**鲁东大学信息与电气工程学院**

**实验报告**

（ 2022 — 2023 学年第 1 学期）

2220180106

**课程名称**  操作系统

**实验题目** 设备管理

**专 业** 软件工程

**班 级**  2102

**姓 名**  莫倩烨

**学 号**  2021220301

2023 年 6 月 3 日

|  |  |  |  |
| --- | --- | --- | --- |
| 实验题目 | 实验5 设备管理 | | |
| 实验类型 | 验证、设计 | 实验日期 | 2023 年 5 月 20 日 |
| 题目来源 | ✔1.必修 2.选修 3.自拟(设计) 4.专题 | | |
| **一、实验目的及要求**  1．**理解**设备管理的概念和任务；  2．**了解**Linux字符设备和块设备的管理机制；  3．**学习**字符设备和块设备的基本管理方法；  4．**学会编写**一个简单的字符设备驱动程序和块设备驱动程序；  5．**了解**磁盘结构以及磁盘上数据的组织方式；  6．**掌握**磁盘访问时间的计算方式；  7．**掌握**常用磁盘调度算法及其相关特性。  **二、实验仪器设备与软件环境**  1．一台装有Windows的微型计算机（Win7或Win10）；  2．在微型计算机上（通过虚拟机）安装了Linux环境；  3．Linux环境下的编辑器（vi/vim、gedit或vscode），编译器（gcc），调试器（gdb）。  **三、实验基础**  **（一）设备管理**  本实验模拟独占设备的分配、回收过程。  为了提高操作系统的可适应性和可扩展性，现代操作系统中都毫无例外地实现了设备独立性，又叫做设备无关性。设备独立性的含义是：应用程序独立于具体使用的物理设备。为了实现独占设备的分配，系统设置数据表格的方式也不相同，在实验中只要设计合理即可。这里仅仅是一种方案，采用设备类表和设备表。  **1、数据结构**  操作系统设置“设备分配表”，用来记录计算机系统所配置的独占设备类型、台数以及分配情况。设备分配表可由“设备类表”和“设备表”两部分组成，如下图：    “设备类表”记录系统中的各类设备，每类设备占用一个登记栏，登记该类设备的总台数、当前有几台设备空闲以及该类设备在设备表中的起始地址。每一台设备在“设备表”中占用一个登记项，同类的若干台设备连续登记在设备表中。设备表中登记每一台设备的绝对号、设备的好坏情况、设备是否分配、设备被哪一个作业占用和设备相对号。 本实验假设系统共有4类设备，分别是输入机2台、打印机3台、磁盘机4台、磁带机1台，共10个设备。  **2、设备分配**  1）当进程申请某类设备时，系统先查“设备类表”；  2）如果该类设备的现存台数可以满足申请要求，则从该类设备的“设备表”始址开始依次查该类设备在设备表中的登记项，找出“未分配”的设备分配给进程；  3）分配后要修改设备类表中的现存台数，把分配给进程的设备标志改为“已分配”且填上占用设备的进程名；  4）把设备的绝对号与相对号的对应关系通知用户，以便用户在分配到的设备上装上存储介质。  **3、设备回收**  当进程执行结束撤离时应归还所占设备，系统根据进程名查设备表，找出进程占用设备的登记栏，把标志修改为“未分配”，清除进程名。同时把回收的设备台数加到设备类表中的现存台数中。  **（二）磁盘调度算法**  **1、磁盘数据的组织**  磁盘上每一条物理记录都有唯一的地址，该地址包括三个部分：磁头号（盘面号）、柱面号（磁道号）和扇区号。给定这三个量就可以唯一地确定一个地址。  **2、磁盘访问时间的计算方式**  磁盘在工作时以恒定的速率旋转。为保证读或写，磁头必须移动到所要求的磁道上，当所要求的扇区的开始位置旋转到磁头下时，开始读或写数据。对磁盘的访问时间包括：寻道时间、旋转延迟时间和传输时间。  **3、磁盘调度算法**  磁盘调度的目的是要尽可能降低磁盘的寻道时间，以提高磁盘 I/O 系统的性能。  1）先进先出算法FIFO：按访问请求到达的先后次序进行调度。这是一种比较简单的磁盘调度算法。它根据进程请求访问磁盘的先后次序进行调度。此算法的优点是公平、简单，且每个进程的请求都能依次得到处理，不会出现某一进程的请求长期得不到满足的情况。此算法由于未对寻道进行优化，在对磁盘的访问请求比较多的情况下，此算法将降低设备服务的吞吐量，致使平均寻道时间可能较长，但各进程得到服务的响应时间的变化幅度较小。  2）最短服务时间优先算法SSTF：优先选择使磁头臂从当前位置开始移动最少的磁盘I/O请求进行调度。该算法选择这样的进程，其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短，该算法可以得到比较好的吞吐量，但却不能保证平均寻道时间最短。其缺点是对用户的服务请求的响应机会不是均等的，因而导致响应时间的变化幅度很大。在服务请求很多的情况下，对内外边缘磁道的请求将会无限期的被延迟，有些请求的响应时间将不可预期。  3）SCAN（扫描算法）：要求磁头臂先沿一个方向移动，并在途中满足所有未完成的请求，直到它到达这个方向上的最后一个磁道，或者在这个方向上没有别的请求为止，后一种改进有时候称作LOOK策略。然后倒转服务方向，沿相反方向扫描，同样按顺序完成所有请求。SCAN算法不仅考虑到将要访问的磁道与当前磁道的距离，更优先考虑的是磁头的当前移动方向。