

**课 程 实 验 报 告**

**课程名称： 操 作 系 统 原 理**

**专业班级： 计算机科学与技术1609**

**学 号： U201614758**

**姓 名： 梅朝瑞**

**指导教师： 谢 夏**

**报告日期： 2019/1/1**

**计算机科学与技术学院**

# 实验三：共享内存与进程同步

## 实验目的

1. 掌握Linux下共享内存的概念与使用方法；
2. 掌握环形缓冲的结构与使用方法；
3. 掌握Linux下进程同步与通信的主要机制。

## 实验内容

1. **程序要求**

利用多个共享内存（有限空间）构成的环形缓冲，将源文件复制到目标文件，实现两个进程的誊抄。

1. **运行环境**

**软件配置：**

主机：Windows 10 专业版 Build 17763.195

虚拟机：VMware Workstation 14 Pro，Deepin 15.8 Linux/GNU

开发环境：Eclipse IDE for C/C++ Developers Oxygen Release (4.7.0)

**硬件：**

AMD Ryzen 5 1600 Six-Core Processor 3.20 GHz

Kinston HyperX Fury DDR4 2400 8G

NVIDIA GeForce GTX 1060 6GB

1. **源程序**

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <unistd.h>

#include <sys/shm.h>

#include <sys/wait.h>

#include <sys/sem.h>

#include <sys/ipc.h>

#include <sys/types.h>

#include <errno.h>

const size\_t CACHE\_LENGTH = 1020;

//采用信号灯机制实现p、v操作的数据结构和函数

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

struct seminfo \*\_\_buf;

};

int P(int semid, int semnum) {

struct sembuf sops = {semnum, -1, SEM\_UNDO};

return (semop(semid, &sops, 1));

}

int V(int semid, int semnum) {

struct sembuf sops = {semnum, +1, SEM\_UNDO};

return (semop(semid, &sops, 1));

}

int main(int argc, char \*argv[]){

//定义共享缓冲区及其信号灯

int ringbuf\_id = shmget(IPC\_PRIVATE, CACHE\_LENGTH + sizeof (unsigned), IPC\_CREAT | IPC\_EXCL | 0666);

int rbuf\_empty = semget(IPC\_PRIVATE, 1, IPC\_CREAT | IPC\_EXCL | 0666);

int rbuf\_max = semget(IPC\_PRIVATE, 1, IPC\_CREAT | IPC\_EXCL | 0666);

int rbuf\_mutex = semget(IPC\_PRIVATE, 1, IPC\_CREAT | IPC\_EXCL | 0666);

int part\_over = semget(IPC\_PRIVATE, 1, IPC\_CREAT | IPC\_EXCL | 0666);

int over = semget(IPC\_PRIVATE, 1, IPC\_CREAT | IPC\_EXCL | 0666);

int get\_id = 0;

int put\_id = 0;

if (( rbuf\_empty < 0 )||( rbuf\_max < 0 )||( rbuf\_mutex < 0 )||( part\_over < 0) || ( over < 0))

{

printf( "semget error!!! \n");

printf("errno is: %d\n",errno);

exit (1 );

}

union semun semopts;

FILE \*out;

FILE \*in;

//打开源和目标文件

if (argc != 3) {

puts("arguments error");

return 0;

}

if ((in = fopen(argv[1], "rb")) == NULL) {

puts("can't open input file");

return 0;

}

if ((out = fopen(argv[2], "wb")) == NULL) {

puts("can't open output file");

fclose(in);

return 0;

}

//信号灯赋值

semopts.val = CACHE\_LENGTH;

semctl(rbuf\_empty, 0, SETVAL, semopts);

semopts.val = 1;

semctl(rbuf\_mutex, 0, SETVAL, semopts);

semopts.val = 0;

semctl(rbuf\_max, 0, SETVAL, semopts);

semctl(part\_over, 0, SETVAL, semopts);

semctl(over, 0, SETVAL, semopts);

//get

if ((get\_id = fork()) == 0) {

unsigned char\* s = (unsigned char\*)shmat(ringbuf\_id, 0, 0);

char ch;

int i = 0;

puts("get:start");

do {

P(rbuf\_empty, 0);

P(rbuf\_mutex, 0);

//加入内容

if(i >= CACHE\_LENGTH){

i = 0;

shmdt(s);

s = (unsigned char\*)shmat(ringbuf\_id, 0, 0);

}

ch = fgetc(in);

