操作系统原理实验报告

姓　　名：

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：

学　　号：

指导教师：

分数	
教师签名	

目　录

实验三 共享内存与进程同步

1.1 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。

1.2 实验内容

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。

1.3 实验设计

1.3.1 开发环境

虚拟机平台：VMware Workstation Pro 14.1.2 build-8497320
操作系统：ubuntu-16.04.1 (32bit)
内存：1GB
处理器：Intel® Core™ i5-8250U CPU @ 1.60GHz
磁盘：20 GB

1.3.2 实验设计

为实现两个进程的文件誊抄工作，需要使用共享内存资源、信号灯资源和进程资源这 3 类主要资源。

文件的誊抄需要两个进程通过使用共享内存组成的环形缓冲区来解决读写差异导致效率不高的问题。写进程负责将待誊抄文件写入环形缓冲区中，而读进程负责把环形缓冲区内的数据写入复制文件中。两个进程的通信同步过程如图 1.1 所示。

在整体的设计中，主进程获取和初始化所需资源；写进程 writebuf 通过信号灯控制，由指针 now_in 指示写入地址，进行写入环形缓冲区的过程；读进程

readbuf 通过信号灯控制，由指针 now_out 指示读出地址，进行读出环形缓冲区的过程。除了环形缓冲区外，还获取了一块 int 类型大小的共享内存，用于文件传输细节的通信。

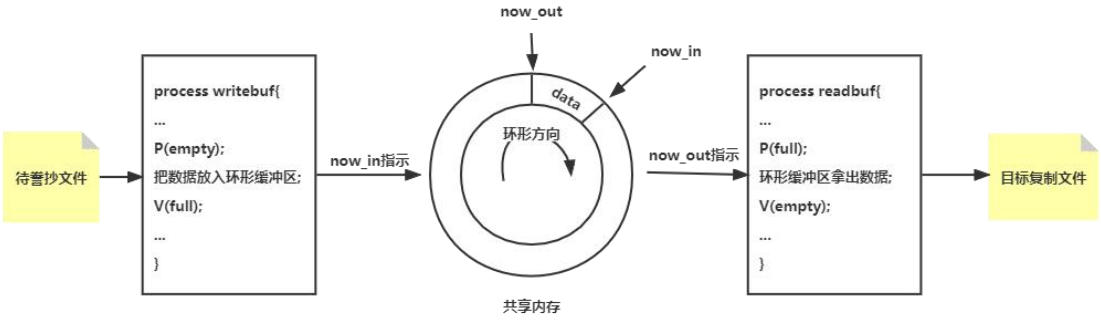

图 1.1 读写进程的同步与通信过程图

具体设计进程设计如下：

主进程：

1. 为使两个进程间能够通信，必须使用同一 Key 值标识符的共享内存与信号灯。故在主进程中自定义 SHMKEY 与 SEMKEY，并且由于文件大小传输信息需要进程间的通信，故也需要定义 SIZEKEY 来标识该共享内存。

2. 使用 shmget 获取一段连续的共享内存，大小为 40960B = 40KB = 4KB*10。即该连续共享内存设置为由 10 个缓冲区构成的环形缓冲，一个缓冲区的大小为 4KB。

3. 使用 shmget 获取一段 int 类型 4B 大小的共享内存，用于存放文件大小信息，进行读写进程间文件大小的通信。

4. 使用 semget 获取包含两个信号灯的信号灯集。其中 empty 指示空闲缓冲区的个数，full 指示满缓冲区的个数。semctl 初始化时，置空闲缓冲区数量为 10，满缓冲区数量为 0。

5. fork 创建两个子进程，写缓冲区进程 writebuf 与读缓冲区进程 readbuf，execv 调用子进程可执行文件。

6. 等待两个子进程结束。删除信号灯集和共存内存组。

写缓冲区进程 writebuf.c：

写缓冲区进程的流程图如图 1.2 所示。具体设计如下：

1. 获取由 SHMKEY 和 SIZEKEY 标识的共享内存，使用 shmat 定位首地址，赋给 char 类型指针 bufaddr 和 int 类型指针 sizeaddr。获取信号灯。

