華中科技大學

课 程 实 验 报 告

课程名称： 操 作 系 统 原 理

专业班级： CS1703

学　　号： U201714670

姓　　名： 范唯

指导教师： 谢夏

报告日期： 2019 年 12 月 17 日

计算机科学与技术学院

实验三：共享内存与进程同步

一、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。

二、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。

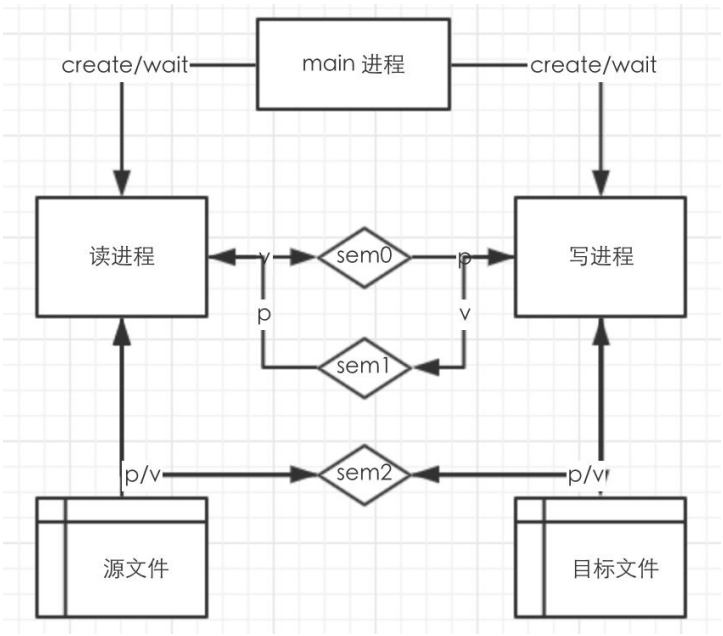
本程序由以下三个进程构成：

・主进程:主进程用于初始化环形缓冲区、启动两个子 进程并等待两个子进程的结束.

・读进程:读进程用于从文件中读取数据并将数据写入环形缓冲区.

・写进程:写进程用于从环形缓冲区中读取数据并写入新文件中.

线程之间的同步通过三个信号量进行，分别为 sem0、1、2.sem0 用于标识读进程读入的数据量,sem1 用于标识环形缓冲中剩余的空间,sem2 用于进程间的互斥，即同时只允许一个进程访问环形缓冲区.如下图



2、运行环境

·本机系统 macOS Catalina.

·VScode 上进行远程连接登陆至华为云服务器——Ubuntu 18.04

·gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)

