

# 实 验 报 告

课 程： 操作系统原理课程设计

院 系： 网络空间安全学院

专业班级： 信安1901班

学 号：

姓 名： 李文重

2022 年 03 月 9日

目录

[实 验 报 告 1](#_Toc14288)

[1.1 实验概述 3](#_Toc9870)

[1.1.1 课设任务 3](#_Toc7921)

[1.1.2 课设目的： 3](#_Toc7139)

[1.1.3 课设内容 3](#_Toc5869)

[1.2 实验设计思路 3](#_Toc9124)

[1.2.1 段描述符表的使用 3](#_Toc18862)

[1.2.2 特权级的使用 5](#_Toc25074)

[1.2.3 页式存储 7](#_Toc27948)

[1.2.4 中断的引入 8](#_Toc13785)

[1.2.5 实现进程切换 9](#_Toc10608)

[1.3 实验程序的难点或核心技术分析 9](#_Toc32706)

[1.4 运行和测试过程 12](#_Toc23479)

[1.5 实验心得和建议 14](#_Toc5253)

[1.6 学习和编程实现参考网址 15](#_Toc1988)

[2 实验二 设备阻塞工作机制 16](#_Toc2559)

[2.1 实验概述 16](#_Toc4251)

[2.1.1 课设任务 16](#_Toc12626)

[2.1.2 课设目的 16](#_Toc23612)

[2.1.3 课设内容 16](#_Toc29760)

[2.2 实验设计思路 16](#_Toc25115)

[2.3 实验程序的难点或核心技术分析 19](#_Toc13888)

[2.4 运行和测试过程 20](#_Toc25188)

[2.5 实验心得和建议 24](#_Toc7986)

[2.6 学习和编程实现参考网址 25](#_Toc13386)

## 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

其实在我看来，本次实验一并没有很多的体现设计这一个环节。所谓的设计更多反而是理解前人在计算机操作系统上的设计以及固化到硬件上的支持的操作。而本次实验更多的是遵照既定的标准，设计出一个能够在时钟驱动下的任务切换系统，如果说有设计的成分的话，更多的是代码如何组成，编程按照怎么样的顺序进行。因此本部分与其说是设计思路，更多的是一个学习笔记，关于我在进行这次试验的编程和调试思路在第三部分。

简单的概括一下，任务一的要求就是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。而段描述符表的使用，特权级的使用，页式存储，中断的引入以及实现任务切换就是本次实验要完成的重点部分。

以下为任务一及子功能实现所需要的原理以及简单代码介绍：

* + 1. 段描述符表的使用

保护模式和实模式主要的区别除了寄存器和地址大小之外，主要在于引入了段描述符表以寻址、和特权级概念来完成安全的保护。

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中的偏移位置，而TI位指示了应当从GDT还是LDT中寻址，RPL表示请求特权级。

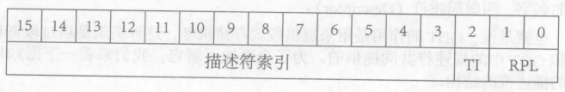
 

图1.1 段描述符 图1.2 选择子

最终代码中GDT定义如图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，目前段基址留空将在后面逐一填上（见图1.4），在代码中通过每段开头顶一个一个标签，然后把实模式下他的位置当作段基址来填入。选择子的定义采用“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，并且不含有“SA\_TIL”，恰好为图1.3中所指特权级为0的的GDT定义形式。