2. 以二进制读形式“rb”打开待誊抄文件“text.txt”。由 fseek 和 ftell 组合算出文件大小 FileLen，rewind 将文件指针重置到文件头。

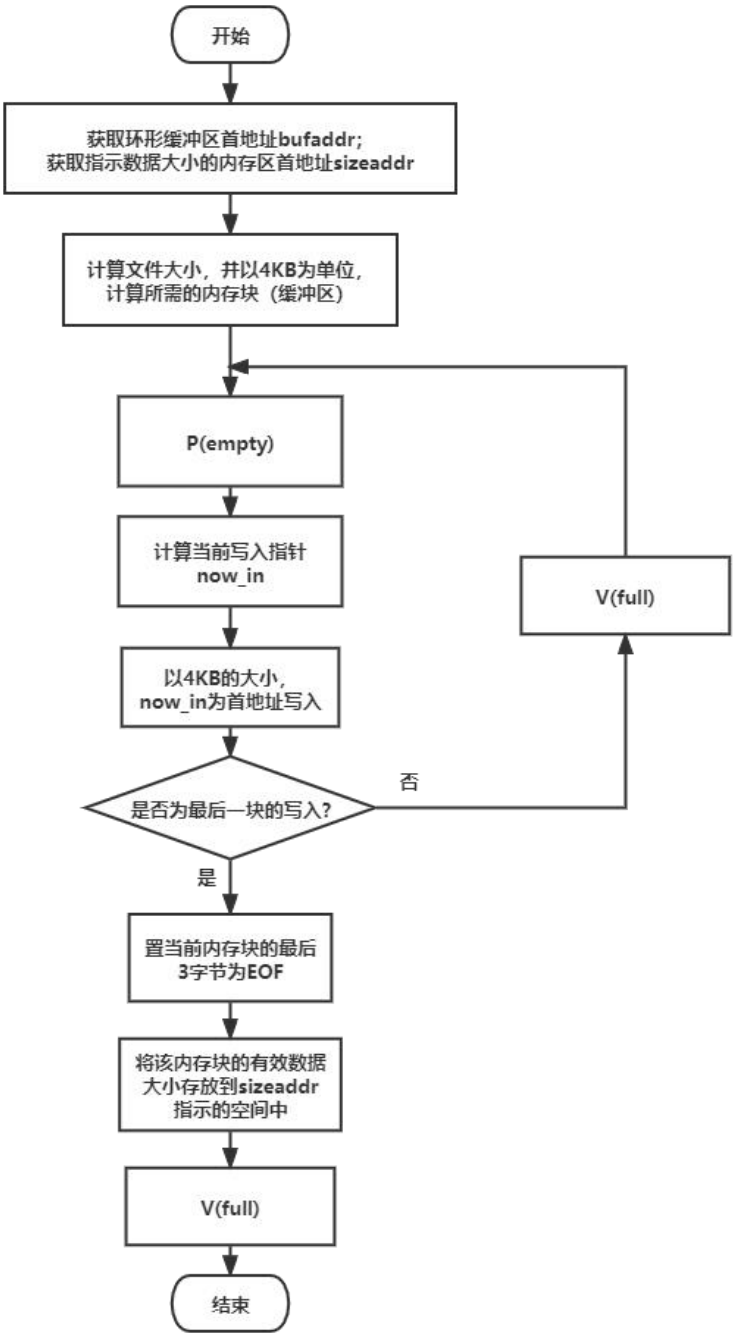

图 1.2 写缓冲区进程 writebuf.c 流程图

3. 计算需要共享内存数，置入 ndnum 中。

4. 进行写入循环，直到将文件的所有内容存入环形缓冲池中。由于读和写是并行的，所以可以实现大文件的放置，当读缓冲区进程读取一个共享内存块时，该块就可以用来放数据了。这也是采用环形缓冲的意义所在，利用有限的缓冲区，实现不限大小的文件誊抄。