3、源程序

```
#include<stdio.h>
#include"main.h"

#include<sys/sem.h>//信号灯
#include<sys/shm.h>//共享内存
#include<sys/wait.h>

#include<string.h>
#include<unistd.h>
#include<stdlib.h>
#include<signal.h>


union semun {
        int                     val;       /* Value for SETVAL */
        struct semid_ds *buf;      /* Buffer for IPC_STAT, IPC_SET */
        unsigned short    *array;    /* Array for GETALL, SETALL */
        struct seminfo    *__buf;    /* Buffer for IPC_INFO */
};//信号灯集
```

```
void P(int semid, int index) {
	struct sembuf sem;
	sem.sem_num = index;
	sem.sem_op = -1;
	sem.sem_flg = SEM_UNDO;
	semop(semid, &sem, 1);
	return ;
}//p 操作

void V(int semid, int index) {
	struct sembuf sem;
	sem.sem_num = index;
	sem.sem_op = 1;
	sem.sem_flg = SEM_UNDO;
	semop(semid, &sem, 1);
	return ;
}//v 操作

static int read_proc = 0, write_proc = 0;//读写进程的进程号

typedef struct _ShareBuffer {
	unsigned char status;		//!< current shared buffer status
	uint32_t size;				//!< current message size
	int	nextshm;				//!< pointer to the next Message
	char data[DATASIZE];		//!< where data is stored
} ShareBuffer;

int read_pro(FILE *inFile, int shmhead, int shmtail, int semid) {
	// start to read file to buffer
	ShareBuffer *bufferTail = (ShareBuffer *)shmat(shmtail, NULL, 0);
	while (1) {
		P(semid, 2);//抢占进程权限.上锁
		P(semid, 1);//共享缓冲区空间减 1
		printf("Read\n");//读取中
		size_t bytesRead;
		if ((bytesRead = fread((void *)(bufferTail->data), 1, DATASIZE, inFile)) == 0) {
			//从文件流中读入 DATASIZE 大小的数据进缓冲区//
			bufferTail->status = STATUS_ALL;
			bufferTail->size = bytesRead;
			fclose(inFile);
			V(semid, 0);//读入文件加 1
			V(semid, 2);//释放权限.
			return 0;
		}
```

```
            bufferTail->size = bytesRead;//缓冲区大小为读入数据块的大小
            shmtail = bufferTail->nextshm;//跳转至下一个缓冲区//
            bufferTail = (ShareBuffer *)shmat(shmtail, NULL, 0);//获取该缓冲区
            V(semid, 0);//读入文件加 1
            V(semid, 2);//释放权限
        }
    }

    int write_pro(FILE *outFile, int shmhead, int shmtail, int semid) {
        ShareBuffer *bufferhead = (ShareBuffer *)shmat(shmhead, NULL, 0);
        while (1) {
            P(semid, 2);//抢占进程权限.上锁
            P(semid, 0);//缓冲区文件减 1
            printf("write\n");//写入文件
            if (bufferhead->status == STATUS_ALL) {
                fwrite((void *)(bufferhead->data), bufferhead->size, 1, outFile);//写入缓冲区数据块大小的文件进入目标文件
                fclose(outFile);//关闭文件
                V(semid, 1);//共享缓冲区数量加 1
                V(semid, 2);//解除权限
                return 0;
            }
            fwrite((void *)(bufferhead->data), bufferhead->size, 1, outFile);//写入缓冲区数据块大小的文件进入目标文件
            shmhead = bufferhead->nextshm;//缓冲区头部转移至下一块缓冲区
            bufferhead = (ShareBuffer *)shmat(shmhead, NULL, 0);//获取该缓冲区
            V(semid, 1);//共享缓冲区空间减 1
            V(semid, 2);//解除权限
        }
    }

    int main(int argc, char *argv[]) {
        if (argc != 3) {
            printf("请在 ./main 后依次输入源文件与目标文件\n");
            return 1;
        }//如果不按要求输出文件路径,进行报错

        FILE *fileIn;
        FILE *fileOut;
        if ((fileIn = fopen(argv[1], "rb")) == NULL) {
            perror("无法读取源文件.");
            return 1;
        }//打开源文件
        if ((fileOut = fopen(argv[2], "wb")) == NULL) {
```

```
        perror("无法创建打开目标文件.");
        return 1;
}//如果目标文件不存在,则创建目标文件

key_t shmkey;
if ((shmkey = ftok("./key", 'b')) == (key_t)(-1)) {
        perror("无法获取 key 值");
        exit(1);
}//获取共享内存 key 值

int shm_head;
if ((shm_head = shmget(shmkey, sizeof(ShareBuffer), IPC_CREAT | 0666)) <= 0) {
        perror("无法创建共享缓冲区");
        exit(1);
}//创建第一个 head 共享缓冲区

ShareBuffer *shareBuffer = (ShareBuffer *)shmat(shm_head, NULL, 0);
if ((int64_t)(shareBuffer) == -1) {
        perror("无法获取共享缓冲区");
        exit(1);
}//获取该共享缓冲区
shareBuffer->status = STATUS_PENDING | STATUS_HEAD | STATUS_TAIL;//停等 I/O 输入

for (int i = 0; i < BUFFERNUM; ++i) {
        key_t shmkey;
        if ((shmkey = ftok("./key", i)) == (key_t)(-1)) {
                perror("无法获取 key 值");
                exit(1);
        }//获取共享缓冲区 key 值
        int idShm;
        if ((idShm = shmget(shmkey, sizeof(ShareBuffer), IPC_CREAT | 0666)) <= 0) {
                perror("无法创建共享缓冲区");
                exit(1);
        }//创建共享缓冲区,原理同上述创建 head 缓冲区一样.
        shareBuffer->nextshm = idShm;
        shareBuffer->status = STATUS_PENDING;
        shareBuffer = (ShareBuffer *)shmat(idShm, NULL, 0);
        if ((int64_t)(shareBuffer) == -1) {
                perror("无法获取共享缓冲区");
                exit(1);
        }
}
shareBuffer->nextshm = shm_head;//将最后一个缓冲区链接到头缓冲区构成循环
```

```
    int semid;
    semid = semget(IPC_PRIVATE, 3, IPC_CREAT | 0666);//创建三个信号灯.
    union semun wp; //  用于写进程的参数
    union semun rp; //  用于读进程的参数
    union semun mutex;
    wp.val = BUFFERNUM; //  写进程的初始值为 BUFFSIZE
    rp.val = 0; //  读进程的初始值为 0
    mutex.val = 1;
    if (semctl(semid, 0, SETVAL, wp) == -1 ||
        semctl(semid, 1, SETVAL, rp) == -1 ||
        semctl(semid, 2, SETVAL, mutex)    == -1) {
        perror("IPC error 1: semctl");
        exit(1);
        }//初始化三个信号灯,并将三个信号灯分配给三个进程.

    if ((read_proc = fork()) == -1) {
        perror("Failed to create process.");
        return 1;
    }//创建读文件子进程
    if (read_proc == 0){
        return read_pro(fileIn, shm_head, shm_head, semid);
    }//调用 read_pro 函数读取文件,并放入缓冲区.
    if ((write_proc = fork()) == -1) {
        perror("Failed to create process.");
        kill(read_proc, SIGKILL);//当 read 进程结束时,销毁 write 进程
        return 1;
    }//创建写文件子进程
    if (write_proc == 0){
        return write_pro(fileOut, shm_head, shm_head, semid);
    }//调用 write_pro 函数将文件写入.
    waitpid(read_proc, NULL, 0);//等待进程结束
    waitpid(write_proc, NULL, 0);//等待进程结束
    if (semctl(semid, 0, IPC_RMID) == -1) {
        perror("销毁信号灯失败\n");
    }//销毁信号灯
    if (shmctl(shm_head, IPC_RMID, NULL) == -1) {
        perror("销毁共享内存失败\n");
    }//销毁共享内存
    return 0;
}
```