例如，当磁头正在自里向外移动时，扫描算法所选择的下一个访问对象应是其将要访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直到再无更外的磁道需要访问才将磁臂换向，自外向里移动。这时，同样也是每次选择这样的进程来调度，即其要访问的磁道，在当前磁道之内，从而避免了饥饿现象的出现。由于这种算法中磁头移动的规律颇似电梯的运行，故又称为电梯调度算法。此算法基本上克服了最短寻道时间优先算法的服务集中于中间磁道和响应时间变化比较大的缺点，而具有最短寻道时间优先算法的优点即吞吐量较大，平均响应时间较小，但由于是摆动式的扫描方法，两侧磁道被访问的频率仍低于中间磁道。  4）C-SCAN（循环扫描）算法：在磁盘调度时，把扫描限定在一个方向，当沿某个方向访问到最后一个磁道时，磁头臂返回到磁盘的另一端，并再次开始扫描。循环扫描算法是对扫描算法的改进。如果对磁道的访问请求是均匀分布的，当磁头到达磁盘的一端，并反向运动时落在磁头之后的访问请求相对较少。这是由于这些磁道刚被处理，而磁盘另一端的请求密度相当高，且这些访问请求等待的时间较长，为了解决这种情况，循环扫描算法规定磁头单向移动。例如，只自里向外移动，当磁头移到最外的被访问磁道时，磁头立即返回到最里的欲访磁道，即将最小磁道号紧接着最大磁道号构成循环，进行扫描。  四、实验内容  1．编写字符设备驱动程序，要求能对字符设备执行打开、读、写、I/O控制和关闭这5项基本操作；  2．编写一个应用程序，测试添加的字符设备和块设备驱动程序的正确性；  3．编写一个简单的块设备驱动程序，实现一套内存中的虚拟磁盘驱动器，并通过实际操作验证块设备驱动是否可以正常工作；  4．本实验通过编程模拟实现几种常见的磁盘调度算法。  1）测试数据及测试结果：    2）使用 C语言编程实现 FIFO、SSTF、SCAN、C-SCAN算法。  五、实验指导  **1、设备管理**  本实验涉及的操作需要管理员权限，因此我们需要切换到root权限或使用sudo命令。具体的操作步骤如下。  1）编写字符设备驱动程序  (1)编写设备驱动源程序，即编写内核模块文件char dev.c和Makefile文件。  (2)使用make命令编译驱动模块。  (3)使用insmod命令安装驱动模块。  (4)创建字符设备文件，方法是使用mknod命令，语法格式为：mknod /dev/文件名 c 主设备号 次设备号。然后使用ls/dev命令查看所创建的字符设备文件。  (5)编写测试程序test.c，访问创建的字符设备文件，并使用gcc编译这个字符设备文件，然后运行。  (6)使用rmmod卸载模块。  (7)使用rm命令删除所创建的字符设备文件。  chardev.c文件中的示例内容如下：  #include <linux/init.h>  #include <linux/kernel.h>  #include <linux/slab.h>  #include <linux/module.h>  #include <linux/moduleparam.h>  #include <linux/fs.h>  #include <linux/u access.h>  #define SUCCESS O  #define DEVICE\_NAME "chardev"  #define BUF\_LEN 80  static int Major;  static int Device\_Open = 0;  static char msg[BUF\_LEN];  static char \*msg\_Ptr;  static int device\_open(struct inode \*inode, struct file \*file);  static int device\_release(struct inode \*inode, struct file \*file);  static ssize\_t device\_read(struct file \*filp, char \*buffer， size\_tlength, loff\_t \*offset);  static ssize\_t device\_write(struct file \*filp, const char \*buff, size\_tlength, loff\_t \*off);  static struct file\_operations fops = {  .read = device\_read,  .write = device\_write,  .open = device\_open,  .release = device\_release  };  //打开设备  static int device\_open(struct inode \*inode, struct file \*file)  {  static int counter=0;  if (Device\_Open)  return -EBUSY;  Device\_Open++;  sprintf(msg,"I already told you %d times Hello world\n", counter++);  msg\_Ptr = msg;  try\_module\_get(THIS\_MODULE);  return SUCCESS;  }  //释放设备  static int device\_release(struct inode \*inode, struct