5. 循环中，首先进行 P(empty)操作，当有空闲缓冲区时进行一次写入缓冲

区操作。置 now_in 指针为 bufaddr + (i % 10) × 4K。i 指示写入块数，now_in 指示当前写入块的首地址。即当前存放数据内存块的首地址，一个共享内存块大小为 4KB。i % 10 表示环形缓冲区是循环写入的。使用 fread 将 fp 大小为 4KB 的二进制数据，存放到以 now_in 为首地址的共享内存中。在存放时一般以一个块的最大空间来存放，也就是 4KB，即使是最后的剩余数据不足 4KB，也可以 4KB 大小存放到缓冲区。只需要在读出时加以限制即可。一次写入操作结束后，进行 V(full)操作。

6. 若到达最后一次存放时，即循环到 ndnum 时，将该内存块的最后附上 EOF，表明结束标识，让缓冲区读出进程得知该块为最后一个块。并且将最后一个块的有效数据大小放入以 sizeaddr 为首地址的 int 类型大小的共享内存中。进行 V(full)操作。

读缓冲区进程 readbuf.c：

读缓冲区进程的具体设计如下：

1. 同样获取由 SHMKEY 和 SIZEKEY 标识的共享内存，得到指针 bufaddr 和 sizeaddr。获取信号灯。

2. 以写形式“w”打开目标复制文件。

3. 进行读缓冲区循环，进行取指示满的信号灯，即 P(full)。置 now_out 指针为 bufaddr + (j % 10) × 4K。j 指示读取块数，now_out 指示当前读取块的首地址。若存在一个已经存放数据的内存块，判断该块结尾是否为 EOF 标识，来判断是否为最后一个数据块。

4. 若不是最后一个数据块，则使用 fwrite，以 now_out 为首地址，大小为 4K 的内存块写入到目标文件 fp 中。进行 V(empty)操作。

5. 若是最后一个数据块，则通过 sizeaddr 拿到写缓冲区进程计算出的有效数据大小。以 now_out 为首地址，大小为*sizeaddr 的内存卡写入到目标文件 fp 中。进行 V(empty)操作。

