操作系统实验七报告

智能科学与技术　　2024 年 6 月 5 日

实验任务

根据课堂给出的基于信号量 PV 操作的读者-写者（读者优先）同步算法实现一个读写锁，并设计一个方案用于测试 pthread 库中提供的读写锁 pthread_rwlock_t 与前述读写锁，分析其表现是否一致。需基于 gcc 和 pthread 库实现上述内容，并根据运行结果进行论述。提示：

1）明确读者优先、读写公平的概念；

2）测试方案可创建若干读、写线程，利用 sleep 设定读写线程间合理的读、写申请顺序，以及模拟读、写的持续时间。

1　实验环境

1. 电脑操作系统：Windows 11 + wsl（Ubuntu）
2. 编程语言：C 语言
3. 辅助工具：

（1）使用 VScode 用于编辑各种文件。

（2）使用 makefile 组织和管理整个实验项目的编译、部署和运行

2　实验原理

根据实验任务中的要求，针对性进行学习与查询资料，得到任务相关的知识与思路：

2.1　基于信号量 PV 操作的读者-写者（读者优先）同步算法

实验原理的核心是基于信号量 PV 操作的读者-写者问题解决方案，它确保在一个共享资源上读者优先于写者进行操作，从而实现读者和写者之间的同步和互斥访问。以下将详细分析算法的步骤及其实现过程中需要注意的关键点。

算法

```
var
    rc: integer;
    wr, mutex: semaphore;

rc := 0;
wr := 1;
mutex := 1;

procedure read;
begin
```

```
    P(mutex);
    rc := rc + 1;
    if rc = 1 then P(wr);
    V(mutex);
    读文件;
    P(mutex);
    rc := rc - 1;
    if rc = 0 then V(wr);
    V(mutex);
end;

procedure write;
begin
    P(wr);
    写文件;
    V(wr);
end;
```

在该算法中，使用了以下变量和信号量：

rc: 整型变量，用于记录当前正在读取文件的读者数量。

wr: 信号量，用于控制读者和写者对共享资源的访问，初始值为 1，表示资源空闲。

mutex : 信号量，用于保护对 ‘rc‘变量的互斥访问，初始值为 1，确保对 ‘rc‘变量的操作是原子性的，防止竞争条件。

算法步骤分析

读者过程 (procedure read)

1. 进入临界区：调用 P(mutex)，确保对 rc 的操作是互斥的。

2. 更新读者计数：将 rc 加 1，表示又有一个读者准备读取文件。

3. 检查是否第一个读者：如果 rc 等于 1，表示此时是第一个读者，需要调用 P(wr) 来锁定写者信号量，从而阻止写者进行写操作。这确保在有读者正在读取文件时，写者无法写文件。

4. 离开临界区：调用 V(mutex)，允许其他进程访问 rc。

5. 读文件。

6. 进入临界区：调用 P(mutex)，确保对 rc 的操作是互斥的。

7. 更新读者计数：将 rc 减 1，表示有一个读者已经完成读取操作。

8. 检查是否最后一个读者：如果 rc 等于 0，表示此时是最后一个读者，需要调用 V(wr) 来释放写者信号量，从而允许写者进行写操作。

9. 离开临界区：调用 V(mutex)，允许其他进程访问 rc。

写者过程 (procedure write)

1. 进入临界区：调用 P(wr)，获取写者信号量，确保写者可以独占访问共享资源。此时，如果有读者正在读取文件，写者必须等待，直到所有读者都完成读取操作。

2. 写文件。

3. 离开临界区：调用 V(wr)，释放写者信号量，允许其他读者或写者进行操作。

通过对该算法的分析和实验，可以有效地解决在并发环境中读者优先的同步问题，确保多个读者可以并发访问共享资源，而写者在无读者访问时可以独占访问。该实验不仅帮助理解信号量 PV 操作的应用，也有助于掌握并发控制的基本原理。

