

本 科 生 课 程 设 计 报 告

课 程： 操作系统课程设计

题 目： 任务切换机制与设备阻塞工作机制研究

院 系： 网络空间安全学院

专业班级： 信安1802班

学 号：

姓 名： 李文轩

2021 年 03 月 26日

目 录

[1 实验一 任务切换机制 1](#_Toc67422133)

[1.1 实验概述 1](#_Toc67422134)

[1.2 实验设计思路 1](#_Toc67422135)

[1.3 实验程序的难点或核心技术分析 4](#_Toc67422136)

[1.4 运行和测试过程 8](#_Toc67422137)

[1.5 实验心得和建议 9](#_Toc67422138)

[1.6 学习和编程实现参考网址 10](#_Toc67422139)

[2 实验二 设备阻塞工作机制 11](#_Toc67422140)

[2.1 实验概述 11](#_Toc67422141)

[2.2 实验设计思路 11](#_Toc67422142)

[2.3 实验程序的难点或核心技术分析 13](#_Toc67422143)

[2.4 运行和测试过程 14](#_Toc67422144)

[2.5 实验心得和建议 16](#_Toc67422145)

[2.6 学习和编程实现参考网址 17](#_Toc67422146)

# 实验一 任务切换机制

## 实验概述

1.1.1 课设任务

理解和重现/实现一个基于极简页式内存机制的保护模式程序。

1.1.2 课设目的：

（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；

（4）熟悉和理解 Linux 编程环境。

1.1.3 课设内容

（一）阅读和理解X86保护模式的系列程序（pmtest1~pmtest7）；

（二）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,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：

① 每个任务使用各自对应的页表；② 每个任务简单地输出A或B。

## 实验设计思路

实验一的主体任务为：由实模式进入保护模式，利用时钟中断完成两个任务的切换，同时内存采用简单的页式存储方式。

本实验借助TSS（任务状态段）完成状态转换，为每个任务分别定义其LDT描述符、数据段、代码段、堆栈段，以上数据结构均在GDT描述符有定义。由于任务完成实模式与保护模式之间的切换，故每个数据结构需有对应的选择子，以便在保护模式下也可完成寻址。

保护模式下的段寄存器储存的是“段描述符”，其中包括段基址、段界限、属性等对一个段的定义，如图1.1所示，为常用代码段、数据段的格式。而图1.2所示的选择子将指向某一段描述符，索引表示段描述符在GDT或LDT中的位置，TI位指示是从GDT还是LDT中寻址，RPL表示特权级。

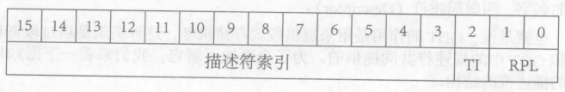
 

图1.1 段描述符 图1.2 选择子

最终代码中定义形式如图1.3所示，“Descriptor”为在“pm.inc”文件中定义的宏，把三个参数按照图1.1的段描述符形式组合，目前段基址留空，在代码中再填入实际段基址，如图1.4所示。图1.5所示的选择子采用“LABEL\_XXX - $$”的形式，“$$”指当前段基址位置，由于单个段描述符长度为8字节，故差值的低三位将为0，而高位代表段描述符相对GDT的偏移，恰好为图1.2中所指特权级为0的的GDT定义形式。

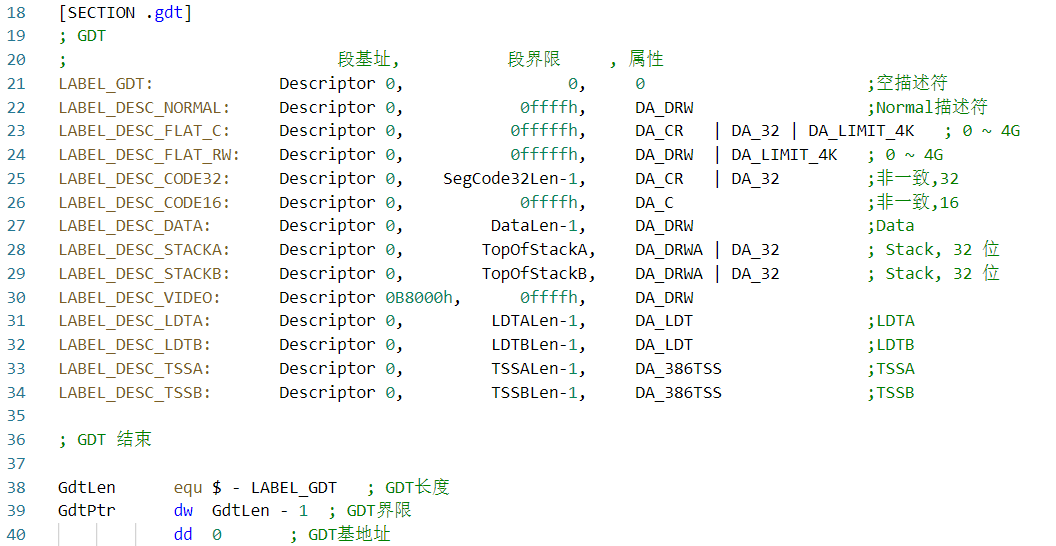


图1.3 GDT定义

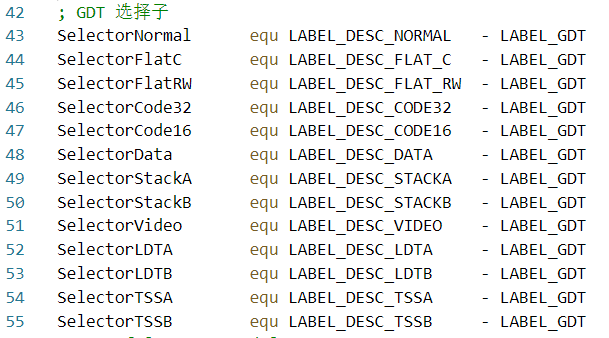
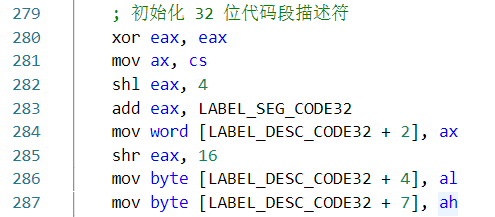
 

图1.4 GDT选择子定义 图1.5 填入段基址

在定义LDT中选择子时，在后面加上“SA\_TIL”即可完成TI位至1，示例如图1.6所示。

进入保护模式主要有两个关键点：（1）打开A20地址线，扩大寻址范围；（2）改变cr0标志位，告知系统处于保护模式下。打开A20可以通过操作端口92h实现。如图1.7所示为CR0结构，CR0中的第0位PE区分实模式和保护模式，将其置1进入保护模式。

