西 安 邮 电 大 学

（计算机学院）

课内实验报告

实验名称：进程/线程同步

专业名称：软件工程

班　　级：软件 1901

学生姓名：刘璇

学号（8 位）：04193026

指导教师：陈莉君

实验日期：年 月 日

一. 实验目的及实验环境

1.实验目的

通过观察、分析实验现象，深入理解理解信号量的原理及特点，并掌握在 POSIX 规范中的信号量的功能及使用方法。

2.实验环境

（1）硬件

- CPU：Intel(R) Core(TM) i7-9750H CPU @ 2.20GHz 2.21 GHz
- 内存：8.0 GB (7.9 GB 可用)
- 显示器：
- 硬盘空间： 512GB(SSD)

（2）软件

- 虚拟机名称及版本：Vmware Pro
- 操作系统名称及版本：Ubuntu
- 编译器：gcc

二. 实验内容

1、实验前准备工作

阅读参考资料，了解信号量机制的实现原理，并对生产着消费者问题，读者写者问题，哲学家就餐问题进行深刻理解。

2、实验内容

以下内容至少选择一种完成（鼓励全部完成，会有不一样的体会）

1）从对生产者消费者问题，视频给出几种方案，可逐步实现，并对其进行比较，有的同学已经实现了这种代码，就不用做了。

2）读者写者问题，从读者优先到写者优先，给出解决方案，并进行分析

3）哲学家就餐问题，从基本方案到解决死锁的方案，并进行分析

3、提问并回答

提出至少两个问题，并给予回答，或同组内，在讨论区，两个同学为一组，一个提问，一个回答。

三. 方案设计

“读者—写者”问题—记录型信号量一个数据文件或记录可被多个进程共享。只要求读文件的进程称为“Reader 进程”，其它进程则称为“Writer 进程”。允许多个进程同时读一个共享对象，但不允许一个 Writer 进程和其他 Reader 进程或 Writer 进程同时访问共享对象。“读者—写者问题”是保证一个 Writer 进程必须与其他进程互斥地访问共享对象的同步问题。读者—写者问题要解决：读、读共享；写、写互斥；写、读互斥。

①定义互斥信号量 wmutex，实现写、写互斥和写、读互斥。

②定义整型变量 Readcount 表示正在读的进程数目。由于只要有一个 Reader 进程在读，便不允许 Writer 进程写，因此仅当 Readcount=0，即无 Reader 进程在读时，Reader 才需要执行 Wait(wmutex)操作。若 Wait(wmutex)操作成功，Reader 进程便可去读，相应地，做 Readcount+1 操作。同理，仅当 Reader 进程在执行了 Readcount 减 1 操作后其值为 0 时，才需执行 signal(wmutex)操作，以便让 Write 进程写。

③由于 Readcount 为多个读进程共享（修改），因此需要以互斥方式访问，为此，需要定义互斥信号量 rmutex，保证读进程间互斥访问 Readcount。

增加一个限制：最多只允许 RN 个读者同时读。

引入信号量 L，并赋予其初值 RN，通过执行 Swait(L, 1, 1)操作，来控制读者的数目。

每当有一个读者进入时，就要先执行 Swait(L, 1, 1)操作，使 L 的值减 1。当有 RN 个读者进入读后，L 便减为 0，第 RN +1 个读者要进入读时，必然会因 Swait(L, 1, 1)操作失败而阻塞。