2.2 pthread 库中提供的读写锁 pthread_rwlock_t

在 POSIX 标准中，pthread 库提供了一套线程同步的机制，包括互斥锁、条件变量以及读写锁。具体到读写锁 pthread_rwlock_t 的实现，POSIX 标准定义了 API，但并没有规定具体的实现细节。这意味着不同的操作系统或库的实现可以有所不同，但基本思路和原理通常是相似的。

关于读写锁 pthread_rwlock_t 的读者和写者哪个优先，是本次实验的研究目标之一，在实验过程与实验结果会进行说明。

2.3 读者优先与读写公平

读者优先和读写公平是两种用于多线程环境中管理读写锁的策略，它们主要在处理读写锁的获取顺序和对线程的公平性上有所不同。读者优先策略意味着当多个线程请求读锁和写锁时，系统会优先满足读锁请求。这种策略的主要优点是可以提高系统的读操作吞吐量，因为读线程可以频繁地获得锁，而不需要等待写线程。然而，这也带来了一个明显的缺点，即写线程可能会被长时间阻塞，导致写线程饥饿。特别是在读线程频繁出现的情况下，写线程可能长时间得不到执行机会。

相比之下，读写公平策略确保读锁和写锁请求按照它们到达的顺序公平地进行处理。这种策略的特点是无论是读线程还是写线程，都不会被长时间饿死，从而保证了系统中的所有操作都能获得合理的执行机会。虽然这种策略可以防止任何一方线程被长时间阻塞，避免饥饿问题，但在某些情况下，可能会导致系统的整体吞吐量低于读者优先策略，因为每次都需要检查公平性。具体来说，读者优先适用于读操作多于写操作的场景，例如缓存系统或数据库查询系统，而读写公平则适用于需要均衡处理读写操作的场景，比如银行系统或库存管理系统。

通过一个例子说明它们的区别：读者优先是说假如我们按照 read write read 的顺序申请，最后是 read read write 的顺序执行，即使 read 比 write 申请得更晚，但只要第一个 read 申请的锁还没结束就还是执行 read，而读写公平就是 read write read。

在实际实现中，可以通过系统提供的扩展属性来配置读写锁的行为。例如，在一些系统中，可以通过非标准的扩展函数设置读写锁的属性，以实现读者优先或写者优先策略。另一方面，如果系统不支持这些扩展属性，可以手动实现自定义的读写锁逻辑，以确保满足应用的需求和性能要求。

3 实验过程

根据实验原理，我们将实验分成两部分，编写 C 语言程序以及分析性能。其中 C 语言程序可由实现读者优先读写锁，创建读写线程进行测试两部分组成。

3.1 实现读者优先读写锁

根据实验原理中的算法，借助 PV 信号量的知识，可以实现一个读者优先读写锁。

1.TestMethod 枚举：定义了测试方法的枚举类型，包括 READ_FIRST、WRITE_FIRST、ALTERNATE 和 RANDOM 四种测试方法，用于测试读写锁的性能和正确性。READ_FIRST：先创建所有读进程，再创建写进程;WRITE_FIRST：先创建所有读进程，再创建写进程;ALTERNATE：交替创建读写进程;RANDOM：随机创建读写进程。

2.LockType 枚举：定义了锁类型的枚举类型，包括 RW_LOCK 和 PTHREAD_RWLOCK 两种类型，用于指定使用的锁类型。

3.rw_lock 结构体：定义了读写锁的结构体，包括了一个整型变量 rc 记录当前进行读操作的线程数量，以及两个信号量 wr_mutex 和 mutex 分别用于读写权限的互斥和读者数量的互斥。