file \*file)  {  Device\_Open--;  module\_put(THIS\_MODULE);  return 0;  }  //读设备  static ssize\_t device\_read(struct file \*filp, char \*buffer, size\_tlength, loff\_t \*offset)  {  if(\*msg\_Ptr==0)  return 0;  copy\_to\_user(buffer, msg\_Ptr, length);  return strlen(msg);  }  **//写设备**  static ssize\_t device\_write(struct file \*filp, const char \*buff, size\_tlength, loff\_t \*off)  {  printk("<1> Sorry this operation is n't supported\n"); **//未实现写的功能**  return -EINVAL;  }  //初始化字符设备  int init\_chardev\_module(void)  {  Major = register\_chrdev(0, DEVICE\_NAME, &fops);  if(Major<0) {  printk("Registering the character device failed with %d\n",Major);  return Major;  }  printk("<1> I was assigned major number %d", Major);  printk("<1> the drive，create a dev file");  printk("<1> mknod/dev/hello c %d 0.\n", Major);  printk("<1> I was assigned major number %d", Major);  printk("<1> the device file\n");  printk("<1> Remove the file device and module when done\n");  return 0;  }  //关闭字符设备  void exit\_chardev\_module(void)  {  unregister\_chrdev(Major, DEVICE\_NAME);  }  MODULE\_LICENSE("Dual BSD/GPL");  module\_init(init\_chardev\_module);  module\_exit(exit\_chardev\_module);  Makefile文件中的示例内容如下：  TARGET = chardev  Obj -m += S(TARGET).o  cc = gcc  KDIR := /lib/modules/$(shell uname -r) /build  PWD := $(shell pwd)  all: modules  modules：  S(MAKE) -C $(KDIR) M=$(PWD) modules  clean：  S(MAKE) -C $(KDIR) M=$(PWD) clean  测试文件test.c中的示例代码如下：  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <sys/types.h>  int main()  {  char buf[4096] = {"I have already told you 1 time hello world");  int fd = open("/dev/hello", O\_RDWR);  int ret = read(fd, buf, sizeof(buf)) ;  buf[ret] = '\0';  printf("%s\n", buf);  }  2）编写块设备驱动程序  (1)编写设备驱动源程序，即编写内核莫块文件simp\_blkdev.c和Makefile文件。  (2)使用make命令编译驱动模块。  (3)使用insmod命令安装驱动模块。  (4)使用lsblk命令列出当前的块设备信息。  (5)格式化设备simp\_blk dev。  (6)创建挂载点并挂载块设备。  (7)查看模块使用情况，会发现模块已被调用。  (8)对块设备驱动进行调用测试。  (9)取消挂载，查看模块调用结果。  (10)使用rmmod命令卸载模块。  simp\_blkdev.c文件的示例内容如下：  #include <linux/module.h>  #include <linux/blkdev.h>  #define SIMP\_BLKDEV\_DISKNAME "simp\_blkdev"  #define SIMP\_BLKDEV\_DEVICEMAJOR COMPAQ\_SMART2\_MAJOR  #define SIMP\_BLKDEV\_BYTES (50\*1024\*1024)  #define SECTOR\_SIZE\_SHIFT 9  static struct gendisk \*simp\_blkdev\_disk;  static struct block\_device\_operations simp\_blkdev\_fops={.owner=THIS\_MODULE};  