设备类型为字符型设备，主设备号与该驱动对应的主设备号相同，次设备号指定为 0，如下：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo mknod /dev/BFDev c 369 0
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-8 创建驱动关联设备文件图

装载内核驱动和卸载内核驱动如下图所示：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo insmod BlockFIFODev.ko
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo rmmod BlockFIFODev
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo dmesg -c | grep FIFO
[  263.741931] FIFO-Device Initialized.
[  270.721646] FIFO-Device Removed.
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-9 加载与卸载驱动结果图

2.4.2 测试样例的构造与说明

对阻塞与非阻塞访问方式分别进行测试，由于实际运行过程中可能存在 read()函数被挂起或 write()函数被挂起两种情况，因此对这两种情况分别构造样例进行检测。样例说明如下表 2-2 所示：

表 2-2 测试样例说明表

访问方式	测试文件	作用
阻塞	HangOnRead_Block.c	测试 read()挂起后被唤醒时是否产生溢出
	HangOnWrite_Block.c	测试 write()挂起后被唤醒时是否产生溢出
非阻塞	HangOnRead_NonBlock.c	测试 read()得不到锁时是否直接返回
	HangOnWrite_NonBlock.c	测试 write()得不到锁时是否直接返回

对样例的测试、预期结果以及具体结果分析，见 2.4.3 节。

2.4.3 测试与结果

HangOnRead_Block.c：

对该测试样例进行测试，得到如下结果：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo ./HangOnRead_Block
^C
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo dmesg -c | grep INFO
[ 6337.617588] [INFO]: Device Write 1 bytes. Used buffer size: 1.
[ 6337.617711] [INFO]: Device Read 1 bytes. Used buffer size: 0.
[ 6339.773734] [INFO]: Device Read 0 bytes. Used buffer size: 0.
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-10 HangOnRead_Block 测试结果图

该程序创建了三个进程，write()函数在执行前等待一秒以保证两个 read()函数均被挂起。当执行完 write()函数后由于仅有一个 read()进程可以获取到读访问资源，因此仅有一个 read()进程被唤醒并读取数据。另一个进程由于唤醒失败而继续在等待队列中等待，因此出现了需要中断程序的操作，当程序中断后，由于缓冲区此时已经为空，因此不能读取到任何资源，而后 read()函数返回。

与预期结果一致。

HangOnRead_NonBlock.c：

对该测试样例进行测试，得到结果如下：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo ./HangOnRead_NonBlock
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo dmesg -c | grep INFO
[ 6725.342144] [INFO]: Read, but buffer is empty.
[ 6725.342206] [INFO]: Read, but buffer is empty.
[ 6726.342685] [INFO]: Device Write 1 bytes. Used buffer size: 1.
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-11 HangOnRead_NonBlock 测试结果图

该程序与其阻塞模式下的测试程序一致，仅改变其访问方式为非阻塞模式。由于 write()函数等待一秒后再执行，因此当两个 read()进程以非阻塞方式尝试读取数据时，缓冲区中不存在数据，故会提示缓冲区为空并立刻返回而非执行挂起。随后进行 write()操作时由于缓冲区为空故可直接写入。

与预期结果一致。

HangOnWrite_Block.c：

对该测试样例进行测试，得到如下结果：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo ./HangOnWrite_Block
^C
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo dmesg -c | grep INFO
[ 8301.930914] [INFO]: Device Write 63 bytes. Used buffer size: 64.
[ 8302.931576] [INFO]: Device Read 64 bytes. Used buffer size: 0.
[ 8302.931688] [INFO]: Device Write 64 bytes. Used buffer size: 64.
[ 8305.035477] [INFO]: Device Write 0 bytes. Used buffer size: 64.
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-12 HangOnWrite_Block 测试结果图

该测试程序首先调用一次 write()函数将缓冲区填充满，然后创建三个进程。read()函数在执行前先等待一秒，以保证两个 write()进程已被挂起。当执行 read()函数后缓冲区中有剩余空间，因此 write()进程被唤醒，同时又由于需要获得写缓冲区锁，因此最终仅有一个进程被唤醒，另一个进程继续等待。因此观察到该程序需要被中断，否则 write()进程持续被挂起无法结束；当进程中断后由于缓冲区

状态为满，因此无法进行 write()操作。另外第一个 write()函数虽然要求写入 64 字节值，但是由于在测试非阻塞方式 read()函数挂起时，在缓冲区中留有 1 字节数据未读出，因此最终只成功写入 63 字节的数据。

与预期结果一致。

HangOnWrite_NonBlock.c：

对该测试样例进行测试，得到如下结果：

```
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo ./HangOnWrite_NonBlock
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$ sudo dmesg -c | grep INFO
[ 8946.024754] [INFO]: Write, but buffer is full.
[ 8946.024860] [INFO]: Write, but buffer is full.
[ 8946.024923] [INFO]: Write, but buffer is full.
[ 8947.025056] [INFO]: Device Read 64 bytes. Used buffer size: 0.
lyg@DESKTOP-C3PNG35:/mnt/e/OS课设/driver3$
```

图 2-13 HangOnWrite_NonBlock 测试结果图

该测试程序与其在阻塞访问方式下的测试程序一致，仅更改访问方式为非阻塞访问。在上一个测试结束后，缓冲区中存在 64 字节的数据，即缓冲区状态为满，此时不能写入数据。而非阻塞访问方式当不能得到锁或缓冲区状态不符和时直接返回，因此先执行的三个 write()函数均无法对缓冲区进行写操作，故直接退出。当执行 read()函数时获取到资源并直接从缓冲区中读出所有数据。

与预期结果一致。

2.5 参考资料

[1]. https://www.kernel.org/doc/htmldocs/kernel-api/kfifo.html

[2]. https://www.kernel.org/doc/htmldocs/kernel-locking/apiref-mutex.html

[3]. https://www.linuxidc.com/Linux/2016-12/137936.htm

[4]. https://github.com/torvalds/linux/blob/master/include/linux/kfifo.h

[5]. https://github.com/torvalds/linux/blob/master/lib/kfifo.c