操 作 系 统

实 验 报 告

实验名称：实验四 同步互斥问题

姓名：王晶

学号：16340217

实验名称：同步互斥问题

一、实验目的：

利用线程同步机制，实现生产者-消费者问题和读者-写者问题。

二、实验要求：

通过文件操作，读入数据，并通过信号量，进行对缓冲操作的保护。完成生产者-消费者问题和读者-写者问题。

三、实验过程：

实验环境是在 macOS 下，基本实现机制和内容和 Linux 一致，但 macOS 下支持创建有名的信号量，因此直接使用 sem_init 会出现错误。此时需要用到 sem_open 函数进行创建，然后通过 sem_unlink 进行删除，否则将一直存在知道内核重启。

生产者-消费者问题的运行结果如下：

```
wangjingdeMBP:OSEx4 wangjing$ ./filetest 6
Producer No.2 produces product No.1
Producer No.5 produces product No.2
Producer No.6 produces product No.3
Comsumer No.1 consumes product No.1
Comsumer No.3 consumes product No.2
Comsumer No.4 consumes product No.3
```

之所以三个生产者先执行，是因为第一个生产者结束时，其他生产者都在等待 mutex 信号量，但消费者还在等待 full 信号量，执行速度上有区别。

其中信号量 empty 用于记录空位，full 用于记录满位，mutex 用于互斥。生产者先等待 empty 信号量，然后等待 mutex 信号量进行操作。而消费者先等待 full 信号量，再等待 mutex 信号量进行操作。

```
void *produce(void *arg)
{
    struct sto *tempsto = (struct sto*)arg;

    while(true){
    sem_wait(empty);
    sleep(tempsto->startTime);
    sem_wait(mutex);
```

```
void *consume(void *arg)
{
    struct sto *tempsto = (struct sto*)arg;

    while(true){
    sem_wait(full);
    sleep(tempsto->startTime);
    sem_wait(mutex);
```

读者-写者问题：

1. 读者优先：

该情况下，多个读者可以同时访问资源，因此不需要设置信号量来限制读者之间对资源的访问，读者和写者之间互斥，而写者之间也是互斥的。

其中 mutex 用于写者之间互斥，writer 用于读者写者之间互斥。

除此之外 read_count 用于记录当前读者的数量，当有读者进入时锁住信号量 writer，直到所有读者线程结束，才释放 writer。

写者只需等待 write 信号量：

```
void *Writer(void *param) {
    struct command* c = (struct command*)param;
    while (true) {
        sleep(c->startTime);
        sem_wait(writer);

        write();
```

而读者要等待 mutex 信号量，并且对 write 信号量进行操作从而阻塞写者：

```
    sleep(c->startTime);
    sem_wait(mutex);

    read_count++;
    if (read_count == 1) {
        sem_wait(writer);
    }
    sem_post(mutex);

    read();
```

结果如下：

```
Create a reader pthread, it's the 1 pthread.
Create a writer pthread, it's the 2 pthread.
Create a reader pthread, it's the 3 pthread.
Create a reader pthread, it's the 4 pthread.
Create a writer pthread, it's the 5 pthread.
Reader pthread 1 requests to read.
Reader pthread 1 begins to read.
Read data 0
Writer pthread 2 requests to write.
Reader pthread 3 requests to read.
Reader pthread 3 begins to read.
Read data 0
Reader pthread 4 requests to read.
Reader pthread 4 begins to read.
Read data 0
Writer pthread 5 requests to write.
Reader pthread 3 stops reading.
Reader pthread 1 stops reading.
Reader pthread 4 stops reading.
Writer pthread 2 begins to write.
Write data 807
Writer pthread 2 stops writing.
Writer pthread 5 begins to write.
Write data 249
Writer pthread 5 stops writing.
```

2. 写者优先：

在该情况下，写者在等待，则新来的读者将不允许进行操作，进入等待，这时用变量 write_count 记录写者的数量，等于 0，即所有写者结束时释放读者的线程。通过信号量 mutexR 在写者线程中阻塞读者线程，信号量 mutexW 在读者线程中阻塞写者线程。ReadAccess 对 read_count 实现互斥，WriteAccess 对 write_count 实现互斥。

通过 writeAccess 对 write_count 进行保护，read_count 同理。并且有写者排队时，通过 mutexR 阻塞读者。

```
cout << "writer pthread
sem_wait(&writeAccess);

write_count++;
if (write_count == 1)
    sem_wait(&mutexR);
sem_post(&writeAccess);
```

写者线程可以畅通无阻的从开始运行到等待操作 mutexR，即阻塞读者线程；而读者线程一开始就要等待 mutexR，这样就是实现了当写者进入等待时，除了正在运行的读者，其他读者都要进入等待，直到写者完成。

```
    sleep(c->startTime);
    cout << "Reader pthread "
    sem_wait(&mutexR);
    sem_wait(&readAccess);

    read_count++;
    if (read_count == 1)
        sem_wait(&mutexW);
    sem_post(&readAccess);
    sem_post(&mutexR);

    cout << "Reader pthread "
    read();
```

运行结果如下：

```
Create a reader pthread-No.1 pthread.
Create a writer pthread-No.2 pthread.
Create a reader pthread-No.3 pthread.
Create a reader pthread-No.4 pthread.
Create a writer pthread-No.5 pthread.
Reader pthread 1 requests to read.
Reader pthread 1 begins to read
Read data 0.
Writer pthread 2 requests to write.
Writer pthread 2 begins to write.
Write data 807.
Reader pthread 3 requests to read.
Reader pthread 3 begins to read
Read data 807.
Reader pthread 4 requests to read.
Reader pthread 4 begins to read
Read data 807.
Reader pthread Writer pthread 3 stops reading.
5 requests to write.
Writer pthread 5 begins to write.
Write data 249.
Reader pthread 1 stops reading.
Writer pthread 2 stops writing.
Writer pthread 5 stops writing.
Reader pthread 4 stops reading.
```