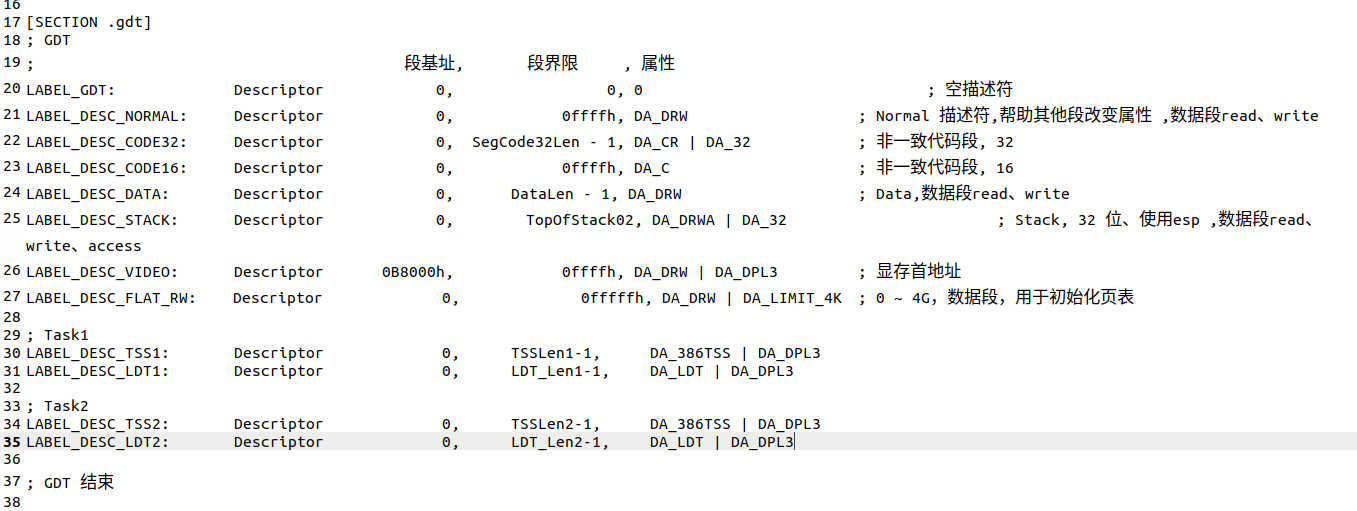


图1.3 GDT定义

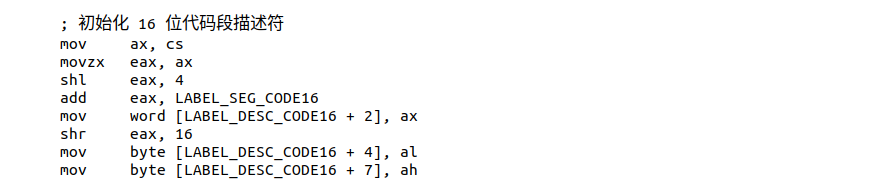


图1.4 GDT中段基址的初始化（篇幅所限，只取一例）

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1.5所示。

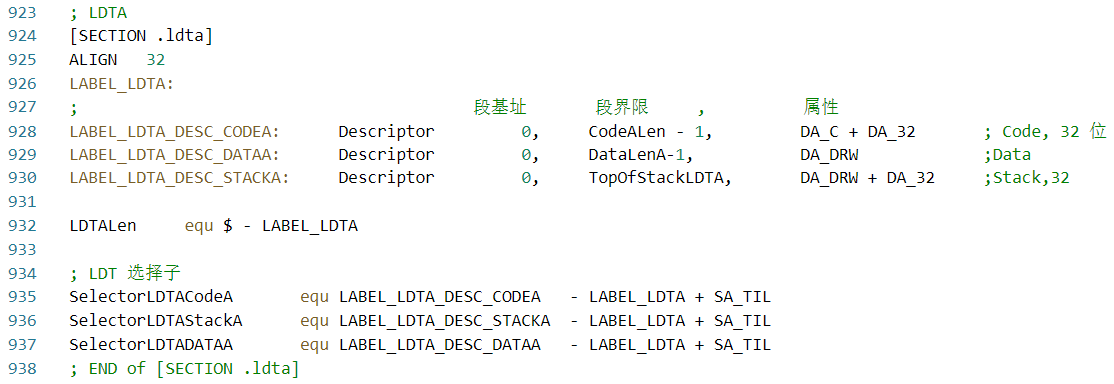


图1.5 LDT定义（其中之一）

在设置了以上部分之后，就可以进入保护模式了。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1.6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之后进入保护模式。

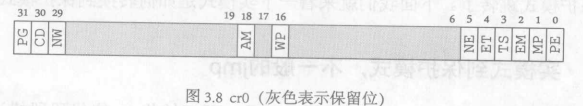


图1.6 CR0结构（灰色表示保留位）

* + 1. 特权级的使用

特权级的使用是为了让不同信任程度的代码运行在不一样的权限上，并且对于访问，跳转等做了严格的规定，这样就可以避免恶意进程破坏系统。

因此我首先依照于渊老师的代码，依次修改了各个段的访问权限（DPL，RPL）。这样可以为每个段规定其可能被调用的位置。

按照书中的说法，call和jmp经常被用来进行特权级跳转，而使用调用门跳转需要遵守的规则如下：

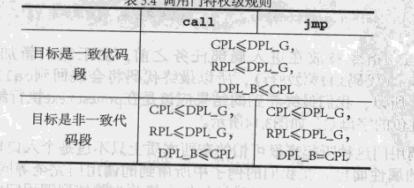


图1.7 调用门跳转的特权级规则

但是上图所示的call和jmp方法实际上都不能用来从ring0跳转到ring3，而这个跳转是把控制权交给指定的进程的重要跳转。而跳转需要借助到系统栈的使用，首先回忆一下一个长调用的返回发生时，他是如何获得新的代码运行位置以及堆栈基址和界限的。



图1.8 有特权级变换的长调用返回时堆栈示意

容易发现，实际上程序是在堆栈中获得之后的执行位置和新的堆栈相关信息的。因此只要提前压入信息，就可以实现需要的跳转，因此这个特权级跳转的最后代码如下图所示：

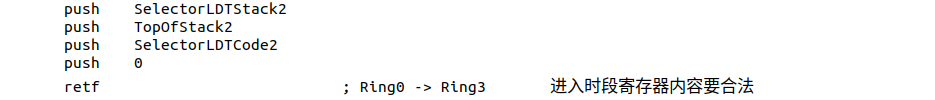


图1.9 从ring0跳转到ring3的代码

简洁的五行代码就实现了这个跳转。其中含义如英文所示。  
 除此之外的特权级跳转涉及到了TSS的使用，此处仅仅概述原理。其他放在第五部分来叙述。

从目前的角度来，TSS的作用主要就是在于保护进程运行的现场以及为进程切换提供必要的信息（主要是新的堆栈信息）。这些功能都可以可以显著的表现在他的构成上：

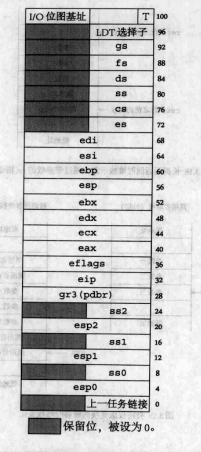


图1.10 32位TSS的结构

可以看到TSS中记录了三个不同的SS和ESP，这是因为当进程想要进入更高权限，总是需要使用jmp或者call指令，此时需要切换任务栈，而返回较低特权级的时候则自然的在栈中保存了返回的信息，因此不需要额外的保存这一部分信息。也正是因为此，可以看到没有记录ring3下的堆栈信息的SS3和ESP3。

在进入一个进城的时候ltr寄存器会存储下此时的TSS的选择子。当进程跳转的时候就会把相关的信息存储到这个TSS中，以期被继续使用。

* + 1. 页式存储

页式存储本质上是借助页表或者页目录把具体的物理的地址和逻辑地址分开。因此在实际访问的时候需要使用这两个表来进行地址转换。使用两级页表的代码的具体的转化过程可见下图：