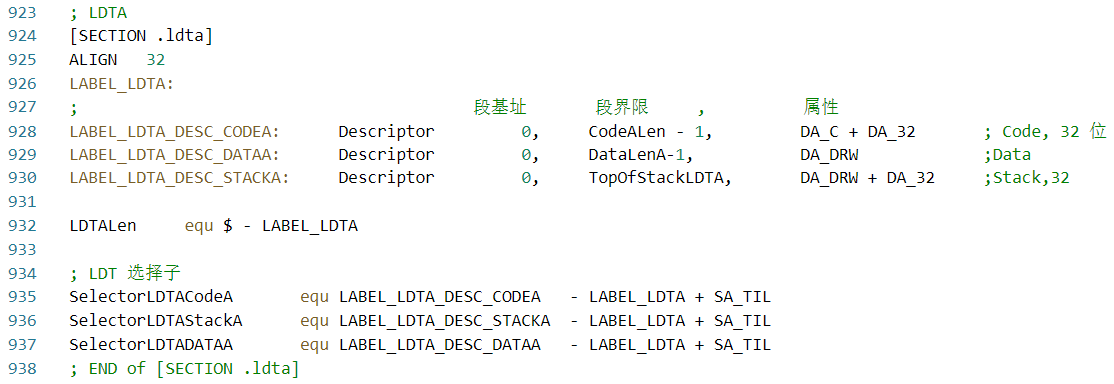


图1.6 LDT\_A定义

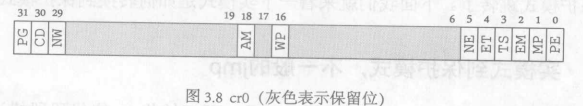


图1.7 CR0结构（灰色表示保留位）

由于涉及任务切换功能，故需一个结构储存任务切换前后的现场（寄存器、堆栈、特权级等），此结构为TSS，图示如1.8，其保存三个特权级的堆栈及相应的段、寄存器信息。在代码中切换任务使用“JMP TSS”指令，其原理为：系统初始将某任务一TSS加载到寄存器ltr中，当使用“JMP TSS”指令时，将把当前现场保存到ltr指向的tss结构中，再将jmp指向的新tss中寄存器状态加载入内存，实现任务切换。代码实现如图1.9所示，其中0级堆栈与ESP、SS对应的堆栈特权级可以不同（本任务未涉及）。而TSS结构中提及的寄存器CR3如图1.10所示。

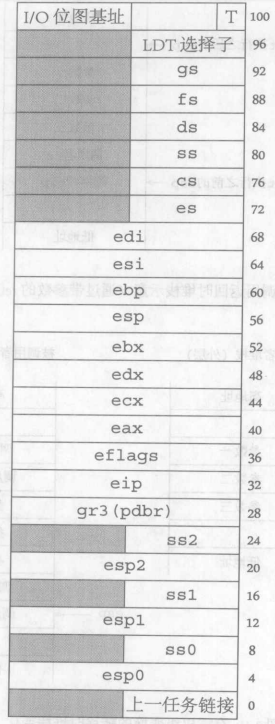
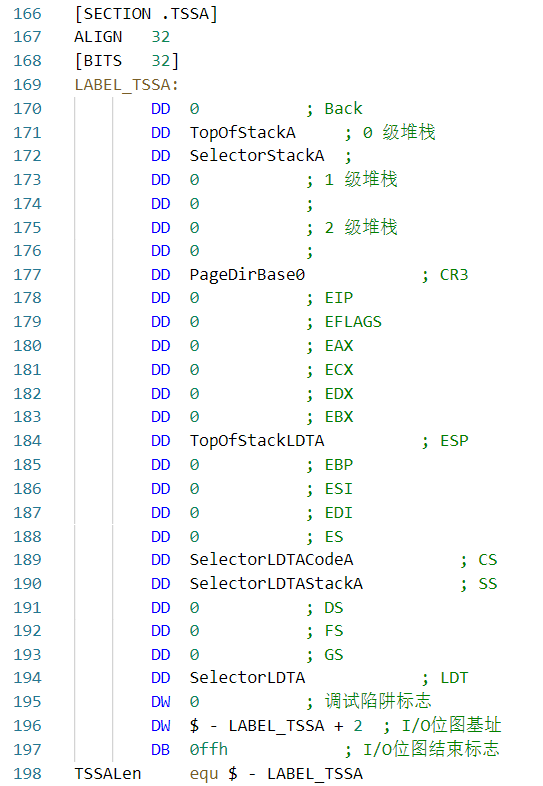
 

图1.8 TSS结构示意 图1.9 TSS\_A定义

CR3是储存页目录信息的寄存器（PDBR），其高20位指向页目录表首地址的高20位，而页目录表的低12位因对齐为0。TSS通过储存CR3信息，完成在页式存储系统中的任务切换。

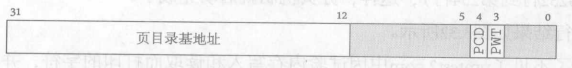


图1.10 CR3结构示意

为能够充分利用内存，发挥页式存储系统的优势，分配页表时需知道内存的可利用空间，避免页目录等分配过多，浪费不必要空间，其中一种方法是利用int 15h中断，得到关于地址范围描述符的信息。代码实现可见图1.14。

由于我们利用时钟中断作为节点进行任务切换，而时钟中断是可屏蔽中断，保护模式下又占据了默认的向量号，所以要通过设置8529A和建立IDT中断表以正确开启时钟中断。将设置8529A的操作写入函数“Init8529A”，如图1.11所示，“io\_delay”是简单的延时函数。设立IDT表如图1.12所示，除20h填入时钟中断外，其余全填入简单函数“SpuriousHandler”，内容为在屏幕右上角显示红色“！”。

