西安交通大学

**操作系统专题实验报告**

班级：计算机86

学号：2182212573

姓名：常傲

2021年1月9日

**目 录**

[1openEuler系统环境实验 1](#_Toc350786740)

[1.1实验目的 1](#_Toc350786741)

[1.2实验内容 1](#_Toc350786742)

[1.3实验思想（或流程图） 1](#_Toc350786743)

[1.4实验步骤 1](#_Toc350786744)

[1.5测试数据设计 1](#_Toc350786745)

[1.6程序运行初值及运行结果分析 1](#_Toc350786746)

[1.7实验总结 1](#_Toc350786747)

[1.7.1实验中的问题与解决过程 1](#_Toc350786748)

[1.7.2实验收获 1](#_Toc350786749)

[1.7.3意见与建议 1](#_Toc350786750)

[1.8附件 1](#_Toc350786751)

[1.8.1附件1 程序 1](#_Toc350786752)

[1.8.2附件3 Readme 1](#_Toc350786754)

[2 模块编程与字符设备驱动 1](#_Toc350786755)

[2.1实验目的 1](#_Toc350786756)

[2.2实验内容 1](#_Toc350786757)

[2.2.1 模块编程 1](#_Toc350786758)

[2.2.2 字符设备驱动 1](#_Toc350786759)

# 1 openEuler系统环境实验

## 1.1实验目的

本实验旨在熟悉基于Kunpeng架构弹性云服务器ECS上openEuler操作系统基本系统环境。通过运行shell命令查看系统信息以达到了解openEuler操作系统的目的。另外运行简单的C程序和汇编程序以了解基于ARMv8-64的开发环境。

## 1.2实验内容

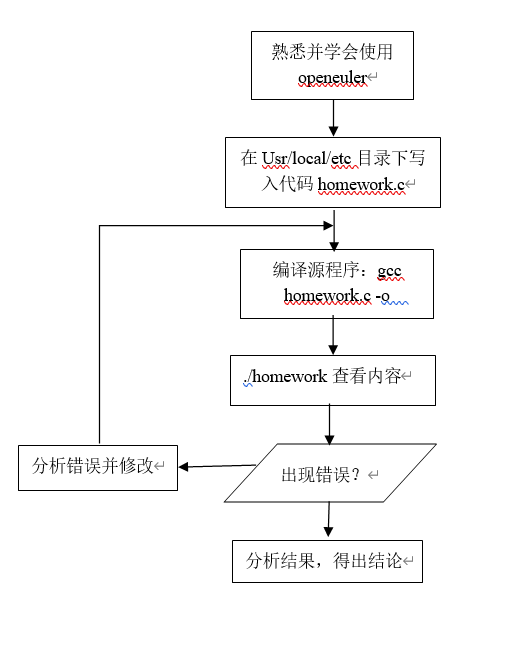
1、熟悉操作命令、编辑、编译、运行程序。完成操作系统原理课程教材P117作业3.4的运行验证。