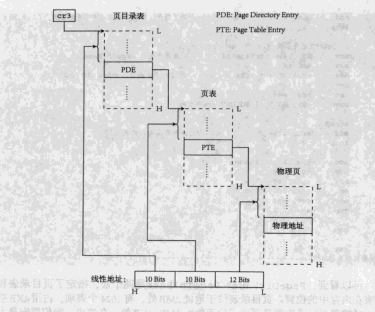


图1.11 三级页表映射的示意图

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，如图1.12。上文中提到的TSS通过储存CR3信息，完成在页式存储系统中的任务切换。

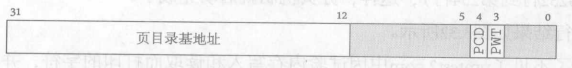


图1.12 CR3结构示意

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，而地址范围描述符每个只描述了一段连续空间的状态，所以需要通过一些代码技巧来进行读取以及进行信息的统计。之后就可以获得需要的页表个数以及大小。

为了方便起见，实际上初始化的页表和线性地址是一样的，这实际上并没有发挥页式的优势。但这部分显然为可能的优化提供了空间。

* + 1. 中断的引入

我们想要由代码实现一个两个进程互相切换，达成闪烁的两个字符串的目的。必须要引入一个控制是否进行切换的变量，而bochs支持了使用8529A来引入外部中断的方法，因此使用8529A的始终中断来控制。

由于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。设置8529A的操作被写入到了函数“Init8529A”，这是因为设置8529A需要通过一系列让人费解的过程，最好进行封装，之后把他视为一个模块来使用。同时设立一个IDT表，来控制收到外部中断时得反应，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“！”。因为以上部分的实现代码全部来自参考书1，因此不再放上代码。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。

* + 1. 实现进程切换

在代码中切换任务使用“JMP TSS”指令，其原理为：系统初始将某任务的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。

而调用这一过程的则是处于时钟中断中断向量处的这一段代码：



图1.13 时钟中断的处理程序

其中NextTask则是一个记录此时运行的是哪个进程的变量，会随着进程切换而改变。

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

实事求是地说，程序的主要代码框架实际上都来自于参考一（于渊老师的书）。因此这一部分重点将放在在途中遇到的问题上。

1. 格式化软盘 需要输入执行指令两次

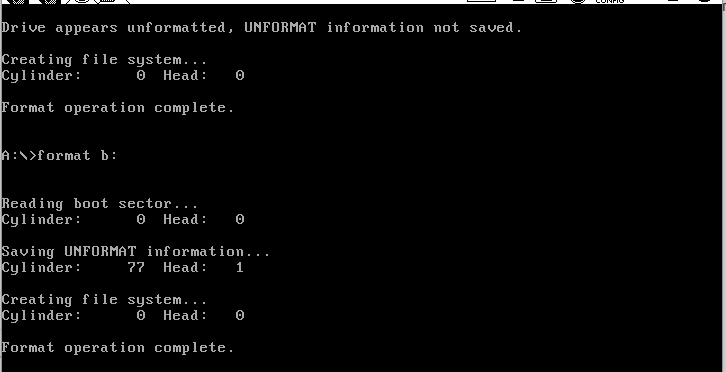


图1.14 格式化软盘的反馈

似乎第一次格式化的时候没有save成功（原因尚不了解），因此需要两次格式化。

1. Mount 步骤里面找不到挂载点

解决方法也很简单，就是在系统中提前makedir出指定路径。

1. 使用pm.inc和lib.inc的时候需要资料查看宏定义的代码’

只看课本中的描述可能会有歧义，导致错误使用宏。

1. 没有被直接触发的程序的TSS没有初始化导致jmp TSS时错误。

这是因为当切换到进程2（没有被直接调用的程序）时，使用的是jmp SlectorTSS:0，因此必须要初始化整个程序的初始状态，否则不知道需要跳到哪里继续执行或者获得堆栈段参数。而第一个程序，也就是被直接调用的程序，则不需要初始化（虽然我的代码出于对称做了初始化），这是因为该进程被中断并且跳转之后，会自动的保存到TSS里面。

1. 没有设置用于每个程序各自的ring0上的堆栈导致的错误

在最开始做的时候，我虽然在TSS中都分配了ring0上的栈段，但是实际上是使用的同一个代码段中的同一部分，也就是说SS和ESP全部相同。这样在运行过程中会直接崩溃。

我考虑问题可能并不出在使用相同的SS上，而在于使用相同的ESP，这是因为当某一个进程在ring0上进行了栈操作，而另一个TSS上的ESP并没有更新，这样导致了栈错误。换句话说，虽然整个运行过程是线性的并且每个进程都是堆栈平衡的，理论上如果可以同步ESP，应该是可以正常运行的。正当我按照这个思路去做的时候，同学给我了一个新的方式，直接在那个堆栈段中开出三个连续的足够大的空间，这样子可以近似的保证不会出现互相使用的情况，具体代码大概如下：

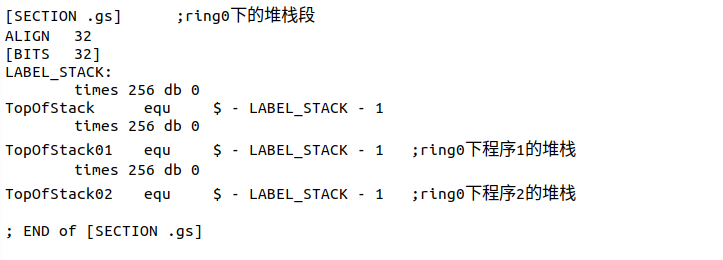


图1.15 重新定义后的全局栈段

这里假设256byte足够大不会越界，而TopOfStack，TopOfStack01，TopOfStack02分别给系统和两个进程的ring0作为ESP。

我这样改了之后，运行时却出现了未知错误，经过排查后发现，是我在GDT中没有及时修改段界限，段界限仍然使用的是TopOfStack，这样子在访问TopOfStack后面的部分的时候就会发生段访问错误。