4、实验结果

・通过 gcc 进行编译.命令为 gcc main.c -o main,得到可执行文件.

```
root@alexfan_Linux:~# cd virtualDesktop/os/os3
root@alexfan_Linux:~/virtualDesktop/os/os3# gcc main.c -o main
root@alexfan_Linux:~/virtualDesktop/os/os3#
```

・然后直接使用./main 进行测试.

```
root@alexfan_Linux:~/virtualDesktop/os/os3# ./main
请在 ./main 后依次输入源文件与目标文件
root@alexfan_Linux:~/virtualDesktop/os/os3#
```

·我们发现会报错,这是因为没有按照指定规则输入源文件和目标文件.

·通过命令 ls 我们发现此时目录中有一个 test.txt,但是没有 copy.txt.

·下面我们运行命令,./main test.txt copy.txt

```
root@alexfan_Linux:~/virtualDesktop/os/os3# ./main text.txt copy.txt
write
write
write
write
write
write
write
```

```
root@alexfan_Linux:~/virtualDesktop/os/os3# ls
copy.txt  key  main  main.c  main.h  text.txt
root@alexfan_Linux:~/virtualDesktop/os/os3#
```

·会发现出现了新文件 copy.txt.通过 md5 对比两个文件.

```
root@alexfan_Linux:~/virtualDesktop/os/os3# md5sum *.txt
9102dd26ed2b617fd3fc74e4ed604f9e  copy.txt
9102dd26ed2b617fd3fc74e4ed604f9e  text.txt
root@alexfan_Linux:~/virtualDesktop/os/os3#
```

使用 md5 对于源文件和目标文件进行校验，发现源文件和目标文件是相同的，说明程序功能正常。

三、实验心得

本次实验是我第一次从操作系统层面编写代码.因为之前并没有接触过信号灯、共享内存的实际操作,所以在这上面投入了大量的时间查看资料与学习实操.

当两个进程通过页表将虚拟地址映射到物理地址时，在物理地址中有一块共同的内存区，即共享内存，这块内存可以被两个进程同时看到。这样当一个进程进行写操作，另一个进程读操作就可以实现进程间通信。但是，我们要确保一个进程在写的时候不能被读，因此我们使用信号量来实现同步与互斥.

而本次实验中我使用了三个信号灯来进行进程间的同步与互斥,信号量 0 用于表示读入的数据量、信号量 1 用于表示缓冲区剩余数量，信号量 2 用于加锁，因此信号 量 0、1、2 的初值分别赋 0，环形缓冲区结点个数、1。最后尝试创建读进程和写进程并等待两个进程结束.在两个进程结束后，释放所有申请的资源。在整个主线程执行的过程中，如果打开文件、创建缓冲区、创建信号量或启动子进程中的任何一步不能成功执行则立即释放已经分配的资源并退出程序.

以上是本次实验的关键点所在,具体编写过程中,对于 shm 和 sem 的操作我主要是通过 man 手册来进行学习.

感谢老师对我的指导与帮助.