\*s = ch;

printf("get:%c\n", \*s);

V(rbuf\_mutex, 0);

V(rbuf\_max, 0);

if (ch == EOF) break;

else{

s++;

i++;

}

} while (1);

fclose(in);

shmdt(s);

//同步结束进程

P(part\_over,0);

P(over, 0);

puts("get:ended");

return 0;

}

//put

if ((put\_id = fork()) == 0) {

unsigned char \* s = (unsigned char\*)shmat(ringbuf\_id, 0, 0);

int i = 0;

char ch;

puts("put:start");

do {

P(rbuf\_max, 0);

P(rbuf\_mutex, 0);

//加入内容

if(i >= CACHE\_LENGTH){

i = 0;

shmdt(s);

s = (unsigned char\*)shmat(ringbuf\_id, 0, 0);

}

ch = \*s;

printf("put:%c\n", ch);

V(rbuf\_mutex, 0);

V(rbuf\_empty, 0);

if(ch != EOF){

fputc(ch, out);

}

if (ch == EOF) break;

else{

s++;

i++;

}

} while (1);

shmdt(s);

fclose(out);

//同步结束进程

V(over, 0);

V(over, 0);

puts("put:ended");

return 0;

}

wait(0); wait(0);

//释放缓冲区和信号灯

shmctl(ringbuf\_id, IPC\_RMID, 0);

semctl(rbuf\_empty, 0, IPC\_RMID, 0);

semctl(rbuf\_mutex, 0, IPC\_RMID, 0);

semctl(rbuf\_max, 0, IPC\_RMID, 0);

semctl(part\_over, 0, IPC\_RMID, 0);

semctl(over, 0, IPC\_RMID, 0);

return 0;