1.4 实验调试

1.4.1 实验步骤

1. 生成 3 个可执行文件，exp3、writebuf、readbuf。执行主进程./exp3。

2. 测试小文件誊抄，即文件大小不超过 4KB*10，测试共享内存的缓冲作用。测试文件大小为 4650B，故需要 2 个 4KB 块，其中最后一个块的有效数据

大小为 554B，测试截图如 1.3 所示。方框内为有效数据大小的通信信息。

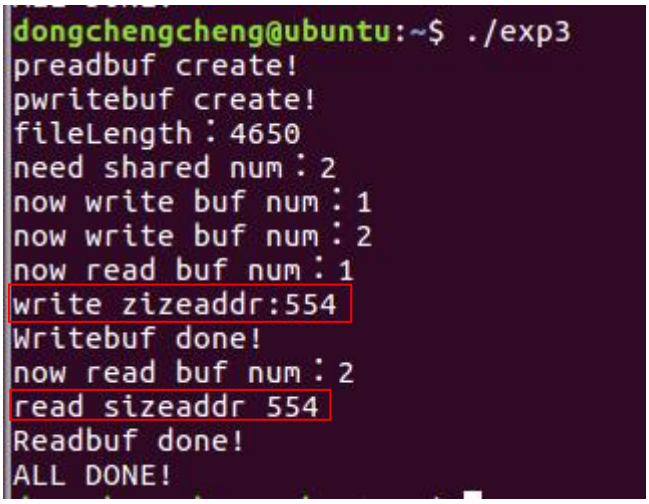

图 1.3 小文件环形缓冲测试

3. 测试大文件誊抄，即文件大小超过 4KB*10，测试环形缓冲区的环形缓冲作用。

测试文件大小 62459B，故需要 16 个 4KB 块，其中最后一个块的有效数据大小为 1019B，测试截图如 1.4 所示。方框内为有效数据大小的通信信息。

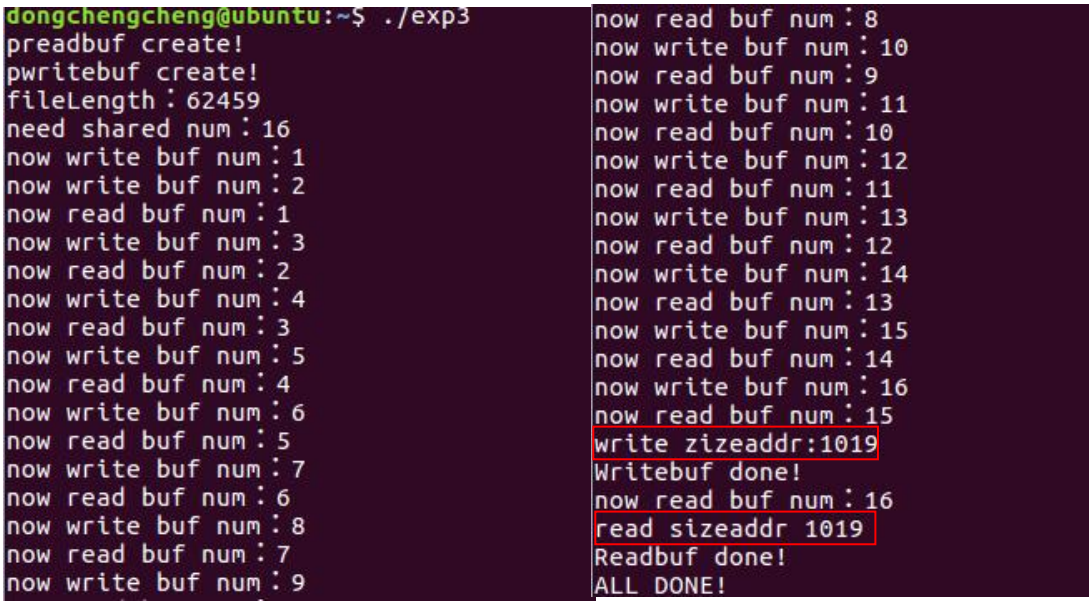

图 1.4 大文件环形缓冲测试

1.4.2 实验调试及心得

调试注意以下几点：

1. 一定要自定义一个标识符 Key 值用作两个子进程间的通信。而不能使用默认 IPC_PRIVATE，因为该标识只能用于父子进程间的通信。但该实验中，需

要通信的是两个子进程，故需要定义一个共同的标识 Key 值，使子进程间拿到的共享内存或信号灯是相同的。

2. 通过 shmat 获取地址时，注意指针类型。由于 fread 和 fwrite 是以字节大小进行 4K 个单元的写入或读取，所以不妨设指针为 char*类型，进行指针计算时，刚好可以以字节大小计算 now_in 与 now_out。而进行有效数据大小的通信时，则用 int*类型指针获取该地址，可以直接使用该数据。

实验心得：

本次实验中，让我掌握了共享内存这一资源的使用，以及通过共享内存进行进程间的同步工作。将一片共享内存作为环形缓冲区，实现读缓冲区进程和写缓冲区进程的并行运行，减小了由读写速率差异带来的影响，大大提高了文件誊抄速率。通过这次实验，也让我真正明白了缓冲区的实际巨大作用。

同时，也进行了进一步优化思考。本次环形缓冲是拿去的一块连续的大空间，其实可以多次获取小空间，将每个小空间的首地址放入索引数组中，就可以很清晰的实现环形缓冲的功能，同时也十分容易理解。并且这样使得空间分散，容易获取，系统实现分散块的获取资源浪费相对较少。并且索引数组相对于链表型的环形缓冲有直接查找的优点，代码实现更为简单。