static struct request\_queue \*simp\_blkdev\_queue;  unsigned char simp\_blkdev\_data[SIMP\_BLKDEV\_BYTES];  /\* 磁盘块设备数据请求处理函数 \*/  static void simp\_blkdev\_do\_request(struct request\_queue \*q) {  struct request \*req; //正在处理的请求队列中的请求  struct bio \*req\_bio; //当前请求的bio  struct bio\_vec \*bvec; //当前请求bio的段  char \*disk\_mem; //需要读/写的磁盘区域  char \*buffer; //磁盘块设备的请求在内存中所处的缓冲区  int i = 0;  while ((req = blk\_fetch\_request(q)) != NULL) {  //判断当前req是否合法  if((blk\_rq\_pos(req)<<SECTOR\_SIZE\_SHIFT) + blk\_rq\_bytes(req)>SIMP\_  BLKDEV\_BYTES) {  printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME": bad request: block = %llu,  count = %u \n", (unsigned long long)blk\_rq\_pos(req),  blk\_rq\_sectors(req));  blk\_end\_request\_all(req, -EIO);  continue;  }  //获取需要操作的内存位置  disk\_mem = simp\_blkdev\_data + (blk\_rq\_pos(req)<<SECTOR\_SIZE\_SHIFT);  req\_bio = req->bio; //获取当前请求的bio  switch (rq\_data\_dir(req)) {  case READ:  while(req\_bio != NULL) {  for(i=0; i<req\_bio->bi\_vcnt; i++) {  bvec = &(req\_bio->bi\_io\_vec[i]);  buffer = kmap(bvec->bv\_page) + bvec->bv\_offset;  memcpy (buffer, disk\_mem, bvec->bv\_len);  kunmap (bvec->bv\_page);  disk\_mem += bvec->bv\_len;  }  req\_bio = req\_bio->bi\_next;  }  \_blk\_end\_request\_all(req, 0);  break;  case WRITE:  while(req\_bio != NULL) {  for(i=0; i<req\_bio->bi\_vcnt; i++) {  bvec = &(req\_bio->bi\_io\_vec[i]);  buffer = kmap(bvec->bv\_page) + bvec->bv\_offset;  memcpy(disk\_mem, buffer, bvec->bv\_len);  kunmap(bvec->bv\_page);  disk\_mem += bvec->bv\_len;  }  req\_bio = req\_bio->bi\_next;  }  \_blk\_end\_request\_all(req, 0);  default:  break;  }  }  }  /\* 模块入口函数 \*/  static int \_initsimp\_blkdev\_init(void) {  int ret;  //(1)在添加设备之前，先申请设备资源  simp\_blkdev\_disk=alloc\_disk(1);  if(!simp\_blkdev\_disk) {  ret = -ENOMEM;  goto err\_alloc\_disk;  }  //(2)设置设备相关属性(如设备名、设备号、请求队列等)  strcpy (simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);  simp\_blkdev\_disk->major = SIMP\_BLKDEV\_DEVICEMAJOR;  simp\_blkdev\_disk->first\_minor = 0;  simp\_blkdev\_disk->fops = &simp\_blkdev\_fops;  //将块设备请求处理函数的地址传入blk\_init\_queue()函数并初始化一个请求队列  simp\_blkdev\_queue = blk\_init\_queue(simp\_blkdev\_do\_request, NULL);  if(!simp\_blkdev\_queue) {  ret = -ENOMEM;  goto err\_init\_queue;  }  simp\_blkdev\_disk->queue = simp\_blkdev\_queue;  