1. ClockHandler中没有保存现场导致错误

在前文中我提到了切换TSS的时候会保存现场，但是这个到底在什么地方进行了TSS切换很容易产生歧义。我刚开始以为在时钟中断的时候，就已经出现了切换，因此错误认为保存的现场是进程被时钟中断打断时的现场。但实际上，这段代码的运行机制是，在jmp TSS的时候把当前的现场保存到TR寄存器中的选择子指向的TSS中。而在这之前，时钟中断已经创建一个需要保存的现场，就是正在运行中的Task，这个现场会存储在全局的堆栈中。 正因为保存在全局的堆栈中只有CS，IP等寄存器的值。所以在时钟中断中，需要利用栈来保存会使用到的寄存器的值，而这个栈中的值应当在后面恢复本进程TSS之后再替换此时的寄存器值（简单来说就是在jmp TSS2指令之后）。整个过程的流程描述如下图所示：

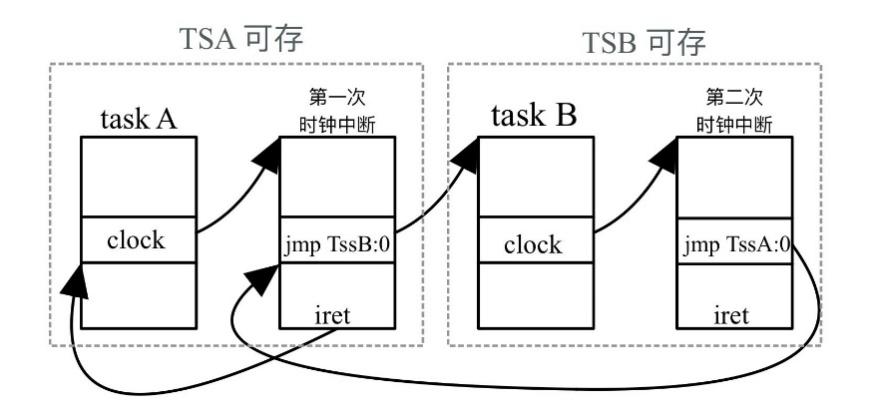


图1.16 程序跳转关系示意图

1. 因为没有及时写回导致读脏数据

首先来看一份我之前的代码。其中DS:NextTask是下一个将要转换的进程名的变量，是一个全局的变量。

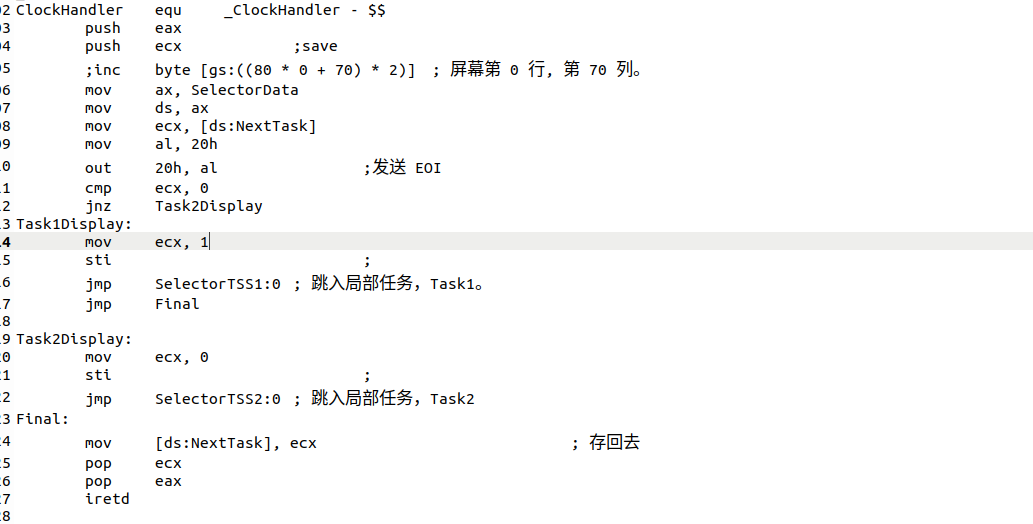


图1.17 一份错误的代码

注意到mov [ds:NextTask], ecx这一步，因为出现在了jmp之后。所以第二个程序被中断之后，另一个中断在[ds:NextTask],中读到的数据依然是之前的结果，导致跳转自己的TSS，发生错误。

## 运行和测试过程

将生成的虚拟软盘映像pm.img，与配置文件bochsrc，头文件lib.inc，dos以及下载的得到的虚拟软盘映像freedos.img（注意使用freedos需要修改初始运行位置）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“mywork.asm”，生成可执行文件“a.com”。

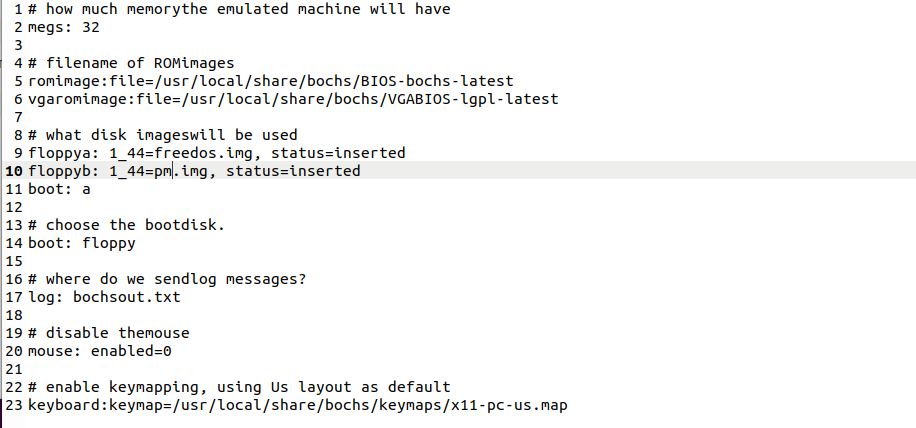


图1.18 配置文件--bochsrc



图1.19 makefile内容

使用bochs -f bochsrc 命令启动Bochs，输入指令“c”使虚拟机继续运行，启动dos系统，进入b盘，在b盘下执行“a.com”，操作如图1.25所示。