附录 实验代码

exp3.c:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<unistd.h>
#include<sys/wait.h>
#define SHMKEY 123      //readbuf 与 writebuf 进程之间的共享内存 key 值
#define SIZEKEY 321    //写进程向读进程传递的放置长度公共内存的 key 值
#define SEMKEY 111    //公共信号灯集 key 值
#define SHMNUM 10     //定义共享内存数
int shmid;
int sizeid;
int semid;    //0 号指示 empty，1 号指示 full
pid_t preadbuf,pwritebuf;
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
int main(){
    //创建共享内存组
    if((shmid = shmget(SHMKEY, 40960, IPC_CREAT|0666)) == -1){      //40960 =
4K * 10
        printf("get shared memory fail!\n");
        exit(1);
    }
    //用于读进程获取剩余的待写字节数
    if ((sizeid = shmget(SIZEKEY, 4, IPC_CREAT|0666)) == -1){
        printf("get size shared memory fail！\n");
        exit(1);
    }
```

```
//创建信号灯组
if((semid = semget(SEMKEY, 2, IPC_CREAT|0666)) == -1){
        printf("get semaphore fail!\n");
        exit(1);
}
union semun sem_arg[2];
sem_arg[0].val = SHMNUM;
sem_arg[1].val = 0;
if(semctl(semid, 0, SETVAL, sem_arg[0]) == -1){
        printf("initialize semaphroe 'empty' fail!");
        exit(1);
}
if(semctl(semid, 1, SETVAL, sem_arg[1]) == -1){
        printf("initialize semaphroe 'full' fail!");
        exit(1);
}
//创建进程 readbuf 和 writebuf
if ((pwritebuf = fork()) == 0) {       //写共享内存进程
        printf("pwritebuf create!\n");
        execv("./writebuf",NULL);
}
else {
        if ((preadbuf = fork()) == 0) { //读共享内存进程
                printf("preadbuf create!\n");
                execv("./readbuf",NULL);
        }
        else{     //父进程
                //等待子进程结束
                wait(NULL);
                wait(NULL);
                //删除信号灯和共享内存组
                semctl(semid, 0, IPC_RMID, NULL);
                shmctl(shmid, IPC_RMID, 0);
                shmctl(sizeid, IPC_RMID, 0);
                printf("ALL DONE!\n");
```