2、在个人PC机上完成一下实验：

\*进程通信与软中断

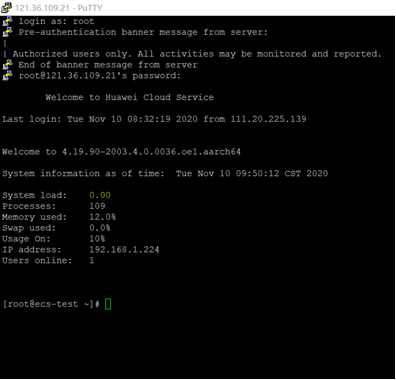
\*内存管理

## 1.3实验思想（或流程图）



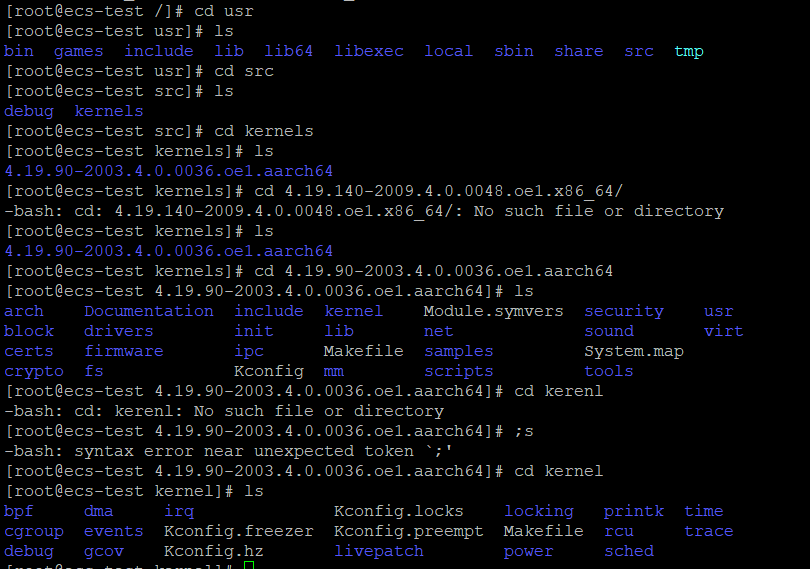
## 1.4实验步骤

1、打开putty，输入用户名及密码，进入操作系统界面。



**第一次实验**

进行一系列的基本操作熟悉openeuler操作系统的基本功能。



在Usr/local/etc目录下创建C语言文件homework.c，写入代码

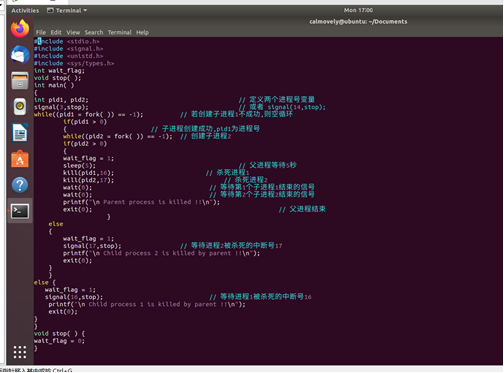


输出如下：

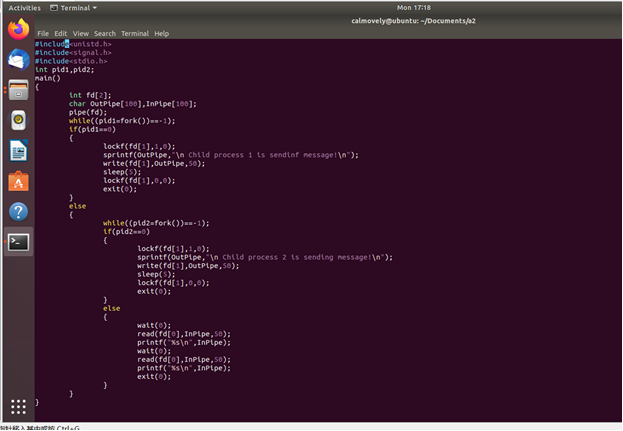
**第二次实验**

将进程的软中断通信和管道通信分别写在.Documents/a1和Documents/a2

软中断通信：



管道通信：



内存管理：

#include<stdio.h>

#include<stdlib.h>

#define \_CRT\_SECURE\_NO\_WARNINGS

#define max\_mem 2048;

typedef struct list

{

struct list \*next;

int size;

struct list \*pre;

int state;

int addr;

}list;

struct list \*head;

struct list \*tail;

void initial()

{

head = (struct list\*)malloc(sizeof(struct list));

tail = (struct list\*)malloc(sizeof(struct list));

head->pre = NULL;

head->next = tail;

tail->pre = head;

tail->next = NULL;

head->state = 1;

tail->state = 0;

head->addr = 0;

tail->addr = 1;

head->size = 0;

tail->size = max\_mem;

}

void print()

{

struct list \*p = head;

while (p != tail)

{

printf("%d \t %d \t %d\n", p->addr, p->size, p->state);

p = p->next;

}

}

void ff(int res)

{

struct list \*p = head;

while (p)

{

if (p->state == 0 && p->size == res)

{

p->state = 1;

return;

}

else if (p->state == 0 && p->size > res)

{

struct list \*t = (struct list\*)malloc(sizeof(struct list));

t->pre = p->pre;

t->next = p;

p->pre->next = t;

p->pre = t;

t->size = res;

p->size -= res;

t->addr = p->addr;

t->state = 1;

while (p)

{

p->addr++;

p = p->next;

}

return;

}

else

{

p = p->next;

}

}

printf("Error:out of range!");

}

void bf(int res)