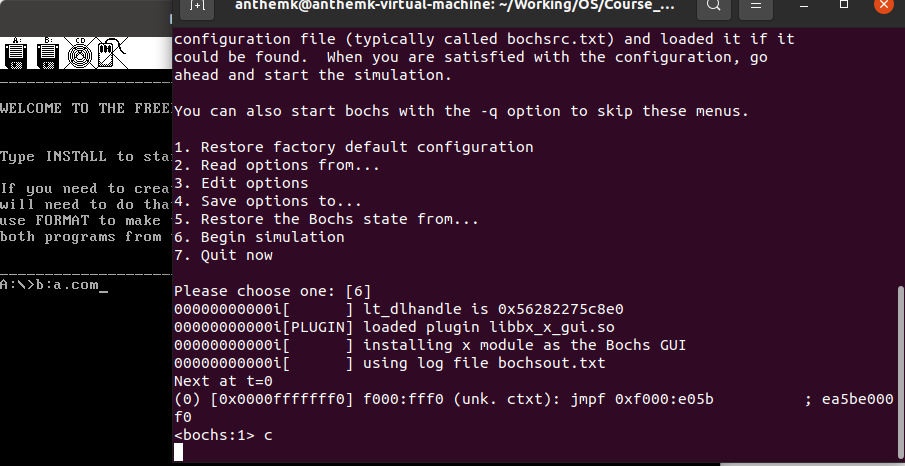


图1.20 启动bochs模拟器并且运行代码

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“IS19”，图片无法很明显的显示效果。

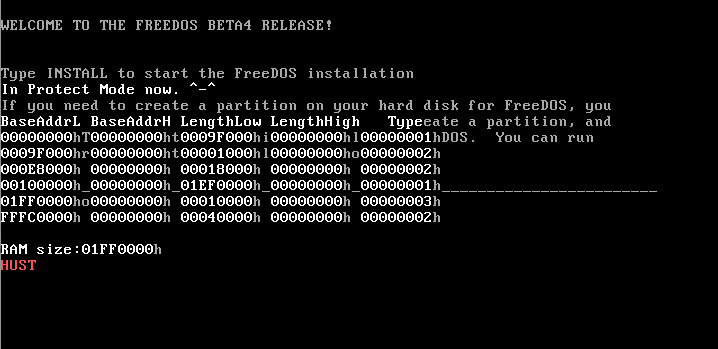
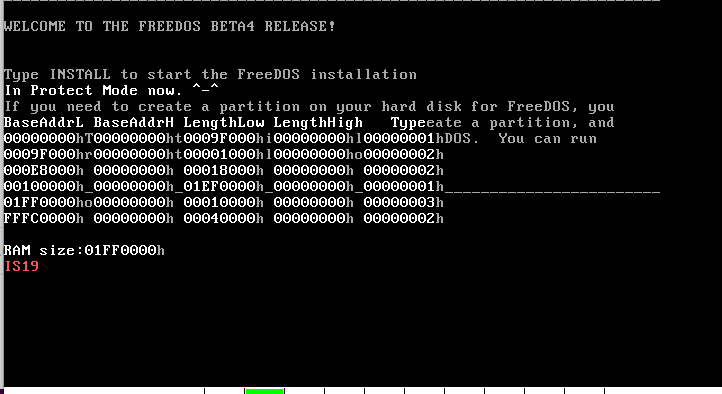


图1.21 显示字符串效果图

## 实验心得和建议

从我本人的实验过程来说，大多数的时间基本上都是在阅读课本。而写代码的部分，大多数都在改上文中提到过的奇奇怪怪的bug。

听同学们说一开始的任务目标并不是很明确，对于最终要实现的结果也不是非常清晰。但是因为我假期完全没有开始这个任务，并且开学的第一周一直在学习课本，等到我开始编写程序的时候已经有很多同学已经做得差不多了，所以我基本上没有遇到不清晰之处。

而在阅读课本中，学到了很多的东西，OS中的重要技术和概念也理解的更清晰了。尤其是关于保护模式的内容。但是我感觉似乎这个课本更多的还是教我们编写一个操作系统，它的讲述顺序是线性的，是一环扣一环的。这种方式当然有利于第一次编写系统的新手，但是另一方面，这种叙述方式让我们比较难去系统的理解为什么前人会这么设计这样有什么优势等等。

而在成功实现了两个字符串的闪烁时，还是很有成就感的，而且只有在亲身经历过这个过程才知道，一个看起来十分简单的字符串闪烁实际上是使用了多少计算机的知识，并且在这个过程中会出现多少意料中或者意料外的情况。

最后吐槽一句Bochs的调试真的很难用，而且因为涉及到实际的地址，很多时候甚至要用手来算地址，然后去看，非常的麻烦。但是考虑到是在比较基础的层面来进行实验，所以没有那么人性化也是完全可以理解的。

建议的话主要有两个：

1. 在做完实验之后，我才发现进程的介绍内容其实是在书（《Orange’s 一个操作系统的实现》）中的第六章中。而在任务书中的参考文献只有这本书前三章。这就让像我这种没仔细看目录的就没有学这一部分，结果就是在编写进程切换的部分花了比较多的时间。因此建议把第六章也加入到参考范围中。

