**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**  2101

**姓 名**  梅英豪

**学 号**  20212203341

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）  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：  这个字符设备驱动程序是一个简单的示例，演示了创建字符设备、打开设备文件、读取数据和关闭设备文件的基本操作。  首先，在 chardev.c 中定义了驱动程序所需的函数和数据结构。其中，device\_open 函数用于打开设备文件并进行必要的初始化。device\_release 函数用于释放设备资源。device\_read 函数用于从设备中读取数据。device\_write 函数用于向设备中写入数据。这些函数是根据字符设备的操作需求进行实现的。  在 init\_chardev\_module 函数中，调用了 register\_chrdev 函数来注册字符设备，并获得分配的主设备号。然后通过 printk 函数打印相关信息。  在 exit\_chardev\_module 函数中，调用了 unregister\_chrdev 函数来注销已注册的字符设备。  Makefile 文件中定义了编译和清理模块的规则，通过执行 make 命令可以编译出 chardev.ko 模块文件。  在使用 insmod 命令安装驱动模块时，可能会遇到权限问题或已存在模块的错误。需要确保以 root 权限或使用 sudo 命令来执行安装和卸载模块的操作。  在创建字符设备文件时，使用 mknod 命令指定主设备号和次设备号来创建设备文件。  最后，通过编写 test.c 程序来测试字符设备驱动程序。test.c 中使用 open 函数打开字符设备文件，使用 read 函数读取设备文件中的数据，并使用 printf 函数打印读取的数据。  需要注意的是，这个驱动程序只是一个简单的示例，没有处理所有的错误和异常情况，也没有实现字符设备的全部功能。在实际开发中，需要根据具体需求进行适当的修改和扩展，以确保驱动程序的正确性和可靠性。  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：  （二）**磁盘调度算法**（40分）  实验代码：  (1)先进先出算法FIFO：  // 先进先出算法（FIFO）  void fifo(int disk\_size, int start, int requests[], int num\_requests)  {  int current = start;  int total\_movement = 0;  printf("FIFO:\n");  printf("起始位置：%d\n", start);  for (int i = 0; i < num\_requests; i++)  {  int distance = abs(current - requests[i]);  total\_movement += distance;  current = requests[i];  printf("移动到磁道 %d，移动距离：%d\n", current, distance);  }  printf("总移动距离：%d\n\n", total\_movement);  }  (2)最短服务时间优先算法SSTF：  // 最短服务时间优先算法（SSTF）  void sstf(int disk\_size, int start, int requests[], int num\_requests)  {  int current = start;  int total\_movement = 0;  int completed[num\_requests]; // 已完成的请求  int num\_completed = 0;  printf("SSTF:\n");  printf("起始位置：%d\n", start);  while (num\_completed < num\_requests)  {  int shortest\_distance = disk\_size + 1; // 初始化为比磁道数大的值  int shortest\_index = -1;  for (int i = 0; i < num\_requests; i++)  {  if (completed[i]) // 跳过已完成的请求  continue;  int distance = abs(current - requests[i]);  if (distance < shortest\_distance)  {  shortest\_distance = distance;  shortest\_index = i;  }  }  total\_movement += shortest\_distance;  current = requests[shortest\_index];  completed[shortest\_index] = 1;  num\_completed++;  