華中科技大学

课 程 实 验 报 告

课程名称： 操 作 系 统 原 理

专业班级： CS1704

学　　号： U201714626

姓　　名： 汪清

指导教师： 谢夏、胡侃

报告日期： 2019/12/21

计算机科学与技术学院

目录

实验三：共享内存和进程控制 3
一、实验目的 3
二、实验内容 3
1、程序要求 3
2、运行环境 3
3、源程序 3
4、实验结果 8
三、实验心得 8

实验三：共享内存和进程控制

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；

2、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。

2、运行环境

软件配置（含操作系统版本）：

操作系统：CentOS7.7.1908 64-bits

编辑器：VSCode

编译器：gcc 4.8.5 20150623(Red Hat 4.8.5-39)

硬件：

处理器：Intel(R) Core(TM) i5-7200U CPU @2.50GHz 2.70GHz

硬盘：20GB

内存：1GB

3、源程序

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/fcntl.h>

#define BLOCK_SIZES 5    //缓冲区个数
#define BUFFER_SIZE 1024 //缓冲区大小

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 */
};
union semun arg;
```

```
int semid; //信号量集合首地址
struct share_buffer
{
    int end;
    char buffer[BUFFER_SIZE];
    struct share_buffer *next;
};
struct share_buffer *start = NULL;

char *share[BLOCK_SIZES]; //指向共享缓冲区
int pid1, pid2;           //子进程标识符
int running = 1;
int shmid[BLOCK_SIZES]; //共享缓冲区标识符

/***对信号量数组 index 编号的信号量做 P 操作***/
void P(int semid, int index)
{
    struct sembuf sem = {index, -1, 0};
    semop(semid, &sem, 1);
}

/***对信号量数组 index 编号的信号量做 V 操作***/
void V(int semid, int index)
{
    struct sembuf sem = {index, +1, 0};
    semop(semid, &sem, 1);
}

void writebuf()
{
    int file_end;
    int times = 0;
    const char *pathname;
    struct share_buffer *in = start;
    int fd; //文件描述符
    char pn[100] = "./input.txt";
    pathname = pn;
    if ((fd = open(pathname, O_RDONLY)) == -1)
    {
        printf("打开源文件失败\n");
        return;
    }
    else
        printf("打开源文件成功\n");
```

```
    while (running)
    {
        P(semid, 0);
        file_end = read(fd, in->buffer, BUFFER_SIZE - 1);
        if (file_end != BUFFER_SIZE - 1 && file_end != 0)
        {
            printf("最后一次从源文件读出%d 字节\n", file_end);
            in->end = file_end;
            V(semid, 1);

            close(fd);
            exit(EXIT_SUCCESS);
        }
        printf("第%d 次从源文件读出\n", ++times);
        in = in->next;
        V(semid, 1);
    }
}

void readbuf()
{
    int times = 0;
    const char *pathname;
    struct share_buffer *out = start;
    int fd; //文件描述符
    char pn[100] = "./output.txt";
    pathname = pn;
    if ((fd = open(pathname, O_WRONLY | O_CREAT, S_IRWXU | S_IXGRP | S_IROTH | S_IXOTH)) ==
-1)
    {
        printf("打开目标文件失败");
        return;
    }
    else
        printf("打开目标文件成功\n");
    while (running)
    {
        P(semid, 1);
        if (out->end != 0) //写到文件尾
        {
            printf("最后一次写入%d 个字节到目标文件\n", out->end);
            write(fd, out->buffer, out->end);
            V(semid, 0);
```

```
            close(fd);
            exit(EXIT_SUCCESS);
        }
        write(fd, out->buffer, BUFFER_SIZE - 1);
        printf("第%d 次写入到目标文件\n", ++times);
        out = out->next;
        V(semid, 0);
    }
}
int main()
{
    int ret;
    int i;
    void *shm = NULL;
    key_t key = 1000;
    struct share_buffer *head = NULL, *tail = NULL;
    int size = sizeof(struct share_buffer);
    /***创建共享内存组***/
    for (i = 0; i < BLOCK_SIZES; i++)
    {
        shmid[i] = shmget(key, size, 0666 | IPC_CREAT);
        if (shmid[i] == -1)
        {
            fprintf(stderr, "shmget fail\n");
            exit(EXIT_FAILURE);
        }
        key++;
        //将共享内存连接到当前进程的地址空间
        shm = shmat(shmid[i], (void *)0, 0);
        if (shm == (void *)-1)
            fprintf(stderr, "shmat fail\n");
        //设置共享内存
        head = (struct share_buffer *)shm;
        head->end = 0;
        head->next = tail;
        if (i == 0)
            start = head;
        tail = head;
    }
    start->next = head;
    /***创建两个信号量***/
    semid = semget(key, 2, IPC_CREAT | 0666);
    if (semid == -1)
    {
```