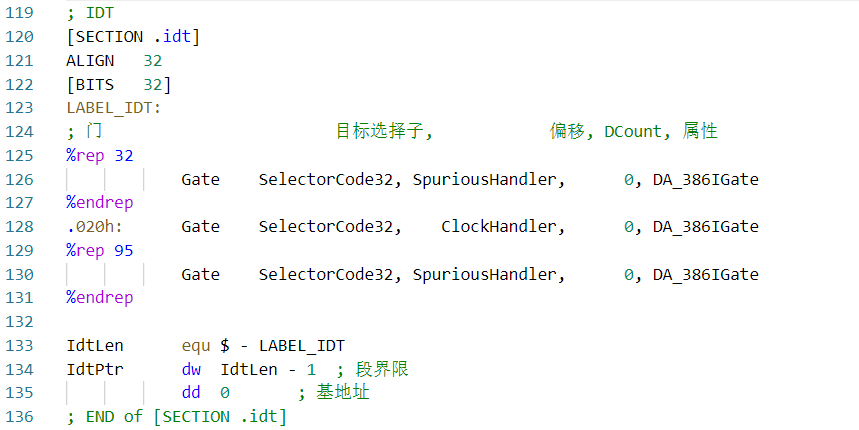


图1.11 操作8529A 图1.12 设立IDT表

故在使用指令“sti”开启中断后，每隔10ms将启动时钟中断，执行“ClockHandler”中的内容，具体内容见1.3节技术分析。

以上为任务一及子功能实现所需要的原理以及简单代码介绍。

## 实验程序的难点或核心技术分析

在此节中，将以程序执行顺序为叙述顺序，简要介绍实现过程，重点介绍核心技术和程序难点。

首先如图1.13所示，完成对必要物理地址声明后，系统进入执行的第一条语句“org 0100h”，此句告知程序程序被载入内存的初始偏移地址，便于BIOS寻址。而后程序跳转至16位实模式代码段开始。

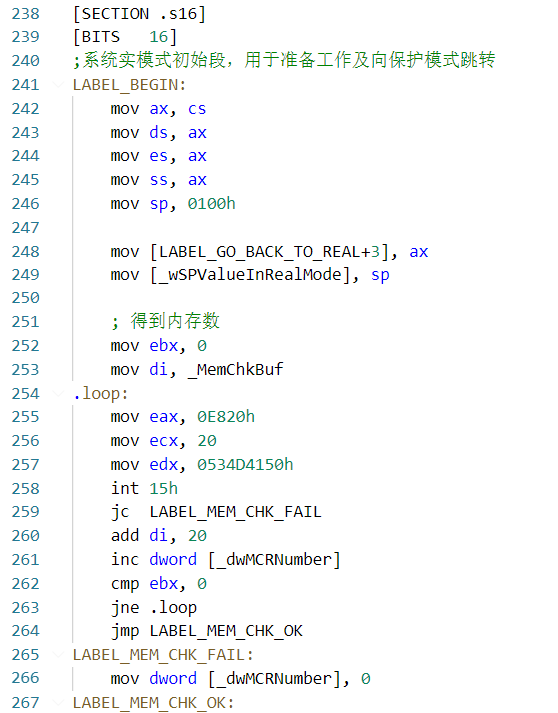
 

图1.13 程序初始化 图1.14 16位代码段信息

如图1.14所示，在16位实模式代码段中，首先完成相关寄存器的初始化，248~249行为最后程序由保护模式跳转回实模式做好准备（但由于任务的死循环性质，任务不会进行至返回实模式阶段，故相关内容忽略）。而后程序利用15号中断搜寻内存可用空间，结果存在变量“dwMCRNumber”中。

在16位代码段中，主要的重复工作为完善GDT中定义的段描述符如图1.5所示，其将GDT中省略的“段基址”分三部分填入段描述符。

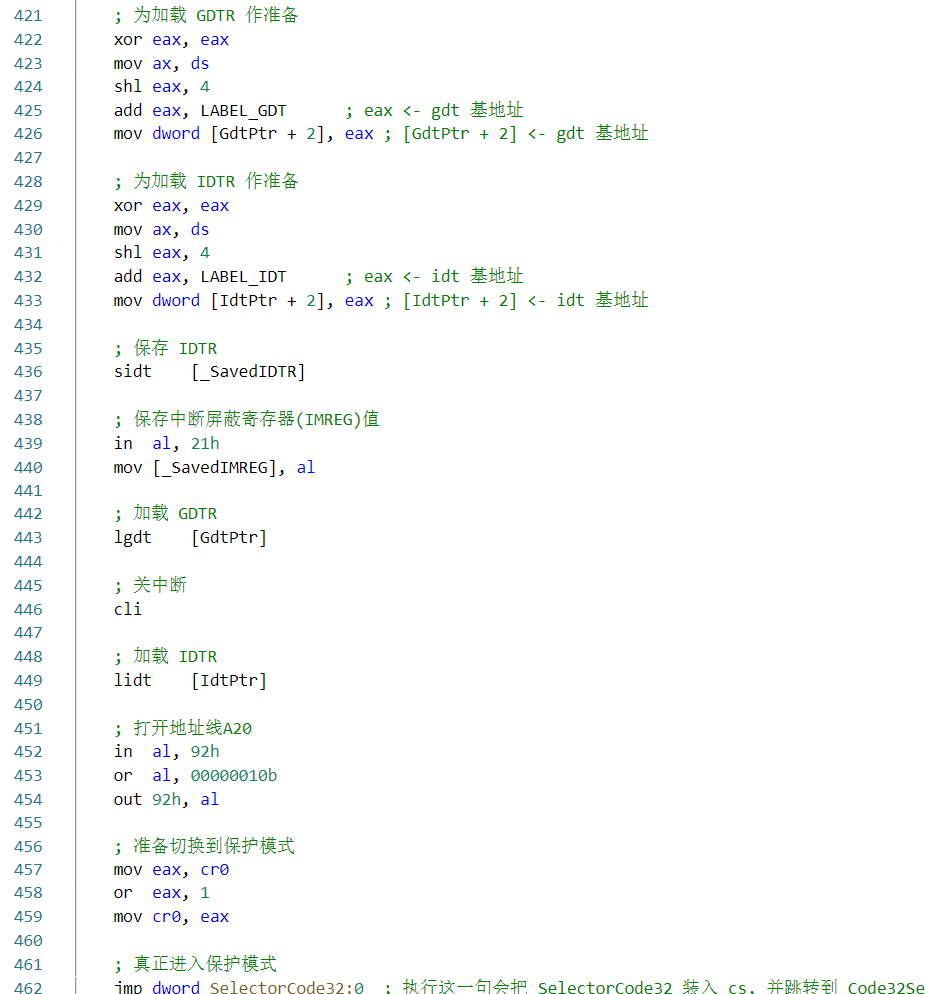
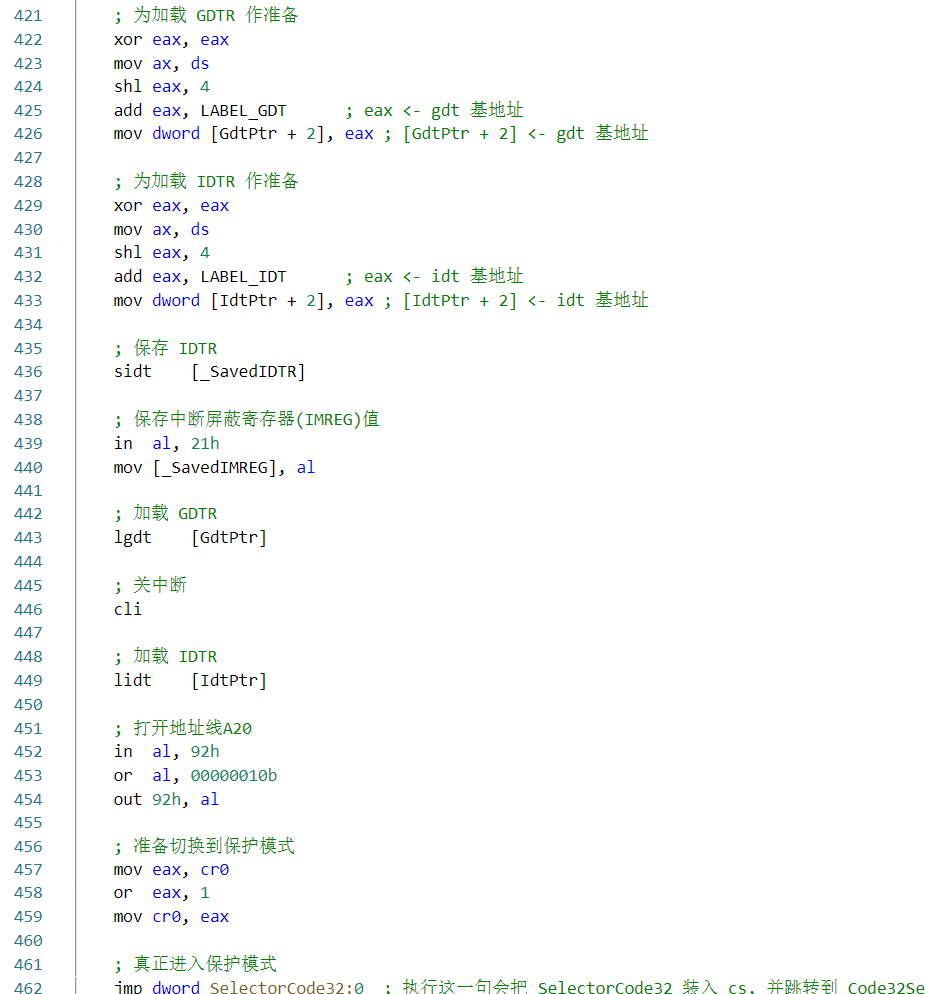
 