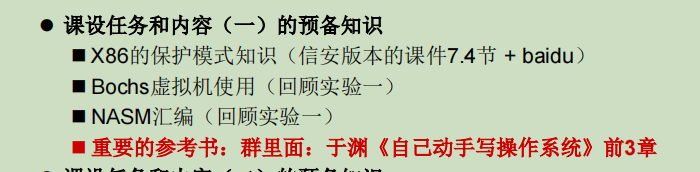


图1.22 任务书片段

1. 在我的理解里，如果要动手实现一个操作系统，那么在进程方面应当实现的是一个进程的管理系统，而不仅仅是具体的两个进程的切换。但是如果真的做进程管理系统，那工作量可能偏大，所以也不太现实。

## 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

[3] <https://blog.csdn.net/weixin_38391755/article/details/80380786>

# 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

一看这个任务要求，这个东西不是上学期做过嘛。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。原本我是想直接使用上学期的代码，不过既然老师推荐了使用kfifo结构，生成可以自动维护读写的缓冲区，和函数“wait\_queue\_head\_t”实现阻塞状态下设备读写队列的维护。所以我转而用这两个来完成了这次试验。

任务用到的主要定义如下：

1. miscdevice，如图2.2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。

设备声明1

图2.1 宏定义

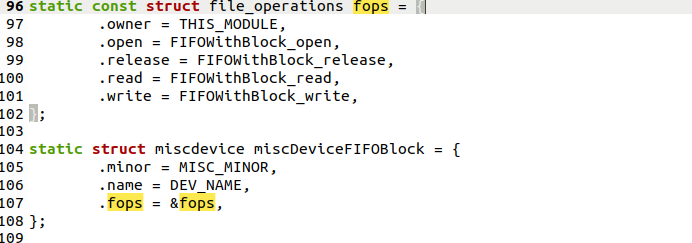


图2.2 数据结构miscdevice

而其中的结构体 file\_operations，是用来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。而owner则根本不是一个操作; 它是一个指向拥有这个结构的模块的指针. 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中, 它被简单初始化为 THIS\_MODULE, 一个在 <linux/module.h> 中定义的宏.

1. \_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。

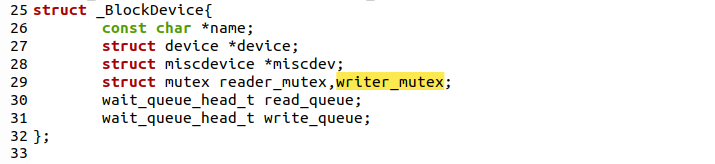


图2.3 Block Device结构体

1. wait\_queue\_head\_t，指示一个设备的等待队列。在\_BlockDevice中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。在阻塞情况下，需要使用这个来访时读空或者写满。
2. mutex，声明两个互斥锁reader\_mutex和writer\_mutex用来防止多个同时读或者写。

经过网上查阅资料可知，kfifo支持了同时读和写而不冲突，但是并不支持多个同时读或者多个同时写，而这个也是好解决的，只需要使用一个互斥锁，使得同时最多只有一个在进行读操作，并且只有一个在进行写操作就好了。

（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

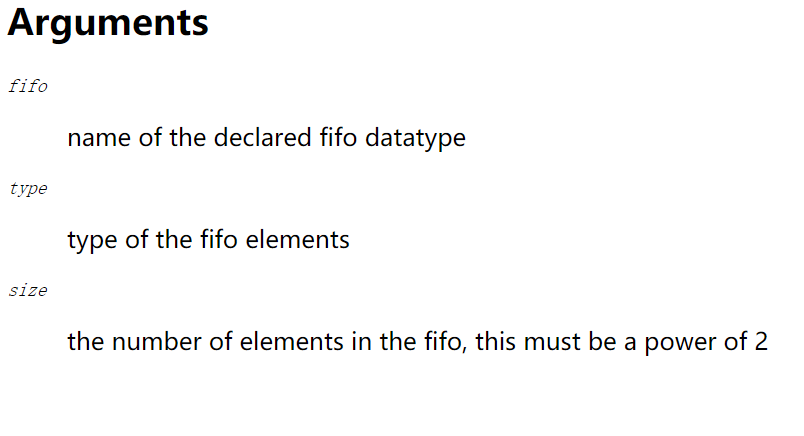


图2.5 DEFINE\_KFIFO的参数解释

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；

（7）mutex\_init、mutex\_lock、mutex\_unlock用来初始化锁，加锁和解锁。

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“FIFOWithBlock\_initt”将使用设备注册并判断注册是否出错，初始化等待队列，初始化互斥锁，使用“module\_init”运行初始化函数，配合命令将模块加载。在初始化BlockDevice的时候出现了一个小问题，这个将在下一部分中来说。

退出函数“FIFOWithBlock\_exitt”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。

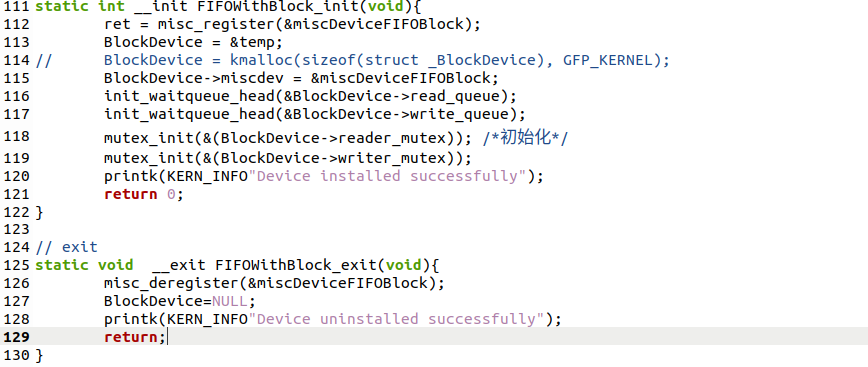


图2.6 初始化与注销模块

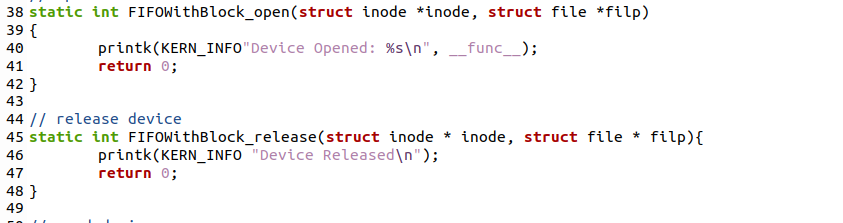


图2.7 open与realease函数

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，因为基本所有的初始化都随着设备的加载而完成，不需要在每次打开时操作。如图2.7所示，这两个函数仅仅用来输出提示信息到kernel，而这一部分信息可以通过dmesg命令来查看。