printf("移动到磁道 %d，移动距离：%d\n", current, shortest\_distance);  }  printf("总移动距离：%d\n\n", total\_movement);  }  (3)扫描算法SCAN：  // 扫描算法（SCAN）  void scan(int disk\_size, int start, int requests[], int num\_requests)  {  int current = start;  int total\_movement = 0;  int direction = 1; // 扫描方向，1表示向外，-1表示向内  printf("SCAN:\n");  printf("起始位置：%d\n", start);  while (1)  {  int shortest\_distance = disk\_size + 1; // 初始化为比磁道数大的值  int shortest\_index = -1;  for (int i = 0; i < num\_requests; i++)  {  if (requests[i] == current)  {  shortest\_index = i;  break;  }  int distance = abs(current - requests[i]);  if (distance < shortest\_distance && (direction \* (requests[i] - current)) >= 0)  {  shortest\_distance = distance;  shortest\_index = i;  }  }  if (shortest\_index == -1) // 如果没有找到更近的请求，则改变扫描方向  {  direction = -direction;  continue;  }  total\_movement += shortest\_distance;  current = requests[shortest\_index];  requests[shortest\_index] = -1; // 标记为已完成  printf("移动到磁道 %d，移动距离：%d\n", current, shortest\_distance);  int all\_completed = 1; // 检查是否所有请求都已完成  for (int i = 0; i < num\_requests; i++)  {  if (requests[i] != -1)  {  all\_completed = 0;  break;  }  }  if (all\_completed)  break;  }  printf("总移动距离：%d\n\n", total\_movement);  }  (4)循环扫描算法C-SCAN：  void cscan(int disk\_size, int start, int requests[], int num\_requests)  {  int current = start;  int total\_movement = 0;  printf("C-SCAN:\n");  printf("起始位置：%d\n", start);  while (1)  {  int shortest\_distance = disk\_size + 1; // 初始化为比磁道数大的值  int shortest\_index = -1;  for (int i = 0; i < num\_requests; i++)  {  if (requests[i] == current)  {  shortest\_index = i;  printf("break\n");  break;  }  int distance = abs(current - requests[i]);  if (distance < shortest\_distance && requests[i] > current)  {  printf("break111111111\n");  shortest\_distance = distance;  shortest\_index = i;  }  }    total\_movement += shortest\_distance;  current = requests[shortest\_index];  requests[shortest\_index] = -1; // 标记为已完成  printf("移动到磁道 %d，移动距离：%d\n", current, shortest\_distance);  int all\_completed = 1; // 检查是否所有请求都已完成  for (int i = 0; i < num\_requests; i++)  {  if (requests[i] != -1)  {  all\_completed = 0;  break;  }  }  if (all\_completed)  break;  // 添加终止条件，例如等待用户输入命令  printf("输入 'q' 停止C-SCAN算法运行：");  char