```
                exit(0);
            }
        }
}
```

writebuf.c:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<unistd.h>
#define SHMKEY 123      //readbuf 与 writebuf 进程之间的共享内存 key 值
#define SIZEKEY 321    //写进程向读进程传递的放置长度公共内存的 key 值
#define SEMKEY 111    //公共信号灯集 key 值
#define SHMNUM 10      //定义共享内存组数
#define SHMSIZE 4096    //共享内存一组为 4K
int shmid;
int sizeid;
int semid;    //0 号指示 empty，1 号指示 full
char *bufaddr;
int *sizeaddr;
void P(int semid, int index)
{
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = -1;
    sem.sem_flg = 0;
    semop(semid, &sem, 1);
    return;
}
void V(int semid, int index)
{
    struct sembuf sem;
```

```
    sem.sem_num = index;
    sem.sem_op = 1;
    sem.sem_flg = 0;
    semop(semid, &sem, 1);
    return;
}
int main(){
    if((shmid = shmget(SHMKEY, 40960, IPC_CREAT|0666)) == -1){
        printf("writebuf：get shared memory fail!\n");
        exit(1);
    }
    bufaddr = shmat(shmid,NULL,0);
    if ((sizeid = shmget(SIZEKEY, 4, IPC_CREAT|0666)) == -1){
        printf("writebuf：get size shared memory fail！\n");
        exit(1);
    }
    sizeaddr = shmat(sizeid,NULL,0);
    if ((semid = semget(SEMKEY, 2, IPC_CREAT|0666))== -1)
    {
        printf("writebuf：get semaphore fail!\n!");
        exit(1);
    }

    FILE *fp = fopen("text.txt","rb");
    if(fp == NULL) printf("open ‘text.txt’ fail\n");
    fseek(fp,0L,SEEK_END);
    int FileLen = ftell(fp); //计算文件长度
    rewind(fp);
    printf("fileLength：%d\n",FileLen);
    int ndnum;
    if(FileLen % SHMSIZE == 0){     //计算所需总缓冲区数
        ndnum = FileLen/SHMSIZE;
    }
    else ndnum = FileLen/SHMSIZE + 1;
    printf("need shared num：%d\n",ndnum);
```

```
    int i = 0;
    char *now_in;
    while(1){
        P(semid,0);
        now_in = bufaddr + (i % SHMNUM) * SHMSIZE;
        fread(now_in,1,SHMSIZE,fp);
        printf("now write buf num：%d\n",i + 1);
        if(i + 1 == ndnum){
            *(now_in + SHMSIZE - 3) = 'E';
            *(now_in + SHMSIZE - 2) = 'O';
            *(now_in + SHMSIZE - 1) = 'F';
            *(sizeaddr) = FileLen % SHMSIZE;
            V(semid,1);
            break;
        }
        i++;
        V(semid,1);
    }
    printf("Writebuf done!\n");
    exit(0);
}
```

readbuf.c:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<unistd.h>
#define SHMKEY 123     //readbuf 与 writebuf 进程之间的共享内存 key 值
#define SIZEKEY 321    //写进程向读进程传递的放置长度公共内存的 key 值
#define SEMKEY 111    //公共信号灯集 key 值
#define SHMNUM 10     //定义共享内存组数
```

```
#define SHMSIZE 4096    //共享内存一组为 4K
int shmid;
int sizeid;
int semid;    //0 号指示 empty，1 号指示 full
char *bufaddr;
int *sizeaddr;
void P(int semid, int index)
{
        struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op = -1;
        sem.sem_flg = 0;
        semop(semid, &sem, 1);
        return;
}
void V(int semid, int index)
{
        struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op = 1;
        sem.sem_flg = 0;
        semop(semid, &sem, 1);
        return;
}
int main(){
        if((shmid = shmget(SHMKEY, 40960, IPC_CREAT|0666)) == -1){
                printf("readbuf：get shared memory fail!\n");
                exit(1);
        }
        bufaddr = shmat(shmid,NULL,0);
        if ((sizeid = shmget(SIZEKEY, 4, IPC_CREAT|0666)) == -1){
                printf("writebuf：get size shared memory fail！\n");
                exit(1);
        }
        sizeaddr = shmat(sizeid,NULL,0);
```

```
    if ((semid = semget(SEMKEY, 2, IPC_CREAT|0666))== -1)
    {
        printf("readbuf：get semaphore fail!\n!");
        exit(1);
    }

    FILE *fp = fopen("text-copy.txt","w");
    int j = 0;
    char *now_out;
    while(1){
        P(semid,1);
        now_out = bufaddr + (j % SHMNUM) * SHMSIZE;
        if(*(now_out + SHMSIZE - 3) == 'E' && *(now_out + SHMSIZE - 2) ==
'O' && *(now_out + SHMSIZE - 1) == 'F'){
            printf("now read buf num：%d\n",j + 1);
            if(*(sizeaddr) == 0){
                fwrite(now_out,1,SHMSIZE,fp);
            }
            else{
                fwrite(now_out,1,*(sizeaddr),fp);
            }
            V(semid,0);
            break;
        }
        printf("now read buf num：%d\n",j + 1);
        fwrite(now_out,1,SHMSIZE,fp);
        j++;
        V(semid,0);
    }
    printf("Readbuf done!\n");
    exit(0);
}
```