set\_capacity(simp\_blkdev\_disk, SIMP\_BLKDEV\_BYTES>>9);  //(3)在入口处添加磁盘块设备  add\_disk(simp\_blkdev\_disk);  return 0;  err\_alloc\_disk:  return ret;  err\_init\_queue：  return ret;  }  /\*模块出口函数\*/  static void \_exitsimp\_blkdev\_exit(void) {  del\_gendisk(simp\_blkdev\_disk); //释放磁盘块设备  put\_disk(simp\_blkdev\_disk); //释放申请的设备资源  blk\_cleanup\_queue(simp\_blkdev\_queue); //清除请求队列  }  MODULE\_LICENSE("GPL");  module\_init(simp\_blkdev\_init);  module\_exit(simp\_blkdev\_exit);  Makefile文件中的示例内容如下：  TARGET = simp\_blkdev  obj -m += $(TARGET).o  cc = gcc  KDIR := /lib/modules/$(shell uname -r) /build  PWD := $(shell pwd)  all: modules  modules:  $(MAKE) -C $(KDIR) M=$(PWD) modules  clean:  $(MAKE) -C $(KDIR) M=$(PWD) clean  **2、磁盘调度算法**  实验中，使用给定的测试数据及测试结果，按照如下的算法流程图编写代码：    六、实验过程（80分）  本实验要求学生按照以下操作步骤（含代码、数据、图表等）完成实验，并提交实验结果截图。  （一）**设备管理**（40分）  1. 编写字符设备驱动程序  实验代码的编写与调试：  （注意：要实现向字符设备写数据的代码）  全部模块程序为：  汇总得到globalvar.c如下    编写读程序  读程序read.c代码如下    编写写程序  写程序write.c代码如下    编写Makefile文件  为了编译模块程序globalvar.c，我们建立makefile文件如下：    实验过程：  (1)使用make命令编译字符设备驱动模块；  (2)使用insmod char dev.ko命令安装编译好的字符驱动模块，使用ls mod|grep char dev命令可以查看该模块是否装载成功；  (3)使用dmesg命令查看系统分配的主设备号；  (4)根据输出的主设备号，利用mknod命令创建设备；  (5)编译并运行测试程序test.c；  (6)卸载模块后，查看字符设备的工作情况。        实验结果讨论：  private\_data 其实是用来保存自定义设备结构体的地址的。自定义结构体的地址被保存在private\_data后，可以在read ,write 等驱动函数中被传递和调用自定义设备结构体中的成员。  globalvar.flag标志当前是否可读，若不可读，则用wait\_event\_interruptible将其挂起到等待队列globalvar.outq。如果可以读，则使用down\_interruptible(&globalvar.sem)进行P操作。接下来更新读指针，len是读的字节数，如果读指针小于写指针，表明新写入了内容，应该读取从写指针到读指针的内容，即len = min(len,(size\_t)(globalvar.wr - globalvar.rd))。如果如指针大于等于写指针，表明在循环缓冲区中写指针已经过了一次循环，应令len = min(len,(size\_t)(globalvar.end - globalvar.rd))将当前读指针到结尾的内容读完，下一次再读到写指针。用copy\_to\_user(buf,globalvar.rd,len)函数将内核空间的数据读取出来，并更新读指针位置，如果读指针在缓冲区末尾则将其循环地置为缓冲区首部。最后进行V操作退出临界区。  2. 编写块设备驱动程序  实验代码的编写与调试：    实验过程：  (1)使用make命令编译块设备驱动模块；  (2)挂载块设备驱动模块simp\_blkdev.ko，并使用lsmod|grep simp\_bikdev命令查看是否挂载成功；  (3)使用lsblk命令列出当前的块设备信息；  (4)格式化块设备simp\_blkdev，建立文件系统；  (5)创建挂载点并挂载块设备；  (6)再次查看模块使用情况以及使用者；  (7)对块设备驱动进行调用测试；  (8)查看资源使用情况；  (9)删除文件并再次查看资源使用情况；  (10)取消挂载，查看模块调用情况；  (11)模块被卸载后，再查看它的情况。  实验结果讨论：  为了编译模块程序globalvar.c，我们建立makefile文件如下写操作和读操作大致相同，只不过读写方向相反。首先用P操作进入临界区，计算写入长度len。如果读指针小于写指针，则可以令len = min(len,(size\_t)(globalvar.end - globalvar.wr)); 表示从当前写指针到缓冲区末尾都可以写入。