input;  scanf(" %c", &input);  if (input == 'q')  break;  }  printf("总移动距离：%d\n\n", total\_movement);  }  (5)主程序：  int main()  {  int disk\_size = 200; // 磁道数  int start = 50; // 起始位置  int requests[] = {60, 70, 80, 90, 100, 110, 120, 130, 140, 150}; // 请求序列  int num\_requests = sizeof(requests) / sizeof(requests[0]);  fifo(disk\_size, start, requests, num\_requests);  sstf(disk\_size, start, requests, num\_requests);  scan(disk\_size, start, requests, num\_requests);  cscan(disk\_size, start, requests, num\_requests);  return 0;  }  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：  先进先出算法（FIFO）：  先进先出算法是最简单的磁盘调度算法，它按照请求的先后顺序进行调度。  优点：实现简单，适用于低负载情况。  缺点：无法考虑磁道间距离，可能导致长期等待时间长的请求。  最短服务时间优先算法（SSTF）：  最短服务时间优先算法选择距离当前磁头位置最近的请求进行调度。  优点：相比FIFO算法，能够减少平均寻道时间，提高系统性能。  缺点：可能出现饥饿现象，即某些请求长期等待，因为始终有更近的请求被优先服务。  扫描算法（SCAN）：  扫描算法按照一个方向（例如向外）扫描并服务请求，直到到达磁道末端，然后改变方向继续扫描。  优点：能够避免饥饿现象，公平地服务请求，且可以减少平均寻道时间。  缺点：可能导致较长的平均等待时间，因为在改变方向之前需要扫描整个磁道。  循环扫描算法（C-SCAN）：  循环扫描算法类似于扫描算法，但它在到达磁道末端时直接返回到磁道0，而不改变方向。  优点：避免了在改变方向时的长时间等待，可以更加均匀地服务请求。  缺点：可能出现较长的等待时间，因为请求必须等待磁头重新到达磁道0。  综上所述，不同的磁盘调度算法在不同的场景下具有各自的优势和局限性。选择适当的算法取决于特定的应用需求和性能要求。需要权衡平均寻道时间、响应时间、公平性等因素，以及考虑负载情况和请求分布等因素，以获得最佳的性能和用户体验。  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。  操作系统中的进程调度算法：我解释了常见的进程调度算法，包括先来先服务（FCFS）、最短作业优先（SJF）、轮转调度（Round Robin）和优先级调度。  磁盘调度算法：你询问了四种磁盘调度算法，分别是先进先出算法（FIFO）、最短服务时间优先算法（SSTF）、扫描算法（SCAN）和循环扫描算法（C-SCAN）。我对每种算法进行了简要的分析，包括其优点和缺点。  C 语言编程问题：你向我展示了一段代码并询问了相关编译错误。我指出了代码中存在的问题，并提供了可能的解决方案和建议。  对四种磁盘调度算法的分析：你要求对四种磁盘调度算法进行分析，我总结了每种算法的特点、优点和缺点。  （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。  字符设备和块设备是在操作系统中用于访问硬件设备的两种主要类型。它们在实现驱动程序的过程中存在一些异同点，下面我将对它们进行分析。  异同点：  a) 数据访问单位：最显著的区别是字符设备以字节为单位进行数据访问，而块设备以块（通常是512字节或更大）为单位进行数据访问。这意味着在字符设备中，每次读取或写入的最小单位是一个字节，而在块设备中，每次读取或写入的最小单位是一个块。  b) 缓存机制：块设备通常具有硬件缓存（例如磁盘缓存），用于加速数据的读取和写入操作。而字符设备通常没有硬件缓存，每次读取或写入都直接从设备传输数据。  c) 寻址方式：块设备可以通过块号来定位要读取或写入的数据块，而字符设备一般是按顺序访问数据。  d) 文件系统支持：块设备常用于构建文件系统，因为文件系统需要随机访问和块级别的操作。而字符设备通常用于串行设备（如终端、串口等），不直接用于文件系统。  实现过程中的异同点：  a) 驱动程序接口：字符设备和块设备的驱动程序接口有所不同。字符设备的驱动程序接口主要包括 open()、read()、write()、close() 等系统调用函数，用于打开、读取、写入和关闭设备。块设备的驱动程序接口则更复杂，通常包括块的读写操作、设备的初始化和释放等函数。  b) 数据处理：字符设备驱动程序通常对数据进行逐字节处理，可以直接从设备读取或写入一个字节。而块设备驱动程序处理数据的单位是块，需要进行块的缓存、管理和传输。  c) 设备访问方式：字符设备驱动程序通常以流式方式（stream）访问设备，即逐字节顺序读取或写入。