## 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（这一标志位通过宏来定义），判断调用状态为阻塞或非阻塞。而我采用或宏为真表示不阻塞，否则表示需要阻塞

（一）设备写：

如图2.8所示“读”函数。首先加锁，表示此时设备的读部分已被独占。然后对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。注意到两个return之前都必须解锁，以防止出现错误的永久性占用。



图2.8 阻塞状态下的read函数

考虑到队列唤醒的问题，加锁必须在加入等待队列之前。如果把加锁放到出队列之后可能会出现这样的情况，两个进程先后被从队列中唤醒，此时由于锁显然会先后读取。但第一个进程读光了缓存，这时候第二个请求面临的局面就成了，读空缓冲区而没有阻塞，导致错误。

如图2.9所示“写”函数，与“读”函数结构类似。首先加锁，表示此时设备的写部分已被独占。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。同样的返回时需要释放互斥锁。

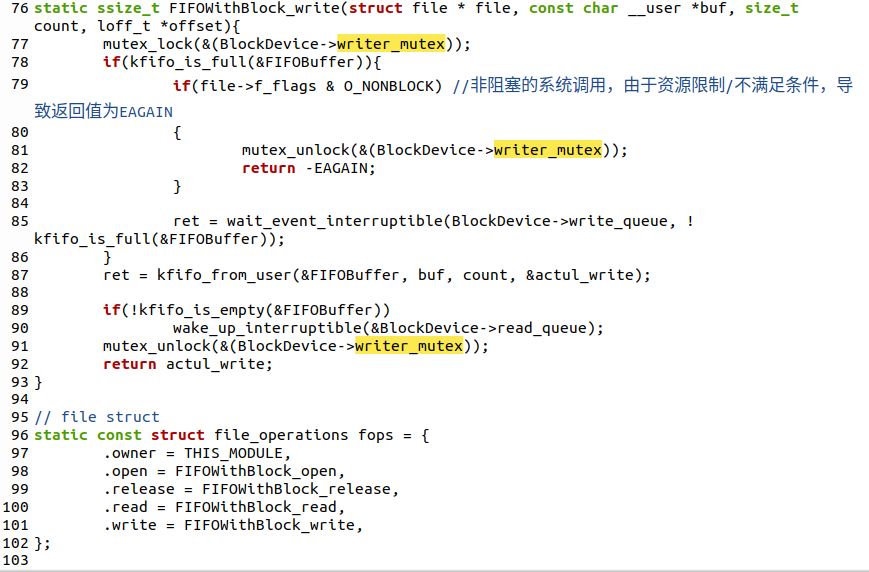


图2.9 阻塞状态下write函数

除此之外，我还遇到了一种奇怪的情况，就是在设备加载的时候，需要把结构体指针赋予实值，也就是说需要为他分配地址，理论上来说，使用kmalloc可以在内核区分配地址，但实际上似乎并不行，会出现错误。这个问题至今没有解决，所以我只能绕过这一步。我采用的方法是直接在程序中申请一个实际的结构体实例出来，这样在编译加载的时候会自动分配空间。理论上来说这个应当在加载的时候被分配，但我很怀疑在实际执行中这个空间会在内核堆栈中分配还是用户空间分配。

## 运行和测试过程

为了更方便的查看运行的状况，编写了两个读的测试程序（阻塞和非阻塞各一个）和一个写程序（提供写入给定值和随机定时写入两种模式），通过这两个程序的运行来检测编写的虚拟设备的运行情况。

首先是写程序的代码如下：



图2.10 写入设备的测试程序代码

可以看到支持了了两种方式，分别是random和正常的把输入的存到设备里。

（一）阻塞状态：

阻塞读代码如下。每次读取一个char字符并且输出，之后sleep1秒。之所以设置成每次只读取一个字符并且中间休眠是因为方便检测多个读程序同时运行的情况。在第11行可以看到，open的时候没有使用宏来屏蔽阻塞。



图2.11 阻塞状态下的读设备测试代码

主函数内容为死循环，不断调用read函数，并休眠固定的时间。

为使实验效果明显，模块设定的内核缓冲区大小BUFFER\_SIZE为32，同时在两终端分别启动“write.out”与“read.out”，写入abcdefg，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。而且显而易见读进程已经被阻塞住了。

IMG_256

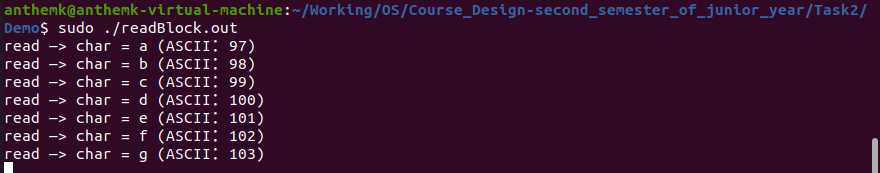
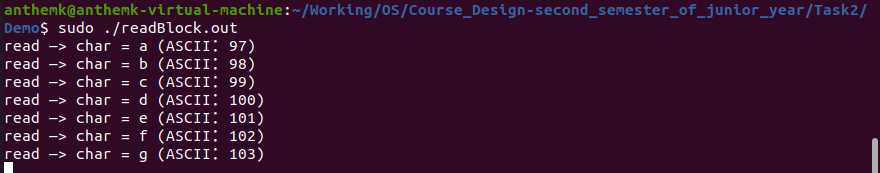