{

int ch;

struct list \*temp = (struct list\*)malloc(sizeof(struct list));

temp->size = res;

temp->state = 1;

struct list \*p = head->next;

struct list \*q = NULL;

while (p)

{

if (p->state == 0 && (p->size >= res))

{

if (q == NULL)

{

q = p;

ch = p->size - res;

}

else if (q->size > p->size)

{

q = p;

ch = p->size - res;

}

}

p = p->next;

}

if (q == NULL)

{

return;

}

else if (q->size == res)

{

q->state = 1;

}

else

{

temp->pre = q->pre;

temp->next = q;

temp->addr = q->addr;

q->pre->next = temp;

q->pre = temp;

q->size = ch;

while (q)

{

q->addr += 1;

q = q->next;

}

return;

}

printf("Rrror: out of range!\n");

return;

}

//最差适应算法

void wf(int res)

{

int ch; //记录最大剩余空间

struct list \*temp = (struct list\*)malloc(sizeof(struct list));

temp->size = res;

temp->state = 1;

list \*p = head->next;

list \*q = NULL; //记录最佳插入位置

while (p) //初始化最大空间和最佳位置

{

if (p->state == 0 && (p->size >= res))

{

if (q == NULL)

{

q = p;

ch = p->size - res;

}

else if (q->size < p->size)

{

q = p;

ch = p->size - res;

}

}

p = p->next;

}

if (q == NULL) return;//没有找到空闲块

else if (q->size == res)

{

q->state = 1;

return;

}

else

{

temp->pre = q->pre;

temp->next = q;

temp->addr = q->addr;

q->pre->next = temp;

q->pre = temp;

q->size = ch;

while (q)

{

q->addr += 1;

q = q->next;

}

return;

}

printf("Error: out of range!\n");

return;

}

void myremove(int n)

{

struct list \*p = head;

while (p->addr != n)

{

p = p->next;

}

p->state = 0;

if (p->next == tail)

{

p->pre->next = tail;

tail->pre = p->pre;

tail->addr--;

}

if (p->pre->state == 0)

{

p->size += p->pre->size;

p->pre = p->pre->pre;

p->pre->next = p;

p->addr--;

}

if (p->next->state == 0 && p->next != tail)

{

p->size += p->next->size;

p->next = p->next->next;

p->next->pre = p;

p = p->next;

while (p)

{

p->addr--;

p = p->next;

}

}

}

void myc(int c, int res)

{

switch (c)

{

case 1:

ff(res); break;

case 2:

bf(res); break;

case 3:

wf(res); break;

default:

break;

}

}

int main()

{

initial();

int choice;

int res = 0;

int c = 0;

printf("选择一下三种方法：\n1)FF算法 2)BF算法 3)WF算法\n");

scanf("%d", &c);

while (1)

{

printf("------------------------------------------\n");

printf("1)增加内存 2)删除内存 3)结束\n");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("选择增加内存的大小\n");

scanf("%d", &res);

myc(c, res);

print();

break;

case 2:

printf("选择删除的序列号\n");

scanf("%d", &res);

myremove(res);

print();

break;

case 3:

return 0;

break;

default:

break;

}

}

printf("Program Exit.\n");

return 0;

}

## 1.5测试数据设计

主要在实验内存管理中涉及到了输入与输出，实验输入如下：

增加内存，输入10

增加内存，输入20

增加内存，输入30

删除内存10

删除内存20

删除内存30

增加内存，输入10

增加内存，输入100

增加内存，输入300

增加

删除内存，30

## 1.6程序运行初值及运行结果分析

结果与实验相符，可以做到通过链表来实现内存的增删查改。

## 1.7实验总结

这次实验让我明白了操作系统的内存管理的进行，对内存的BF,WF,FF三种算法有了更加深刻的认识，也明白了各种算法的特性和碎片处理的关系。虽然实验没办法做到尽善尽美，仍然有很多瑕疵，结果可能不尽如人意，但我已尽我所能。

### 1.7.1实验中的问题与解决过程

### 1.问题描述

### 2.解决过程：（网址，参考资料），具体解决方法

### 1.7.2实验收获

### 1.7.3意见与建议

## 1.8附件

### 1.8.1附件1 程序

内存管理：

#include<stdio.h>

#include<stdlib.h>

#define \_CRT\_SECURE\_NO\_WARNINGS

#define max\_mem 2048;

typedef struct list

{

struct list \*next;

int size;

struct list \*pre;

int state;

int addr;