块设备驱动程序则可以通过块缓冲区进行数据的批量读取和写入，提高效率。  d) 中断处理：块设备驱动程序通常需要处理磁盘驱动器的中断，以便在数据传输完成时及时通知系统。而字符设备驱动程序一般不涉及中断处理。  总的来说，字符设备和块设备在实现驱动程序的过程中存在一些明显的异同点。它们的数据访问单位、缓存机制、寻址方式以及驱动程序接口和数据处理方式都有所不同。这些差异主要是由于字符设备和块设备的应用领域和数据访问特性不同所导致的。  （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？  对四种算法进行时间复杂度分析可以帮助我们了解它们的效率。下面是对每个算法的时间复杂度分析和效率比较：  先进先出算法（FIFO）：  时间复杂度：O(n)  效率：FIFO算法是一种简单的调度算法，它按照请求的先后顺序进行服务，没有考虑请求的大小或位置。因此，FIFO算法的效率较低，可能导致长时间等待和延迟。  最短服务时间优先算法（SSTF）：  时间复杂度：O(n^2)  效率：SSTF算法根据当前位置选择最近的请求进行服务，尽量减少寻道时间。在最坏情况下，当所有请求都集中在同一位置附近时，SSTF算法的效率可能较低。但在一般情况下，SSTF算法能够较好地提高磁盘访问的效率。  扫描算法（SCAN）：  时间复杂度：O(n^2)  效率：SCAN算法从当前位置开始，按照一个方向依次服务请求，直到达到最边缘，然后改变方向继续服务。SCAN算法相对于FIFO和SSTF算法更加公平，可以避免某些请求长时间等待。然而，如果请求分布不均匀，可能会导致一些请求长时间等待或延迟。  循环扫描算法（C-SCAN）：  时间复杂度：O(n^2)  效率：C-SCAN算法类似于SCAN算法，但它在到达最边缘后直接返回起始位置继续服务，而不改变方向。这样可以确保所有请求都得到平均的服务时间，减少长时间等待。C-SCAN算法在一般情况下能够提供较好的磁盘访问效率。  综上所述，各个算法的效率从高到低的排序为：C-SCAN > SCAN > SSTF > FIFO。但需要注意的是，具体的效率还受到请求的分布情况和磁盘访问模式的影响，不同的应用场景可能对算法的效率有不同的要求。因此，在选择算法时需要根据具体情况进行综合考虑。  （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？  当所有硬盘都设计成固态硬盘（SSD）时，最合适的磁盘调度算法是扫描算法（SCAN）或循环扫描算法（C-SCAN）。  固态硬盘（SSD）相较于传统的机械硬盘（HDD）具有较快的访问速度和较低的寻道延迟。SSD的访问时间主要受到读取/写入操作的延迟和数据块的分布情况的影响。  扫描算法（SCAN）和循环扫描算法（C-SCAN）适用于固态硬盘的原因如下：  最小化寻道延迟：固态硬盘没有机械部件需要移动，因此寻道延迟很小。SCAN和C-SCAN算法按照一个方向连续服务请求，可以最小化寻道时间并提高磁盘的吞吐量。  均衡磁盘负载：SSD的读取/写入操作相对均衡，不像机械硬盘有物理位置的限制。SCAN和C-SCAN算法能够平均分配磁盘的服务时间，减少某些请求长时间等待。  虽然固态硬盘在访问速度方面已经很快，但选择SCAN或C-SCAN算法可以进一步优化磁盘访问的效率，提高系统的整体性能。  2．分析、总结  磁盘调度算法的选择依赖于具体的应用场景和硬件特性。不同的算法在不同的情况下可能表现出更好的性能。  先进先出算法（FIFO）是一种简单且易于实现的算法，但它可能导致长等待时间和低吞吐量。  最短服务时间优先算法（SSTF）能够最小化寻道延迟，提供较低的响应时间和较高的吞吐量。然而，它可能导致部分请求长时间等待（饥饿现象）。  扫描算法（SCAN）和循环扫描算法（C-SCAN）适用于固态硬盘（SSD），可以最小化寻道延迟，提高磁盘的吞吐量，并平衡磁盘的负载。  在分析磁盘调度算法的效率时，需要考虑算法的时间复杂度、平均等待时间、平均寻道时间等指标。具体的效率取决于应用场景、请求负载的特点和硬件的特性。  综上所述，选择最合适的磁盘调度算法应该综合考虑系统的性能需求、硬件特性和应用场景，以达到最佳的性能和用户体验。  3．体会  通过分析不同的磁盘调度算法，我意识到每个算法都有其独特的优势和限制。先进先出算法简单直观，但可能导致性能下降；最短服务时间优先算法能够提供较低的延迟，但存在饥饿问题；扫描算法和循环扫描算法适用于固态硬盘，能够平衡负载和提高吞吐量。  在选择磁盘调度算法时，需要综合考虑硬件设备、应用需求和性能指标，以找到最适合的算法。这要求我们对系统的特性和工作原理有深入的了解，并灵活应用各种算法来满足特定需求。  此外，磁盘调度算法的效率不仅仅取决于算法本身，还受到请求负载和硬件设备的影响。因此，在实际应用中，我们需要不断监测和评估系统性能，以优化磁盘访问并提高整体效率。  总而言之，磁盘调度算法是操作系统中一个重要而复杂的领域，需要综合考虑多个因素来做出决策。通过对这些算法的学习和分析，我对操作系统的设计和性能优化有了更深入的认识，并意识到在实际应用中需要权衡各种因素来实现最佳的磁盘调度策略。 | | | |
| 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 | | | | |