}

1. **实验结果**

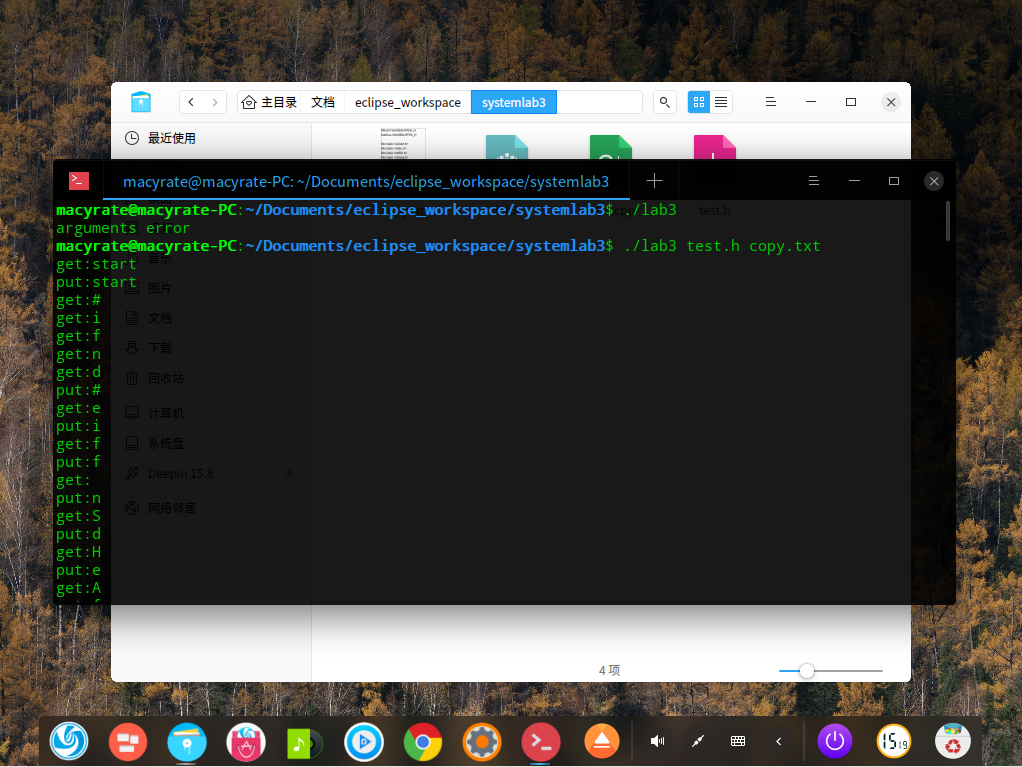


图3-1 程序开始执行

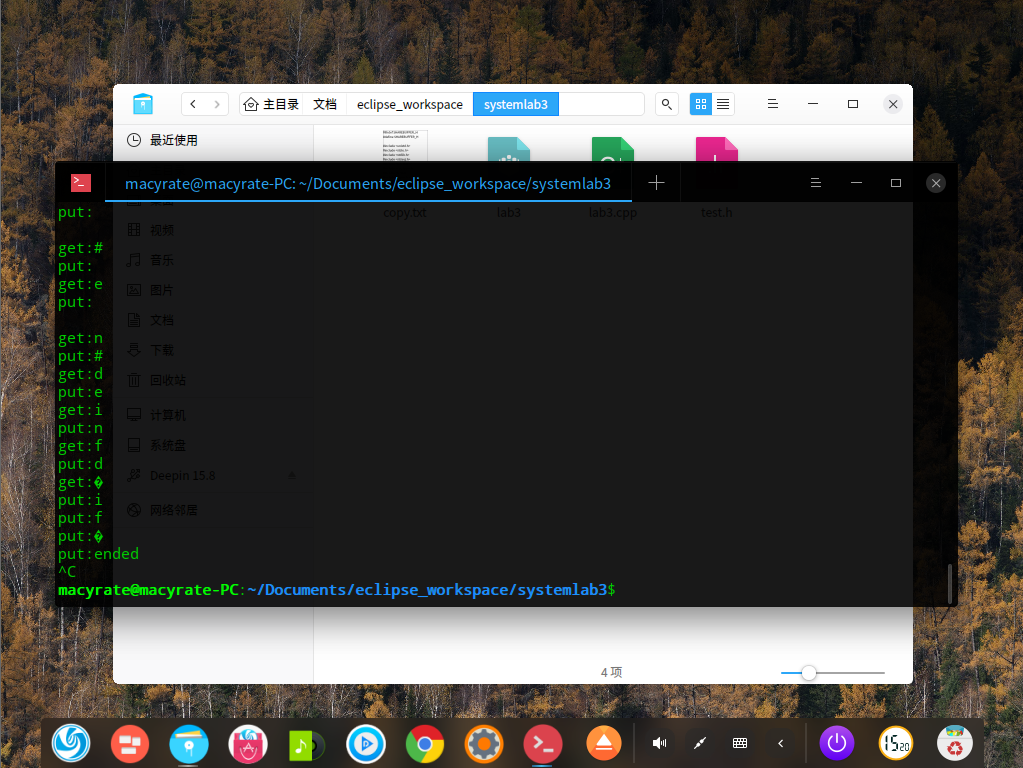


图3-2 程序结束执行

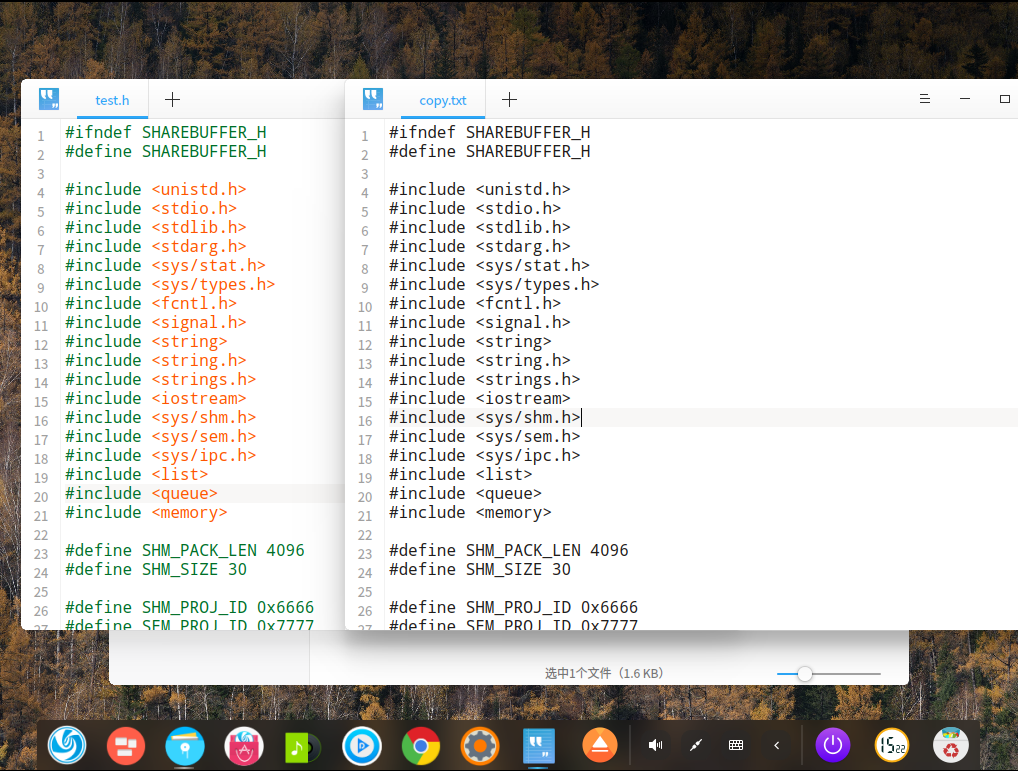


图3-3 复制结果（始）

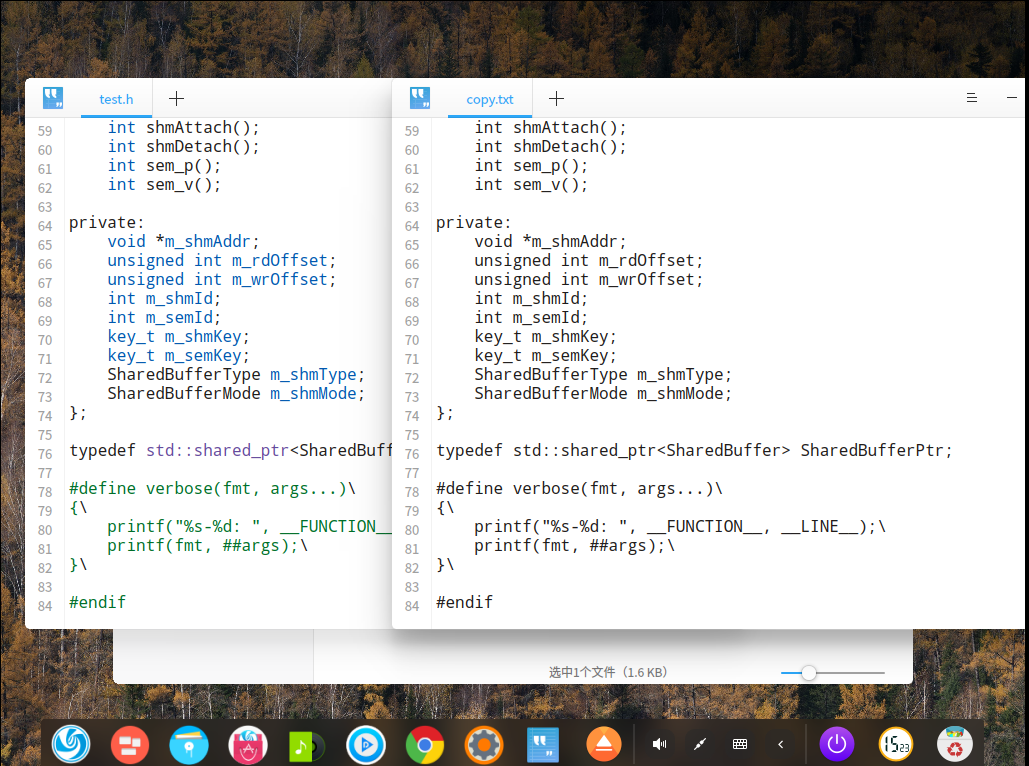


图3-4 复制结果（末）

## 实验心得

这是操作系统的第三次实验。

其实，我也不知道自己实现的到底能不能算作是环形缓冲（笑）。

我的实现方法是开辟一个共享内存区域，将两个指针指向开头和规定的共享区域大小的结尾处，再用一个指针指向当前调用的地址。当占用满共享内存后，将指针指回开头处。

但是，这样会否导致某些字节还未被读取就被覆写？这是我所没有考虑的。不过，大概只要一把梭，把缓冲区设定得够大，足以匹配两边进程之间的速度差异，就可以避免这种问题（所以我足足设置了1020字节）。

按照文档内的说法，我理应开辟数个缓冲区，将他们用指针连接起来成为环状，这是一种做法。

另一种则是使用数组：我想，我这种只开辟一个缓冲区，用三个指针构成环的方法应该可以被视为是char型数组吧。（那么，就没什么问题了）