```
    perror("create semget error");
    return;
}
/***对两个信号量赋初值***/
arg.val = BLOCK_SIZES;
ret = semctl(semid, 0, SETVAL, arg); //信号量 semid【0】为 BLOCK_SIZE
arg.val = 0;
ret = semctl(semid, 1, SETVAL, arg); //信号量 semid【1】为 0
if (ret < 0)
{
    perror("ctl sem error");
    semctl(semid, 0, IPC_RMID, arg);
    return -1;
}
/***创建子进程***/
if ((pid1 = fork()) < 0)
{
    perror("Fail to fork");
    exit(EXIT_FAILURE);
}
else if (pid1 == 0)
    writebuf();
else
{
    if ((pid2 = fork()) < 0)
    {
        perror("Fail to fork");
        exit(EXIT_FAILURE);
    }
    else if (pid2 == 0)
        readbuf();
    else
    {
        /***等待子进程结束***/
        waitpid(pid1, NULL, 0);
        waitpid(pid2, NULL, 0);
        printf("子进程已结束\n");
        /***删除信号量***/
        semctl(semid, 0, IPC_RMID, arg);
        semctl(semid, 1, IPC_RMID, arg);
        printf("信号量删除成功\n");
        /***删除缓冲区***/
        for (i = 0; i < BLOCK_SIZES; i++)
            if (shmctl(shmid[i], IPC_RMID, 0) == -1)
```

```
                {
                    fprintf(stderr, "shmctl(IPC_RMID) failed\n");
                    exit(EXIT_FAILURE);
                }
            printf("缓冲区删除成功\n");
            printf("父进程结束\n");
        }
    }
    return 0;
}
```

4、实验结果

对文件编译生成可执行文件，如图 1-1 所示

```
[wangqing@localhost OS_exp]$ gcc -o exp3 exp3.c
[wangqing@localhost OS_exp]$
```

图 1-1 编译代码

运行文件如图 1-2，1-3 所示：

```
[wangqing@localhost OS_exp]$ ./exp3
打开源文件成功
打开目标文件成功
第1次从源文件读出
第2次从源文件读出
第3次从源文件读出
第4次从源文件读出
第5次从源文件读出
第1次写入到目标文件
第2次写入到目标文件
第3次写入到目标文件
第4次写入到目标文件
第5次写入到目标文件
第6次从源文件读出
第7次从源文件读出
第8次从源文件读出
第9次从源文件读出
第10次从源文件读出
第6次写入到目标文件
```

图 1-2 运行可执行文件

```
第19次写入到目标文件
第20次写入到目标文件
第23次从源文件读出
第24次从源文件读出
第25次从源文件读出
第21次写入到目标文件
第22次写入到目标文件
第23次写入到目标文件
第24次写入到目标文件
最后一次从源文件读出683字节
第25次写入到目标文件
最后一次写入683个字节到目标文件
子进程已结束
信号量删除成功
缓冲区删除成功
父进程结束
```

图 1-3 运行可执行文件

比较文件如图 1-4 所示，并无差异

```
[wangqing@localhost OS_exp]$ cmp input.txt output.txt
[wangqing@localhost OS_exp]$
```

图 1-4

三、实验心得

实验一为进程控制。在做该实验之前，对中断一知半解，只知道有这么一个概念和中断可以用来干什么，通过该实验的进行，终于知道了中断如何实现，对中断的概念和作用的理解也就更加深刻。

实验二为线程同步与通信。此次实验比较简单，可以说让我体验了一下信号灯是如何写成可编译执行的程序，在此之前，只能写出伪代码，但我觉得此次实

验有所欠缺的是应该加上 PV 操作底层实现原理，只用已经封装好的库函数并不能学到特别多。

实验三为共享内存和进程控制。此次实验较前两次实验难度有所提升，可以说是在前两个实验的基础上添加了部分，此次实验检查的过程中与老师的讨论让我有所收获，我知道了两个进程使用同一个变量的时候应该加上一个信号灯，否则在时间片到的时候未完成的操作在下一个时间片完成就可能发生条件变化而错误。

实验四为 linux 文件系统。我觉得此次实验只要了解数据结构中的树的遍历就不会太难。

总之，此次操作系统实验并不是很难，不过让我对课程的进一步理解有一定的帮助，期待操作系统的课设。