图1.15(1) 16位代码段信息(1) 图1.15(2) 16位代码段信息(2)

图1.15 进入保护模式前准备

在进入保护模式之前，如图1.15所示，需要将GDT和IDT加载到对应位置以供调用。由于实模式和保护模式下调用中断的方式不同，故需提前关闭中断以免出现错误。打开地址线A20并将CR0的PG位置1，完成准备后跳转到32位代码段中。

但此处需注意，16位与32位寻址方式不同，在16位段中利用32位长地址跳转，需在“jmp”和地址之间加“dword”声明，否则地址会被截断。

进入保护模式（code32）之后，首先将初始化段寄存器与8529A。而后检查系统空余内存并向屏幕输出内存信息，其中空内存大小被存入变量“dwMemSize”中。输出内存信息的函数可见附录代码。

为使分页存储运行达到预期目的，首先调用函数“PagingDemo”将taskA、taskB、显示函数复制到内存指定地址，即图1.13开头指定的地址，而后调用函数“SetupPaging”，将页目录及页表初始化，举例如图1.16所示。将页表的地址填入页目录中，将内存地址填入页表中。

其中由于内存需分配给两个任务，在图1.16(2)中第666行计算页表数时，将原本的除数4M乘2。

而初始化完成后，还需将taskA、B对应函数填入指定页表下如图1.16(2)所示，使之后调用该线性地址时，系统能顺利从线性地址对应的物理地址找到该函数。

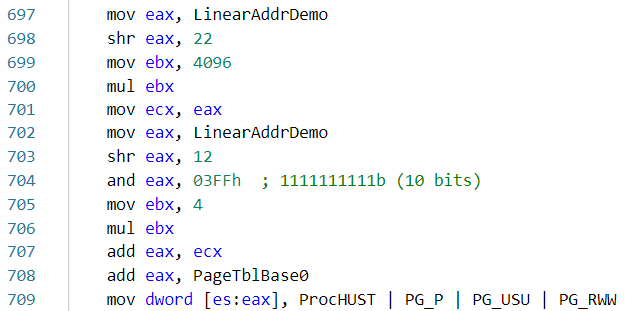


图1.16(1) 函数SetupPaging(1) 图1.16(2) 函数SetupPaging(2)

图1.16 函数SetupPaging

页式储存初始化完毕后，将TSS与LDT加载入相应寄存器中如图1.17所示。随后利用堆栈跳入taskA。

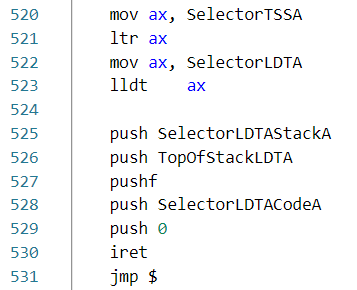
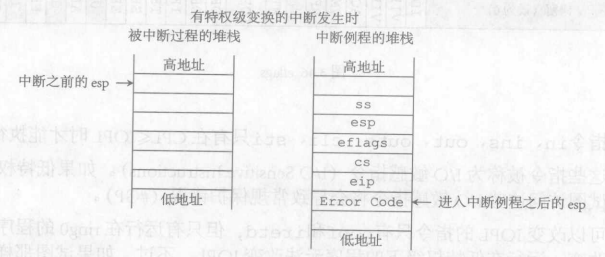
 

图1.17 跳入taskA 图1.18 中断发生时堆栈变化

利用堆栈进入taskA的原理参见图1.18的堆栈变化，将ss、esp、eflags、cs压入栈，eip值设0表示从程序开头开始，随后利用iret将堆栈内值弹入寄存器，程序流程切入taskA。而此前线性地址“LinearAddrDemo”与ltr和lldt的初值均填入taskA相关值。

如图1.19所示，taskA代码段的内容十分简单。SelectorVideo在图1.4中定义，指向显存，将其放入gs中为后续输出做准备。函数ShowProcDemo如图1.20所示，调用线性地址“LinearAddrDemo”上的函数，由于taskA和taskB对应不同的页目录，故调用“同一函数”应实际上对应不同函数，从而显示不同内容。