}list;

struct list \*head;

struct list \*tail;

void initial()

{

head = (struct list\*)malloc(sizeof(struct list));

tail = (struct list\*)malloc(sizeof(struct list));

head->pre = NULL;

head->next = tail;

tail->pre = head;

tail->next = NULL;

head->state = 1;

tail->state = 0;

head->addr = 0;

tail->addr = 1;

head->size = 0;

tail->size = max\_mem;

}

void print()

{

struct list \*p = head;

while (p != tail)

{

printf("%d \t %d \t %d\n", p->addr, p->size, p->state);

p = p->next;

}

}

void ff(int res)

{

struct list \*p = head;

while (p)

{

if (p->state == 0 && p->size == res)

{

p->state = 1;

return;

}

else if (p->state == 0 && p->size > res)

{

struct list \*t = (struct list\*)malloc(sizeof(struct list));

t->pre = p->pre;

t->next = p;

p->pre->next = t;

p->pre = t;

t->size = res;

p->size -= res;

t->addr = p->addr;

t->state = 1;

while (p)

{

p->addr++;

p = p->next;

}

return;

}

else

{

p = p->next;

}

}

printf("Error:out of range!");

}

void bf(int res)

{

int ch;

struct list \*temp = (struct list\*)malloc(sizeof(struct list));

temp->size = res;

temp->state = 1;

struct list \*p = head->next;

struct list \*q = NULL;

while (p)

{

if (p->state == 0 && (p->size >= res))

{

if (q == NULL)

{

q = p;

ch = p->size - res;

}

else if (q->size > p->size)

{

q = p;

ch = p->size - res;

}

}

p = p->next;

}

if (q == NULL)

{

return;

}

else if (q->size == res)

{

q->state = 1;

}

else

{

temp->pre = q->pre;

temp->next = q;

temp->addr = q->addr;

q->pre->next = temp;

q->pre = temp;

q->size = ch;

while (q)

{

q->addr += 1;

q = q->next;

}

return;

}

printf("Rrror: out of range!\n");

return;

}

//最差适应算法

void wf(int res)

{

int ch; //记录最大剩余空间

struct list \*temp = (struct list\*)malloc(sizeof(struct list));

temp->size = res;

temp->state = 1;

list \*p = head->next;

list \*q = NULL; //记录最佳插入位置

while (p) //初始化最大空间和最佳位置

{

if (p->state == 0 && (p->size >= res))

{

if (q == NULL)

{

q = p;

ch = p->size - res;

}

else if (q->size < p->size)

{

q = p;

ch = p->size - res;

}

}

p = p->next;

}

if (q == NULL) return;//没有找到空闲块

else if (q->size == res)

{

q->state = 1;

return;

}

else

{

temp->pre = q->pre;

temp->next = q;

temp->addr = q->addr;

q->pre->next = temp;

q->pre = temp;

q->size = ch;

while (q)

{

q->addr += 1;

q = q->next;

}

return;

}

printf("Error: out of range!\n");

return;

}

void myremove(int n)

{

struct list \*p = head;

while (p->addr != n)

{

p = p->next;

}

p->state = 0;

if (p->next == tail)

{

p->pre->next = tail;

tail->pre = p->pre;

tail->addr--;

}

if (p->pre->state == 0)

{

p->size += p->pre->size;

p->pre = p->pre->pre;

p->pre->next = p;

p->addr--;

}

if (p->next->state == 0 && p->next != tail)

{

p->size += p->next->size;

p->next = p->next->next;

p->next->pre = p;

p = p->next;

while (p)

{

p->addr--;

p = p->next;

}

}

}

void myc(int c, int res)

{

switch (c)

{

case 1:

ff(res); break;

case 2:

bf(res); break;

case 3:

wf(res); break;

default:

break;

}

}

int main()

{

initial();

int choice;

int res = 0;

int c = 0;

printf("选择一下三种方法：\n1)FF算法 2)BF算法 3)WF算法\n");

scanf("%d", &c);

while (1)

{

printf("------------------------------------------\n");

printf("1)增加内存 2)删除内存 3)结束\n");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("选择增加内存的大小\n");

scanf("%d", &res);

myc(c, res);

print();

break;

case 2:

printf("选择删除的序列号\n");

scanf("%d", &res);

myremove(res);

print();

break;

case 3:

return 0;

break;

default:

break;