四．测试数据、运行结果以及调试过程截图

```c
//semaphores
sem_t RWMutex, mutex1, mutex2, mutex3, wrt;
int writeCount, readCount;


struct data {
        int id;
        int opTime;
        int lastTime;
};

//读者
void* Reader(void* param) {
        int id = ((struct data*)param)->id;
        int lastTime = ((struct data*)param)->lastTime;
        int opTime = ((struct data*)param)->opTime;

        sleep(opTime);
        printf("Thread %d: waiting to read\n", id);

        sem_wait(&mutex3);
        sem_wait(&RWMutex);
        sem_wait(&mutex2);
        readCount++;
        if(readCount == 1)
                sem_wait(&wrt);
        sem_post(&mutex2);
        sem_post(&RWMutex);
        sem_post(&mutex3);

        printf("Thread %d: start reading\n", id);
        /* reading is performed */
        sleep(lastTime);
        printf("Thread %d: end reading\n", id);

        sem_wait(&mutex2);
        readCount--;
        if(readCount == 0)
                sem_post(&wrt);
        sem_post(&mutex2);

        pthread_exit(0);
}
```

```
lx0420@ubuntu:~/czxt$ ./writer
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3Create the 1 thread: Reader
Create the 2 thread: Writer
Create the 3 thread: Reader
Create the 4 thread: Reader
Thread 1: waiting to read
Thread 1: start reading
Thread 2: waiting to write
Thread 3: waiting to read
Thread 4: waiting to read
Thread 1: end reading
Thread 2: start writing
Thread 2: end writing
Thread 3: start reading
Thread 4: start reading
Thread 3: end reading
Thread 4: end reading
```

```c
/*
*    读者优先
*/

# include <stdio.h>
# include <stdlib.h>
# include <time.h>
# include <sys/types.h>
# include <pthread.h>
# include <semaphore.h>
# include <string.h>
# include <unistd.h>

//semaphores
sem_t wrt, mutex;
int readCount;

struct data {
    int id;
    int opTime;
    int lastTime;
};

//读者
void* Reader(void* param) {
    int id = ((struct data*)param)->id;
    int lastTime = ((struct data*)param)->lastTime;
    int opTime = ((struct data*)param)->opTime;

    sleep(opTime);
    printf("Thread %d: waiting to read\n", id);
    sem_wait(&mutex);
    readCount++;
    if(readCount == 1)
        sem_wait(&wrt);
    sem_post(&mutex);

    printf("Thread %d: start reading\n", id);
    /* reading is performed */
    sleep(lastTime);
    printf("Thread %d: end reading\n", id);

    sem_wait(&mutex);
```

```
lx0420@ubuntu:~/czxt$ gcc reader.c -o reader -lpthread
lx0420@ubuntu:~/czxt$ ./reader
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3Create the 1 thread: Reader
Create the 2 thread: Writer
Create the 3 thread: Reader
Create the 4 thread: Reader
Thread 1: waiting to read
Thread 1: start reading
Thread 2: waiting to write
Thread 3: waiting to read
Thread 3: start reading
Thread 4: waiting to read
Thread 4: start reading
Thread 3: end reading
Thread 1: end reading
Thread 4: end reading
Thread 2: start writing
Thread 2: end writing
```

五．总结

1．实验过程中遇到的问题及解决办法；

2．对设计及调试过程的心得体会。

在没有程序占用临界区时，读者与写者之间的竞争都是公平的，所谓的不公平(优先)是在读者优先和写者优先中，优先方只要占有了临界区，那么之后所有优先方的程序(读者或写者)便占有了临界区的主导权，除非没有优先方程序提出要求，否则始终是优先方的程序占有临界区，反观非优先方即使某一次占有了临界区，那么释放过后(此时回到了没有程序占有临界区的情况)，非优先方又要重新和优先方公平竞争，所谓的优先可以理解为优先方在占有临界区后便可以对临界区进行"垄断"。

六．附录：源代码（电子版）

读者优先：

```
/*
*    读者优先
*/
# include <stdio.h>
# include <stdlib.h>
# include <time.h>
# include <sys/types.h>
```

```c
# include <pthread.h>
# include <semaphore.h>
# include <string.h>
# include <unistd.h>

//semaphores
sem_t wrt, mutex;
int readCount;

struct data {
	int id;
	int opTime;
	int lastTime;
};

//读者
void* Reader(void* param) {
	int id = ((struct data*)param)->id;
	int lastTime = ((struct data*)param)->lastTime;
	int opTime = ((struct data*)param)->opTime;

	sleep(opTime);
	printf("Thread %d: waiting to read\n", id);
	sem_wait(&mutex);
	readCount++;
	if(readCount == 1)
		sem_wait(&wrt);
	sem_post(&mutex);

	printf("Thread %d: start reading\n", id);
	/* reading is performed */
	sleep(lastTime);
	printf("Thread %d: end reading\n", id);

	sem_wait(&mutex);
	readCount--;
	if(readCount == 0)
```

```
        sem_post(&wrt);
    sem_post(&mutex);
    pthread_exit(0);
}

//写者
void* Writer(void* param) {
    int id = ((struct data*)param)->id;
    int lastTime = ((struct data*)param)->lastTime;
    int opTime = ((struct data*)param)->opTime;

    sleep(opTime);
    printf("Thread %d: waiting to write\n", id);
    sem_wait(&wrt);

    printf("Thread %d: start writing\n", id);
    /* writing is performed */
    sleep(lastTime);
    printf("Thread %d: end writing\n", id);

    sem_post(&wrt);
    pthread_exit(0);
}

int main() {
    //pthread
    pthread_t tid; // the thread identifier

    pthread_attr_t attr; //set of thread attributes

    /* get the default attributes */
    pthread_attr_init(&attr);

    //initial the semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&wrt, 0, 1);
    readCount = 0;
```

```
    int id = 0;
    while(scanf("%d", &id) != EOF) {

        char role;          //producer or consumer
        int opTime;         //operating time
        int lastTime;       //run time

        scanf("%c%d%d", &role, &opTime, &lastTime);
        struct data* d = (struct data*)malloc(sizeof(struct data));

        d->id = id;
        d->opTime = opTime;
        d->lastTime = lastTime;

        if(role == 'R') {
            printf("Create the %d thread: Reader\n", id);
            pthread_create(&tid, &attr, Reader, d);

        }
        else if(role == 'W') {
            printf("Create the %d thread: Writer\n", id);
            pthread_create(&tid, &attr, Writer, d);
        }
    }

    //信号量销毁
    sem_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```

写者优先：