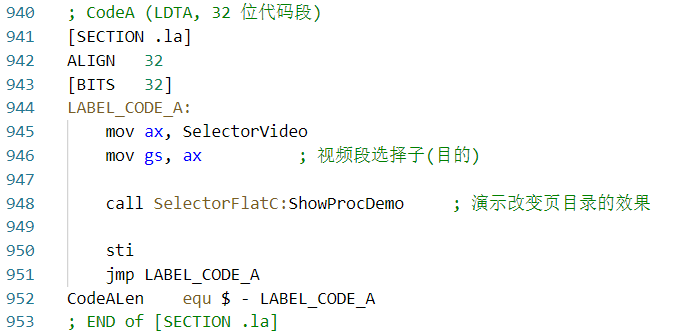
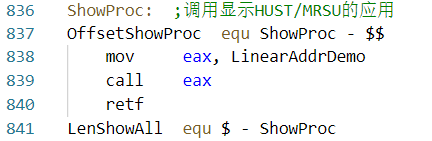
 

图1.19 taskA代码段 图1.20 show函数

可以看到taskA的结尾指令“jmp LABEL\_CODE\_A”重新跳到此段开头，无限执行该段指令，结合“jmp TSS”可实现交替显示某字符串的功能，在下文结合时钟中断中的内容详述。

在taskA利用“sti”开启时钟中断后，程序进入如图1.21所示的 ClockHandler内，首先保存现场并设置必要的寄存器，后发送EOI给8529A，以便能接受下一个中断。变量“Current”在SelectorData段中定义，初始值为0，每次时钟中断时，先将当前值反置，再根据其值判断下一次跳转去往taskA还是taskB。



图1.21 时钟中断内容

在系统第一次循环时， ClockHandler将运行到“jmp SelectorTSSB:0”处，保留当前现场，并跳入taskB中重复上述过程，而taskB中断时将重新开启一个ClockHandler，重新判断执行jmp指令，在第二个ClockHandler判断完执行到“jmp SelectorTSSA:0”后，将返回到图1.21所示560行位置继续执行，最后以“iret”返回taskA，再次重复直至中断跳入taskB。以上文字描述，可转换成示意图如图1.22所示。

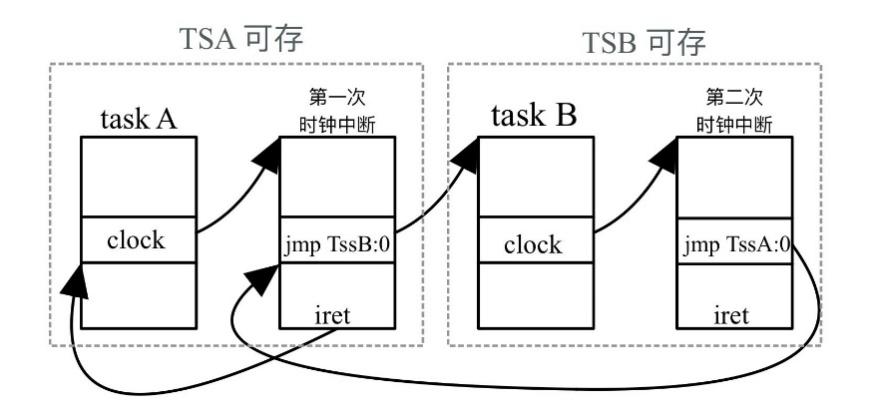


图1.22 第一次任务切换图示

其中ClockHandler必须保留现场。虽然使用tss做任务切换，理论上可以自动保护现场，但在时钟中断使用寄存器却没有保护的话，有可能在jmp TSS之前丢失掉原任务的信息，导致任务切换无法持续进行，且经实验验证必须保护现场。

## 运行和测试过程

将生成的文件pm.img，与配置文件bochsrc，头文件lib.inc，dos软盘freedos.img放到统一目录下。配置文件bochsrc内容如图1.23所示。使用如图1.24所示makefile编译文件“High.asm”，生成可执行文件“High.com”。

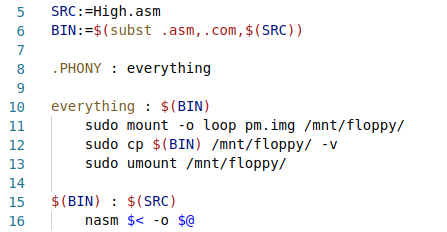
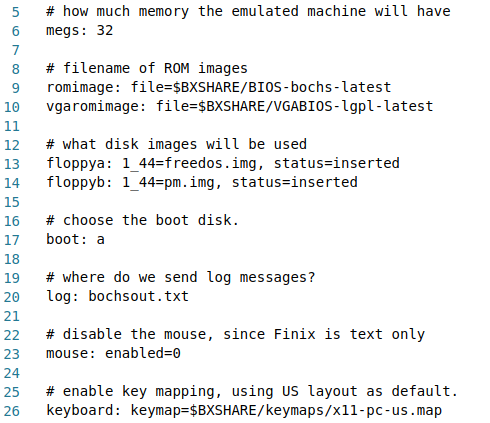


图1.23 配置文件bochsrc 图1.24 makefile内容

使用bochs命令启动虚拟机，输入指令“c”使虚拟机继续运行，启动dos系统，在b盘下执行“High.com”，操作如图1.25所示。

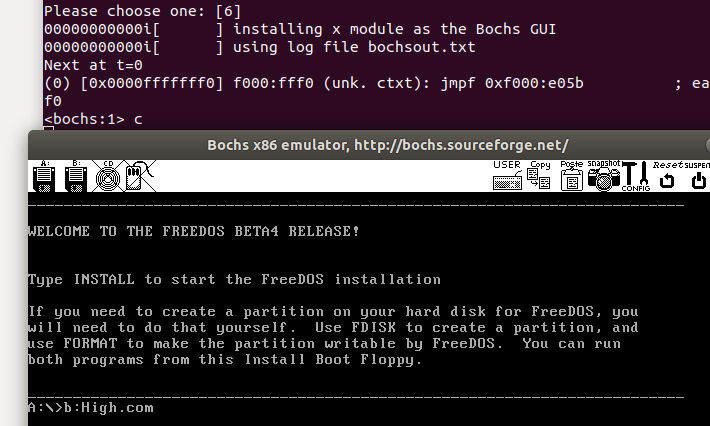


图1.25 启动bochs模拟器

执行效果如图1.26、图1.27所示，屏幕中部交替显示红色字符串“HUST”和“MRSU”，闪现效果可见测试视频。



图1.26 显示字符串“HUST”



图1.27 显示字符串“MRSU”

## 实验心得和建议

本次实验的难点主要在阅读参考文献，理解各模式、功能的实现原理，并从各有实现功能的文献中将所需功能组合起来，而最重要的是理解自己要做什么，需要实现的效果是什么，该如何安排函数使达到效果目的。