}

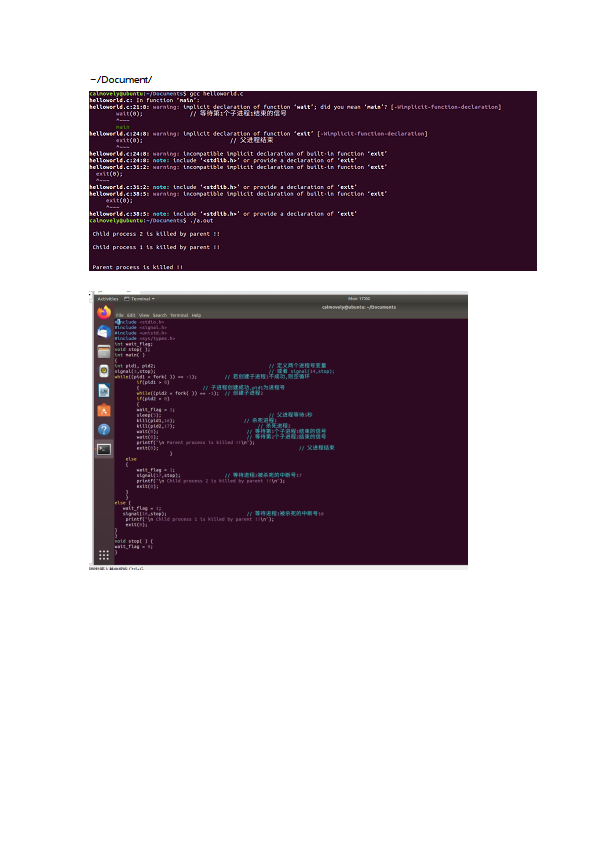
}

printf("Program Exit.\n");

return 0;

}

### 1.8.2附件2 Readme



# 2 模块编程与字符设备驱动

## 2.1实验目的

理解LINUX字符设备驱动程序的基本原理；

掌握字符设备的驱动运作机制；

学会编写字符设备驱动程序

## 2.2实验内容

### 2.2.1 模块编程

（1）一个字符(char)设备是一种可以当作一个字节流来存取的设备(如同一个文件); 一个字符驱动负责实现这种行为。这样的驱动常常至少实现open, close, read, 和 write 系统调用.

（2）字符设备通过文件系统结点来存取, 例如/dev/tty1和/dev/lp0。在一个字符设备和一个普通文件之间唯一有关的不同就是,你经常可以在普通文件中移来移去, 但是大部分字符设备仅仅是数据通道, 你只能顺序存取

### 2.2.2 字符设备驱动

编写一个简单的字符设备驱动程序，以内核空间模拟字符设备，完成对该设备的打开，读写和释放操作， 并编写聊天程序实现对该设备的同步和互斥操作。

## 2.3实验总结

本次实验涉及到了模块的调动和设备的驱动，虽然看不太懂意思但是也做的马马虎虎。

### 2.3.1实验中的问题与解决过程

### 1.问题描述

实验中如何升级用户权限和使用root用户进行内核访问；如何通过信号量pv操作来实现聊天功能。

### 2.解决过程：

通过百度知道了使用su达到root用户，通过chmod 777 实现权限升级。最后驱动模块，使用两个terminal实现互斥访问，达到聊天的目的。

### 2.3.2实验收获

这一次，是操作系统的最后一次实验了，通过这次实验课我对操作系统的喜好更深刻一步，在前三次的基础上我掌握了Linux设备和模块系统的基本原理、结构和实现方法，并且掌握了Linux设备的建立、打开、读/写、执行、属性等系统调用的使用，学会设计简单的模块系统并实现一组操作。虽然实验部分的学习结束了，但是我深知自己学习操作系统的道路还没有停止，我会更加努力，知识结合实践，完成更好的成绩。