```
/*
*    写者优先
*/
```

```c
# include <stdio.h>
# include <stdlib.h>
# include <time.h>
# include <sys/types.h>
# include <pthread.h>
# include <semaphore.h>
# include <string.h>
# include <unistd.h>

//semaphores
sem_t RWMutex, mutex1, mutex2, mutex3, wrt;
int writeCount, readCount;


struct data {
	int id;
	int opTime;
	int lastTime;
};

//读者
void* Reader(void* param) {
	int id = ((struct data*)param)->id;
	int lastTime = ((struct data*)param)->lastTime;
	int opTime = ((struct data*)param)->opTime;

	sleep(opTime);
	printf("Thread %d: waiting to read\n", id);

	sem_wait(&mutex3);
	sem_wait(&RWMutex);
	sem_wait(&mutex2);
	readCount++;
	if(readCount == 1)
		sem_wait(&wrt);
	sem_post(&mutex2);
	sem_post(&RWMutex);
```

```
    sem_post(&mutex3);

    printf("Thread %d: start reading\n", id);
    /* reading is performed */
    sleep(lastTime);
    printf("Thread %d: end reading\n", id);

    sem_wait(&mutex2);
    readCount--;
    if(readCount == 0)
        sem_post(&wrt);
    sem_post(&mutex2);

    pthread_exit(0);
}

//写者
void* Writer(void* param) {
    int id = ((struct data*)param)->id;
    int lastTime = ((struct data*)param)->lastTime;
    int opTime = ((struct data*)param)->opTime;

    sleep(opTime);
    printf("Thread %d: waiting to write\n", id);

    sem_wait(&mutex1);
    writeCount++;
    if(writeCount == 1){
        sem_wait(&RWMutex);
    }
    sem_post(&mutex1);

    sem_wait(&wrt);
    printf("Thread %d: start writing\n", id);
    /* writing is performed */
    sleep(lastTime);
    printf("Thread %d: end writing\n", id);
```

```
    sem_post(&wrt);

    sem_wait(&mutex1);
    writeCount--;
    if(writeCount == 0) {
        sem_post(&RWMutex);
    }
    sem_post(&mutex1);

    pthread_exit(0);
}

int main() {
    //pthread
    pthread_t tid; // the thread identifier

    pthread_attr_t attr; //set of thread attributes

    /* get the default attributes */
    pthread_attr_init(&attr);

    //initial the semaphores
     sem_init(&mutex1, 0, 1);
     sem_init(&mutex2, 0, 1);
     sem_init(&mutex3, 0, 1);
     sem_init(&wrt, 0, 1);
     sem_init(&RWMutex, 0, 1);

     readCount = writeCount = 0;

    int id = 0;
    while(scanf("%d", &id) != EOF) {

        char role;       //producer or consumer
        int opTime;      //operating time
        int lastTime;    //run time
```

```c
        scanf("%c%d%d", &role, &opTime, &lastTime);
        struct data* d = (struct data*)malloc(sizeof(struct data));

        d->id = id;
        d->opTime = opTime;
        d->lastTime = lastTime;

        if(role == 'R') {
            printf("Create the %d thread: Reader\n", id);
            pthread_create(&tid, &attr, Reader, d);

        }
        else if(role == 'W') {
            printf("Create the %d thread: Writer\n", id);
            pthread_create(&tid, &attr, Writer, d);
        }
    }

    sem_destroy(&mutex1);
    sem_destroy(&mutex2);
    sem_destroy(&mutex3);
    sem_destroy(&RWMutex);
    sem_destroy(&wrt);

    return 0;
}
```