就整体课设所做的工作而言，用在任务一上的时间比任务二要多很多，一方面是因为阅读参考文献需要用大量时间，还有一方面是前期对任务目的不清楚。“任务切换”的效果是怎样的，需不需要实现特权级的切换等等。

另外参考文献前三章对于重点数据结构“TSS”的介绍和应用还是少了点，唯一应用是利用堆栈实现任务切换，也没有对“JMP TSS”这条指令的解释，可以说后期对TSS及语句的知识了解主要来自与网络。而知识缺乏可能导致实验过程中出现各种问题，花费更多的时间来调试试错，例如我的程序在后期整体配置都正确，但却只能从taskA跳至taskB，再也无法跳回去。

此处要真心感谢慕冬亮老师的耐心指导，帮助我仔细检查了程序，提出各种对程序bug的猜想，测试了几十分钟，最后发现“ClockHandler”中未保存eax、gs、ds的现场，这导致程序无法从jmp tssA中调回A，因为现场被破坏了。

如果老师能提前告知实验的效果如何（即实现两个字符串的交替闪现），提供一些对JMP TSS指令的参考资料，或是提醒学生从网上搜寻相关资料，我想任务一实验的完成应该会更顺利些。

对实验考核无更好的意见，当下形式可以较好的检测出学生是否仔细阅读参考文献，而时间与任务量也比较合适，没有充分的时间查找资料，但对资料了解充分的学生可以充分做完题目。

## 学习和编程实现参考网址

1. 于渊编著. Orange'S一个操作系统的实现. 北京：电子工业出版社, 2009.06.
2. https://blog.csdn.net/m0\_37329910/article/details/89944946

# 实验二 设备阻塞工作机制

## 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 课设目的

（1）理解操作系统对计算机系统的保护概念；

（2）理解CPU保护模式概念和操作系统保护模式的编写；

（3）理解CPU和操作系统对段机制和页机制的支持；

（4）理解页式内存管理概念和极简实现方案。

2.1.3 课设内容

（一）Linux下编写设备驱动程序：

（1）内设固定大小缓冲区BUFFER，读/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。

（二）编写不少于2个对设备进行读/写的测试应用程序：

观察缓冲区变化与读/写进程的阻塞/被唤醒的同步情况。

## 实验设计思路

任务要求，在系统内核设定固定大小的缓冲区，并实现读写不遗漏不重复，上学期的操作系统实验中有类似于此的实验内容，而此次课程设计相比上次的不同点在于，借用kfifo结构，生成可以自动维护读写的缓冲区。并借用函数“wait\_queue\_head\_t”实现阻塞状态下设备读写队列的维护。

任务用到的主要结构如下：

（1）miscdevice，如图2.1所示。可声明一类型为“misc”，主设备号为10的设备，其中“.minor”项表示次设备号；“.name”项表示设备在系统中名称，系统将在“/dev”目录下使用该名称建立一个设备文件；“.fops”项与另一数据结构“file\_operations”关联，指示该设备的所有操作。

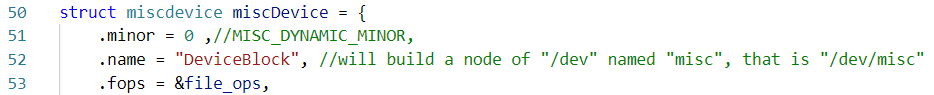


图2.1 数据结构miscdevice

（2）file\_operations，如图2.2所示，指示设备文件的操作函数。在此例中，设备共有open、release、read、write四个操作，分别对应一个函数。

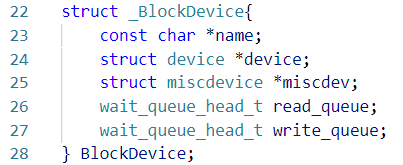
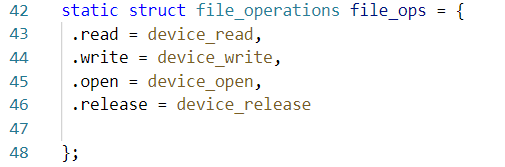


图2.2 数据结构file\_operations 图2.3 数据结构\_BlockDevice

（3）\_BlockDevice，如图2.3所示，声明一个设备单元，将一个misc设备与两个读写队列绑定在一起，便于维护。读写队列仅在阻塞情况下会用到。

（4）wait\_queue\_head\_t，如图2.3中所示，指示一个设备的等待队列，如在2.3中便定义了两个此类型队列分别用于读、写，当某一进程申请此设备而此设备正在被使用，则进程被挂起放入到相应目的的等待队列中，等待被唤醒。

（5）宏定义DEFINE\_KFIFO，代码如图2.4所示，其在Linux Kernel API库中的定义如图2.5所示，说明三个参数的意义分别为：“fifo结构的名称”、“fifo结构元素的类型”、“fifo结构的大小（必须为2的幂次）”。



图2.4 宏定义DEFINE\_KFIFO

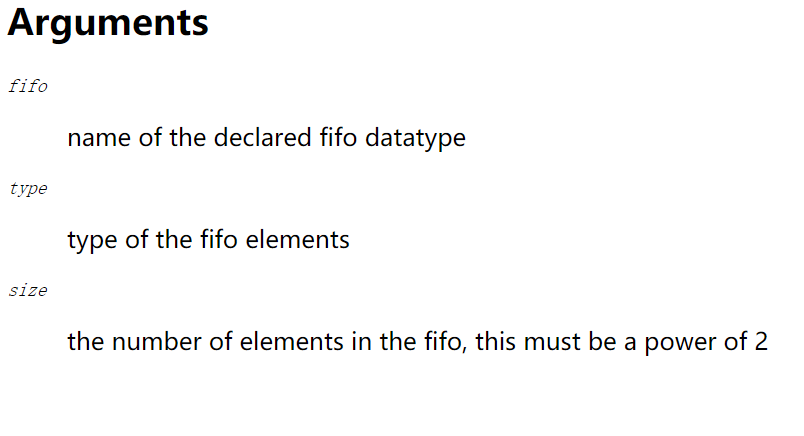


图2.5 DEFINE\_KFIFO的参数解释

以上为程序使用的基本数据结构定义，下面对所用到的主要函数进行简单解释。

（1）kfifo\_is\_empty、kfifo\_is\_full：判断缓冲区为空/满，判断成功则返回0；

（2）wait\_event\_interruptible：将某进程挂起到设定队列中，并设置挂起条件；

（3）wake\_up\_interruptible：将某一进程唤醒；

（4）misc\_register、misc\_deregister：注册与注销misc设备；

（5）init\_waitqueue\_head：初始化等待队列。

（6）kfifo\_to\_user、 kfifo\_from\_user：从用户空间中读、写数据

在4个设备操作函数之外，还有两个基本函数用于加载和退出操作所在模块，代码实现如图2.6所示。

其中初始化函数“lwx\_dev\_init”将使用设备注册并判断注册是否出错，初始化等待队列，使用“module\_init”运行初始化函数，配合命令将模块加载。