### 2.3.3意见与建议

希望实验能从头讲起，说的细致一些，最好课时多一点。

**实验结果：**



## 2.4附件include <linux/init.h> #include <linux/kernel.h> #include <linux/module.h> //original,syscall 78 function: gettimeofday // new syscall 78 function: print "No 78 syscall has changed to hello" and return a+b #define sys\_No 78 unsigned long old\_sys\_call\_func; unsigned long p\_sys\_call\_table=0xc0361860; // find in /boot/System.map-'uname -r' asmlinkage int hello(int a,int b) //new function { printk("No 78 syscall has changed to hello"); return a+b; } void modify\_syscall(void) { unsigned long \*sys\_call\_addr; sys\_call\_addr=(unsigned long \*)(p\_sys\_call\_table+sys\_No\*4); old\_sys\_call\_func=\*(sys\_call\_addr); \*(sys\_call\_addr)=(unsigned long)&hello; // point to new function } void restore\_syscall(void) { unsigned long \*sys\_call\_addr; sys\_call\_addr=(unsigned long \*)(p\_sys\_call\_table+sys\_No\*4); \*(sys\_call\_addr)=old\_sys\_call\_func; // point to original function } static int mymodule\_init(void) { modify\_syscall(); return 0; } static void mymodule\_exit(void) { restore\_syscall(); } module\_init(mymodule\_init); module\_exit(mymodule\_exit); MODULE\_LICENSE("GPL"); 附件 2： ifneq ($(KERNELRELEASE),) obj-m := modify\_syscall.o #obj-m指编译成外部模块 else KERNELDIR := /lib/modules/$(shell uname -r)/build #定义一个变量，指向内 核目录 PWD := $(shell pwd) modules: $(MAKE) -C $(KERNELDIR) M=$(PWD) modules #编译内核模块 endif clean: $(MAKE) -C $(KERNELDIR) M=$(PWD) clean 附件 3： #include<linux/module.h> #include<linux/init.h> #include<linux/fs.h> #include<asm/uaccess.h> #include<linux/wait.h> #include<linux/semaphore.h> #include <linux/uaccess.h> #include <linux/sched.h> #include <linux/cdev.h> #include <linux/types.h> #include<linux/kdev\_t.h> #include <linux/device.h> #define MAXNUM 100 #define MAJOR\_NUM 456 //主设备号 ，没有被使用 struct Scull\_Dev{ struct cdev devm; //字符设备 struct semaphore sem; //信号量，实现读写时的 PV 操作 wait\_queue\_head\_t outq; //等待队列，实现阻塞操作 int flag; //阻塞唤醒标志char buffer[MAXNUM+1]; //字符缓冲区 char buffer[MAXNUM+1]; //字符缓冲区 char \*rd,\*wr,\*end; //读，写，尾指针 }; struct Scull\_Dev globalvar; static struct class \*my\_class; int major=MAJOR\_NUM; static ssize\_t globalvar\_read(struct file \*,char \*,size\_t ,loff\_t \*); static ssize\_t globalvar\_write(struct file \*,const char \*,size\_t ,loff\_t \*); static int globalvar\_open(struct inode \*inode,struct file \*filp); static int globalvar\_release(struct inode \*inode,struct file \*filp); struct file\_operations globalvar\_fops = { .read=globalvar\_read, .write=globalvar\_write, .open=globalvar\_open, .release=globalvar\_release, }; static int globalvar\_init(void) { int result = 0; int err = 0; dev\_t dev = MKDEV(major, 0); 操作系统专题实验报告 2021 年 1 月 9 日 35 if(major) { //静态申请设备编号 result = register\_chrdev\_region(dev, 1, "charmem"); } else { //动态分配设备号 result = alloc\_chrdev\_region(&dev, 0, 1, "charmem"); major = MAJOR(dev);} if(result < 0) return result; cdev\_init(&globalvar.devm, &globalvar\_fops); globalvar.devm.owner = THIS\_MODULE; err = cdev\_add(&globalvar.devm, dev, 1); if(err) printk(KERN\_INFO "Error %d adding char\_mem device", err); else{ printk("globalvar register success\n"); sema\_init(&globalvar.sem,1); //初始化信号量 init\_waitqueue\_head(&globalvar.outq); //初始化等待队列 globalvar.rd = globalvar.buffer; //读指针 globalvar.wr = globalvar.buffer; //写指针 globalvar.end = globalvar.buffer + MAXNUM;//缓冲区尾指针 globalvar.flag = 0; // 阻塞唤醒标志置 0 } my\_class = class\_create(THIS\_MODULE, "chardev0"); device\_create(my\_class, NULL, dev, NULL, "chardev0"); return 0; } static int globalvar\_open(struct inode \*inode,struct file \*filp) { try\_module\_get(THIS\_MODULE);//模块计数加一 printk("This chrdev is in open\n"); return(0); } static int globalvar\_release(struct inode \*inode,struct file \*filp) { module\_put(THIS\_MODULE); //模块计数减一 printk("This chrdev is in release\n"); return(0); } static void globalvar\_exit(void) { device\_destroy(my\_class, MKDEV(major, 0)); class\_destroy(my\_class); cdev\_del(&globalvar.devm); unregister\_chrdev\_region(MKDEV(major, 0), 1); //注销设备 } static ssize\_t globalvar\_read(struct file \*filp,char \*buf,size\_t len,loff\_t \*off) { if(wait\_event\_interruptible(globalvar.outq,globalvar.flag!=0)) //不可 读时 阻塞读进程 { return -ERESTARTSYS; } if(down\_interruptible(&globalvar.sem)) //P 操作 { return -ERESTARTSYS; } globalvar.flag = 0; printk("into the read function\n"); printk("the rd is %c\n",\*globalvar.rd); //读指针 if(globalvar.rd < globalvar.wr) len = min(len,(size\_t)(globalvar.wr - globalvar.rd)); //更新读写长度 else len = min(len,(size\_t)(globalvar.end - globalvar.rd)); printk("the len is %ld\n",len); if(copy\_to\_user(buf,globalvar.rd,len)) { printk(KERN\_ALERT"copy failed\n"); up(&globalvar.sem); return -EFAULT; } printk("the read buffer is %s\n",globalvar.buffer); globalvar.rd = globalvar.rd + len; if(globalvar.rd == globalvar.end) globalvar.rd = globalvar.buffer; //字符缓冲区循环 up(&globalvar.sem); //V 操作 return len; } static ssize\_t globalvar\_write(struct file \*filp,const char \*buf,size\_t len,loff\_t \*off) { if(down\_interruptible(&globalvar.sem)) //P 操作 { return -ERESTARTSYS; } if(globalvar.rd <= globalvar.wr) len = min(len,(size\_t)(globalvar.end - globalvar.wr)); else len = min(len,(size\_t)(globalvar.rd-globalvar.wr-1)); printk("the write len is %ld\n",len); if(copy\_from\_user(globalvar.wr,buf,len)) { up(&globalvar.sem); //V 操作 return -EFAULT; } printk("the write buffer is %s\n",globalvar.buffer); printk("the len of buffer is %ld\n",strlen(globalvar.buffer)); globalvar.wr = globalvar.wr + len; if(globalvar.wr == globalvar.end) globalvar.wr = globalvar.buffer; //循环 up(&globalvar.sem); //V 操作 globalvar.flag=1; //条件成立，可以唤醒读进程 wake\_up\_interruptible(&globalvar.outq); //唤醒读进程 return len; } module\_init(globalvar\_init); module\_exit(globalvar\_exit); MODULE\_LICENSE("GPL"); ifneq ($(KERNELRELEASE),) obj-m := globalvar.o #obj-m 指编译成外部模块 else KERNELDIR := /lib/modules/$(shell uname -r)/build #定义一个变量,指向内 核目录 PWD := $(shell pwd) modules: $(MAKE) -C $(KERNELDIR) M=$(PWD) modules Endif #include<sys/types.h> #include<unistd.h> #include<sys/stat.h> #include<stdio.h> #include<fcntl.h> #include<string.h> int main() { int fd,i; char msg[101]; fd= open("/dev/chardev0",O\_RDWR,S\_IRUSR|S\_IWUSR); 操作系统专题实验报告 2021 年 1 月 9 日 38 if(fd!=-1) { while(1) { for(i=0;i<101;i++) msg[i]='\0'; read(fd,msg,100); printf("%s\n",msg); if(strcmp(msg,"quit")==0) { close(fd); break; } } } else { printf("device open failure,%d\n",fd); } return 0; } #include<sys/types.h> #include<unistd.h> #include<sys/stat.h> #include<stdio.h> #include<fcntl.h> #include<string.h> int main() { int fd; char msg[10000]; fd= open("/dev/chardev0",O\_RDWR,S\_IRUSR|S\_IWUSR); if(fd!=-1) { while(1) { printf("Please input the globar:\n"); scanf("%s",msg); write(fd,msg,strlen(msg)); if(strcmp(msg,"quit")==0) { close(fd); break; } } } else { printf("device open failure\n"); } return 0; }