图2.12 阻塞状态下的读写测试

完成了对这些字符的读取之后，使得缓冲区为空之后，可以使用ps指令看到readBlcok程序已经进入了sleep状态。



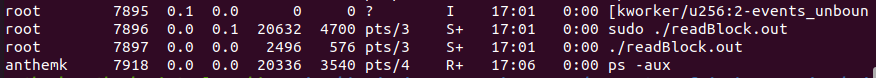
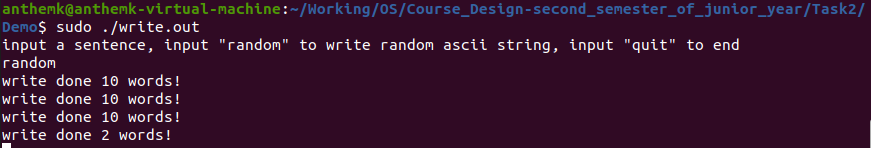


图2.13 读进程陷入阻塞

同样的如果只运行写入使得写满缓冲区也会触发阻塞：



IMG_256

图2.14 写进程陷入阻塞

同时可以使用命令“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。

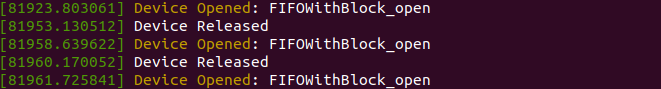


图2.15 内核日志

（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，输出一个错误信息。而且为了在ps中看到程序始终在运行没所以没有设置1秒的sleep。改动如图2.19所示



图2.16 非阻塞状态下read函数改动（写函数改动部分基本相同）

大多数运行结果都和阻塞相同，篇幅所限这一部分主要展示非阻塞的运行过程。

这是在读的过程中读完了整个缓冲区，因此陷入无限循环中。



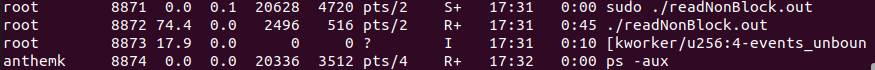
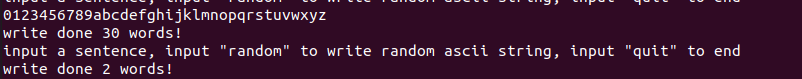


图2.17 读进程没有陷入阻塞

通过ps指令可以看到进程是一直处于Run的状态中。说明阻塞没有发生。

（三）并行测试

因为写需要手动输入并且具有时间所以采用多个read并行来。为了保证会发生冲突，需要去掉sleep而设置一个总的运行次数，然后停止，当数据足够多的时候可以认为同时读会发生。



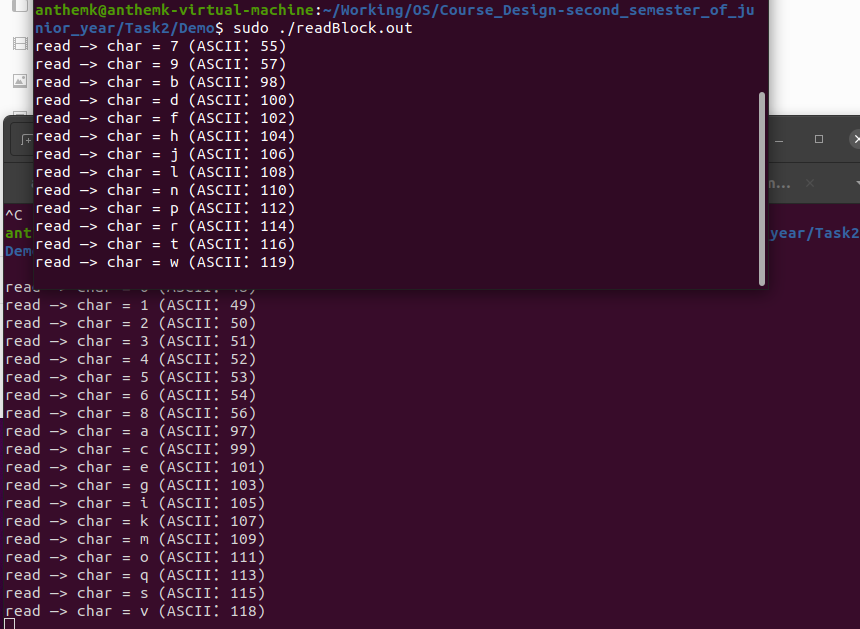


图2.18 并行测试结果

## 实验心得和建议

本次实验更多的是上学期的实验中的一个延申，而且因为老师比较明确的规定了使用的技术，因此代码里很多都是直接抄的，并没有很多我自己的部分。所以总的来说这个实验没有很多的需要学习、理解的内容，因此自然的做的比实验一快得多。

唯一让我卡了一段时间的地方是在传递flag的时候，没有加上O\_RDWR。这里我抄博客的时候以为是博主在他的任务中自定义的宏，就直接给去掉了，后来才发现是Linux定义的允许读、写打开，只有加上才能正确的运行。

任务二相对任务一较简单，给的资料很充足，再加上之前的知识积累，实现的难度不算高。

没有什么特别要提得到的建议。

## 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/u011298001/article/details/86063987>
4. <https://blog.csdn.net/weixin_38391755/article/details/80380786>
5. <https://blog.csdn.net/ll148305879/article/details/92834622>
6. <https://blog.csdn.net/allen6268198/article/details/8112551>