退出函数“lwx\_dev\_exit”将设备注销并输出提示信息，使用“module\_exit”运行退出函数，配合命令将模块卸载。

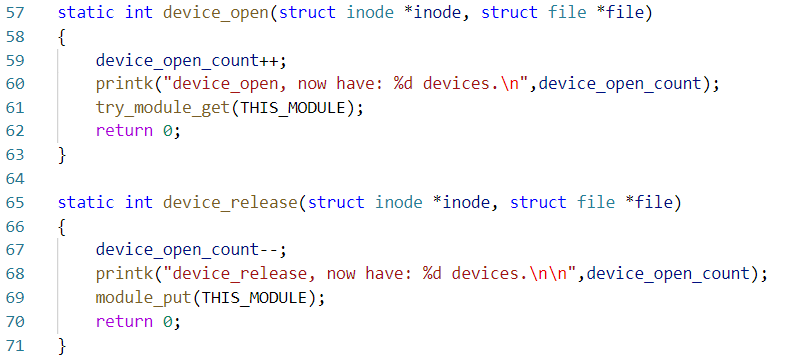


图2.6 初始化与注销模块 图2.7 open与realease函数

设备操作函数中“open”与“release”是较为通用，而与任务主题关系不大的函数，如图2.7所示，其主要作用为输出提示信息。其中device\_open\_count指示当前已申请设备的数量。

## 实验程序的难点或核心技术分析

实验程序的核心在于如何实现设备的读和写，设备读写时通过区分设备文件指针的某一标志位，判断调用状态为阻塞或非阻塞。

（一）设备写：

如图2.8所示“读”函数，首先对fifo缓冲区是否为空进行状态判断，若fifo为空，则判断调用状态为阻塞还是非阻塞。阻塞状态下将当前进程挂起到read等待队列中，非阻塞状态下则报错并返回。若fifo非空，则程序从fifo缓冲区中读取最长为len的数据放到用户空间缓冲区buffer中，并记录实际读取字节数。完成读取后，检查fifo是否已满，若不满则可从“写”等待队列中唤醒一进程。函数最后返回实际读取字节数。



图2.8 阻塞状态下的read函数

如图2.9所示“写”函数，与“读”函数结构类似，其先对缓冲区fifo状态是否为满判断。若fifo已满，阻塞状态下将当前写进程挂起，非阻塞状态则报错退。，若fifo未满，则可从buffer缓冲区中读取数据放置到fifo缓冲区中，记录实际写入字节数。写入完成后，判断fifo是否为空，若不空则可唤醒一“读”函数。函数最后返回此次实际写入字节数。

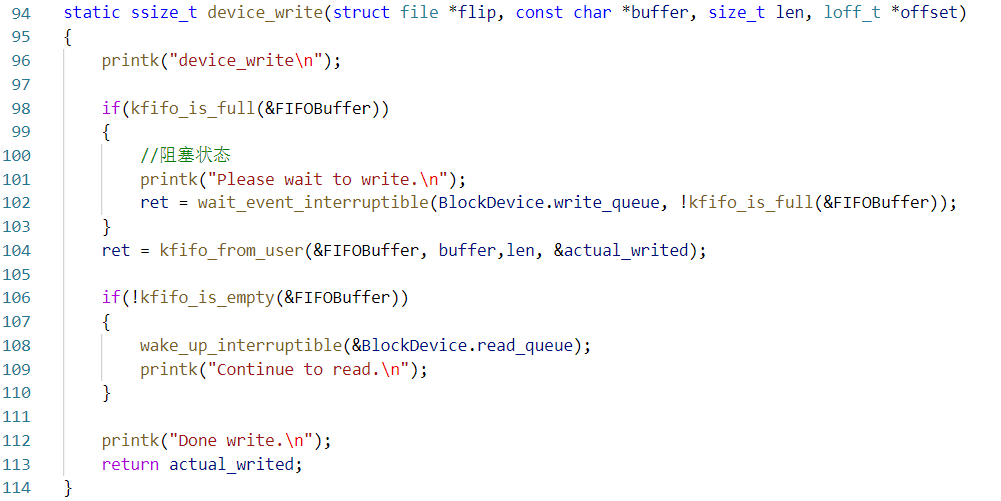


图2.9 阻塞状态下write函数

## 运行和测试过程

为阻塞与非阻塞状态程序分别编写了读写测试程序，在linux环境下测试，同时使用两个终端窗口分别运行读、写程序，观察运行结果。

（一）阻塞状态：

read函数如图2.12所示，write函数如图2.13所示。两函数前半部分相同，即打开该设备文件（申请设备），并报错。而后read函数设定长度为9的空缓冲区buf，储存read函数读出的数据，并向用户打印读取到的字节数及内容。write函数设定长度为10的数组a，使用write函数向内核空间写入，并返回成功写入的字节数。

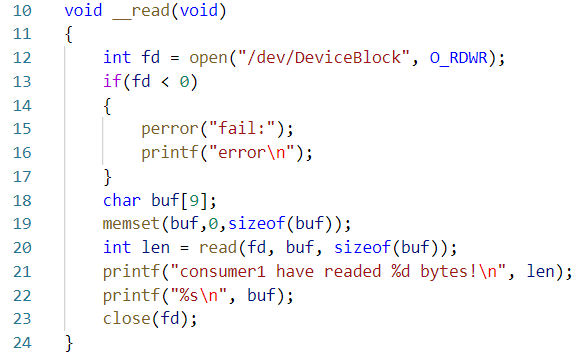
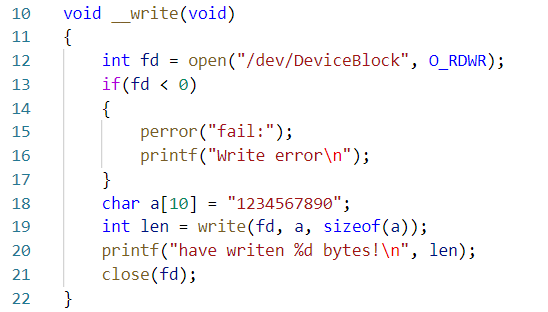
 