如果读指针大于写指针，则只能写入从写指针到读指针之前的位置，即len = min(len,(size\_t)(globalvar.rd-globalvar.wr-1));，否则会破坏还未读取的内容，读取时也不能读取写指针后面、本应读取到的内容。最后更新写指针，进行V操作退出临界区，通过 wake\_up\_interruptible(&globalvar.outq)唤醒读进程。采用这种方式实际上写入的优先级更高。  （二）**磁盘调度算法**（40分）  实验代码：  (1)先进先出算法FIFO：    (2)最短服务时间优先算法SSTF：    (3)扫描算法SCAN：    (4)循环扫描算法C-SCAN：    (5)主程序：    实验结果：  (1)先进先出算法FIFO：    (2)最短服务时间优先算法SSTF：    (3)扫描算法SCAN：    (4)循环扫描算法C-SCAN：    性能分析：  1）先来先服务算法（FCFS）： 优点：公平；如果请求访问的磁道比较集中的话，算法性能还算过的去 缺点：如果有大量进程竞争使用磁盘，请求访问的磁道很分散，则FCFS在性能上很差，寻道时间长。  2）最短寻道时间优先法（SSTF）： 优点：性能较好，平均寻道时间短 缺点：可能产生“饥饿”现象，在附录给的测试文件中，假设在处理18号磁道的访问请求时又来了一个38号磁道的访问请求，处理38号磁道的访问请求时又来了一个18号磁道的访问请求。如果有源源不断的18号、38号磁道的访问请求到来的话，150、160、184号磁道的访问请求就永远得不到满足，从而产生“饥饿”现象。  六．实验结果分析（20分）  1．实验思考  （1）总结并分析实验中出现的问题及对应的解决方法。  在编写字符设备驱动程序，要求能对字符设备执行打开、读、写、I/O控制和关闭这5项基本操作时，步骤如下：    （2）分析字符设备与块设备的驱动程序，指出它们在实现过程中的异同点。  1.字符设备只能以字节为最小单位访问，而块设备以块为单位访问，例如512字节，1024字节等  2.块设备可以随机访问，但是字符设备不可以  3.字符和块没有访问量大小的限制，块也可以以字节为单位来访问  （3）通过对每个算法进行时间复杂度分析对比，每个算法的效率如何？  1）当磁道初始位置在整体访问磁道数据的偏大或者偏小位置的时候，FIFS时间复杂度为O(n)最低，其他两个算法时间复杂度均为O(n^2) 但是FIFS的平均寻道长度明显高于其他两个算法，弃用。  处于偏小位置的时候可以使用SCAN算法中移动臂向内访问算法和SSTF算法，这两个算法在这种情况时间复杂度相同，平均寻道长度也接近。  处于偏大位置时相反。  （2）当磁道初始位置在整体访问磁道数据的中间位置的时候，FIFS算法时间复杂度最小，平均寻道长度与其他两种算法相差不多，所以可以使用FIFS算法  对于三种算法的稳定性来看：  FIFS>SCAN>SSTF  （4）若所有硬盘全部设计成固态硬盘，哪个磁盘调度算法最合适？  因为电子硬盘读取速度极快而且不涉及磁盘调度臂的问题，所以应该使用时间复杂度最低的FIFS算法，该算法时间复杂度为：O(n);  2．分析、总结  设备驱动程序  设备驱动程序是内核和硬件设备之间的接口，设备驱动程序屏蔽硬件细节，且设备被映射成特殊的文件进行处理。每个设备都对应一个文件名，在内核中也对应一个索引节点，应用程序可以通过设备的文件名来访问硬件设备。Linux 为文件和设备提供了一致性的接口，用户操作设备文件和操作普通文件类似。例如通过 open() 函数可打开设备文件，建立起应用程序与目标程序的连接；之后，可以通过read() 、write()等常规文件函数对目标设备进行数据传输操作。设备驱动程序封装了如何控制设备的细节，它们可以使用和操作文件相同的、标准的系统调用接口来完成打开、关闭、读写和I/O 控制等操作，而驱动程序的主要任务也就是要实现这些系统调用函数。设备驱动程序是一些函数和数据结构的集合，这些函数和数据结构是为实现设备管理的一个简单接口。操作系统内核使用这个接口来请求驱动程序对设备进行I/O操作。  字符驱动程序相关数据结构  字符设备是以字节为单位逐个进行I/O 操作的设备，不经过系统的 I/O 缓冲区，所以需要管理自己的缓冲区结构。  在字符设备驱动程序中，主要涉及 3 个重要的内核数据结构，分别是file\_operations、file和inode。当用户访问设备文件时，每个文件的file结构都有一个索引节点inode与之对应。在内核的inode结构中，有一个名为i\_fop成员，其类型为file\_operations。file\_operations定义文件的各种操作，用户对文件进行诸如open、close、read、write等操作时，Linux 内核将通过file\_operations结构访问驱动程序提供的函数。内核通过这3个数据结构的关联，将用户对设备文件的操作转换为对驱动程序相关函数的调用。  file 代表一个打开了的文件。它由内核在使用 open()函数时建立，并传递给该文件上进行操作的所有函数，直到最后的close()函数。当文件的所有操作结束后，内核会释放该数据结构。  3．体会  了解了磁盘调度的目的是要尽可能降低磁盘的寻道时间，提高性能，相比于之前的实验，难度很大。 | | | |
| 七、指导教师评语及成绩  教师签名：  **年 月 日**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 优秀 | 良好 | 中等 | 及格 | 不及格 | | | | |