图2.12 阻塞状态read函数 图2.13 阻塞状态write函数

主函数内容为死循环，如图2.14所示， 不断调用read / write函数，并休眠一定范围内的随机时间。

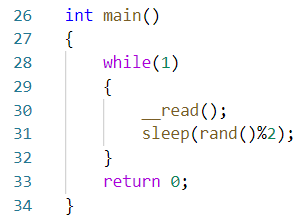


图2.14 函数主体

为使实验效果明显，模块设定的内核缓冲区大小BUFFER\_SIZE为16，同时在两终端分别启动“write.out”与“read.out”，在运行一段时间后终止，结果输出如图2.15所示。仔细对比两窗口可发现，read与write函数吞吐字节数相等，且read函数做到按顺序、无遗漏的读入。

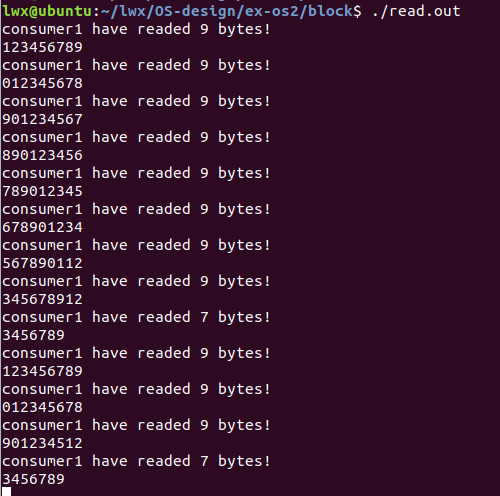
 

图2.15(1) 运行write.out终端输出 图2.15(2) 运行read.out终端输出

图2.15 阻塞状态测试（一）

使用命令“dmesg”查看内核日志如图2.16所示，截取部分效果明显信息，可看到同时有多个进程申请设备，且有相应提示信息输出。

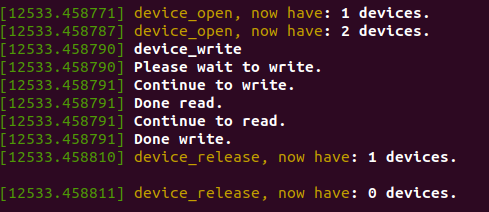


图2.16 查看内核日志

若仅单独启动read / write函数，write函数将在写入16字节后由于缓冲区已满而挂起，read也将因缓冲区全空而挂起，运行截图如图2.17、图2.18所示。

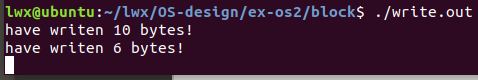
 

图2.17 阻塞状态测试（二）write函数 图2.18 阻塞状态测试（三）read函数

（二）非阻塞状态

非阻塞状态相对于阻塞状态，对read / write函数的改动有两处，一是改变open设备文件的方式，加入参数“O\_NONBLOCK”以表示非阻塞状态调用设备，二是当成功读出/写入的字节数为负数（调用失败）时，提示并退出。改动如图2.19所示。（以read程序为例）

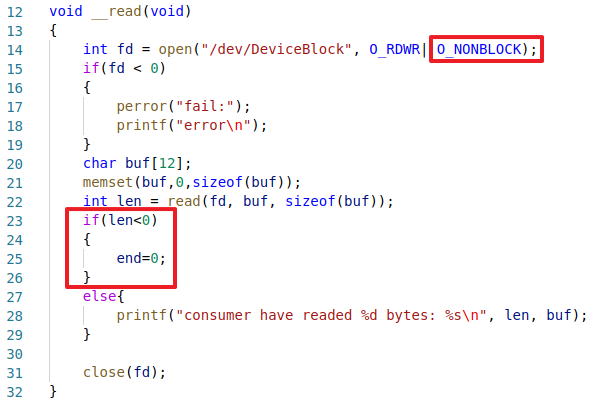
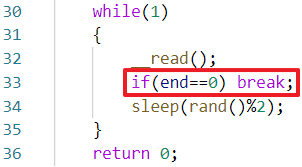
 

图2.19(1) 非阻塞状态下read函数改动 图2.19(2) 非阻塞状态下main函数改动

图2.19 非阻塞状态读函数改动

将用户缓冲区大小改为12，使读出略快于写入，程序可自行终止。首先启动write.out，写入数据后运行read.out，运行截图如图2.20所示。

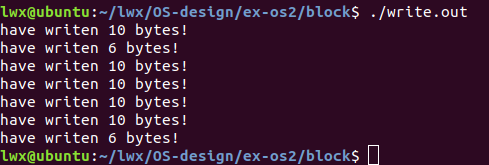
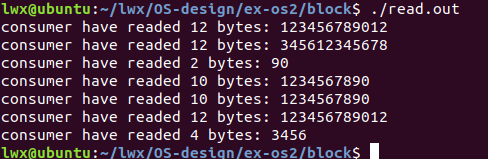
 

图2.20(1) 运行write.out终端输出 图2.20(2) 运行read.out终端输出

图2.20 非阻塞状态测试（一）

write.out运行一段时间后自行终止（而非如图2.15(1)一般被手动终止），随后read.out也结束。仔细对比两窗口可发现，read与write函数吞吐字节数相等，且read函数做到按顺序、无遗漏的读入。

内核信息与图2.16类似，不再赘述，单独运行write.out或read.out与图2.17、图2.18效果类似，但最后进程将自行终止而非挂起，效果如图2.21、图2.22所示。

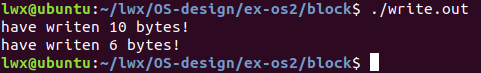
 

图2.21 非阻塞状态测试（二）write函数 图2.22 非阻塞状态测试（三）read函数

## 实验心得和建议

本次实验的原理等均在上学期的操作系统原理中讲过，类似的实验也在操作系统原理实验中涉及过，这次课设由于不用自己维护读写指针还更容易些。相比module内容，测试画的时间更长些。

起初在write函数的数组a中设定的数字长度比a的空间少一，即包括终止符，而后来经测试发现，将终止符‘\0’写入内核，将导致read时输出被截断，输出信息不符合预期，故后来调整使数字长度与数组长度一致。

而read函数中加入在每次内核read之前，都对buf缓冲区清零。在之前的测试中，发现无论显示读出几个字符，均会输出9或12的满额字符，后经检查发现，若不将buf清零，缓冲区内又无终止符结束字符串，有可能将之前未被覆盖的内容读出，从而影响显示效果。

任务二相对任务一较简单，给的资料很充足，也有之前的知识积累，测试内容与学习内容相符，无特别建议。

## 学习和编程实现参考网址

1. https://www.kernel.org/doc/htmldocs/kernel-api/index.html
2. https://blog.csdn.net/zhulinfeiba/article/details/5011573
3. https://blog.csdn.net/allen6268198/article/details/8112551