其中 TestMethod 枚举和 LockType 枚举只是作为辅助解析命令行参数。（本实验实验 makefile 进行实验，并且支持两个命令行参数，分别指定四种测试案例方法中的哪一种以及使用哪种读写锁）

```
// 定义测试方法枚举
typedef enum {
    READ_FIRST,
    WRITE_FIRST,
    ALTERNATE,
    RANDOM
} TestMethod;

// 定义锁类型枚举
typedef enum {
    RW_LOCK,
    PTHREAD_RWLOCK
} LockType;

// 定义读写锁结构体
typedef struct {
    int rc; // 读者数量
    sem_t wr_mutex;
    sem_t mutex; // 读者数量互斥信号量
} rw_lock;

// 初始化读写锁
void rw_lock_init(rw_lock *lock) {
    lock->rc = 0;
    sem_init(&lock->wr_mutex, 0, 1);
    sem_init(&lock->mutex, 0, 1);
}

// 读操作
void rw_lock_read_lock(rw_lock *lock) {
    sem_wait(&lock->mutex);
    lock->rc++;
    if (lock->rc == 1) {
        sem_wait(&lock->wr_mutex);
    }
    sem_post(&lock->mutex);
}

// 读操作完成
void rw_lock_read_unlock(rw_lock *lock) {
    sem_wait(&lock->mutex);
    lock->rc--;
```

```
    if (lock->rc == 0) {
        sem_post(&lock->wr_mutex);
    }
    sem_post(&lock->mutex);
}

// 写操作
void rw_lock_write_lock(rw_lock *lock) {
    sem_wait(&lock->wr_mutex);
}

// 写操作完成
void rw_lock_write_unlock(rw_lock *lock) {
    sem_post(&lock->wr_mutex);
}
```

3.2 主函数：测试

主函数文件 main.c 中主要执行的任务为：

1. 全局变量锁声明

2. 自定义读线程和写线程函数

3. 创建读线程和写线程函数

4. 测试方法集成函数

5.main 函数：解析命令行参数，确定测试方法和锁类型，并调用 test 函数进行测试。根据命令行参数设置读写锁的属性，然后执行相应的测试方法。

3.2.1 全局变量锁声明

attr 用于决定 pthread 读写锁的读写优先，这是在实验结果之后发现的属性，会在实验结果中说明。

```
#define NUM_READERS 3
#define NUM_WRITERS 2
#define READ_TIME 1
#define WRITE_TIME 1

// 全局变量声明
rw_lock read_first_lock; // 自定义读写锁
pthread_rwlock_t pthread_lock; // pthread读写锁
pthread_rwlockattr_t attr; //用于决定读写优先
```

3.2.2 读线程执行函数和写线程执行函数

定义了自定义读者优先读写锁 rw_lock 和 pthread 读写锁下的读线程执行函数和写线程执行函数。其中，读线程负责获取读锁并读取数据，写线程负责获取写锁并写入数据。属于非常常规的操作，不

再赘述。

eg.

```
// 自定义读者优先读写锁读线程函数
void *Read_first_reader(void *arg) {
    rw_lock_read_lock(&read_first_lock);
    printf("Read_first␣reader␣%ld␣is␣reading...\n", (long)arg);
    sleep(READ_TIME);
    rw_lock_read_unlock(&read_first_lock);
    printf("Read_first␣reader␣%ld␣finished␣reading.\n", (long)arg);
    return NULL;
}
```

3.2.3 封装创建读线程和写线程的函数

根据指定的读者数量 num_readers 或写者数量 num_writers 创建对应数量的读线程和写线程。根据锁类型的不同，选择调用自定义读写锁或 pthread 读写锁的相应函数。这里是封装了两个创建多个线程的函数，为了使测试那里的代码更加简洁明了。

```
// 创建读线程
void create_reader_threads(pthread_t *threads, long num_readers, LockType lock_type,
     long ld_start) {
    for (long i = ld_start; i < num_readers + ld_start; i++) {
        if (lock_type == RW_LOCK) {
            pthread_create(&threads[i - ld_start], NULL, Read_first_reader, (void *)i
                );
        } else if (lock_type == PTHREAD_RWLOCK) {
            pthread_create(&threads[i - ld_start], NULL, pthread_reader, (void *)i);

        }
    }
}

// 创建写线程
void create_writer_threads(pthread_t *threads, long num_writers, LockType lock_type,
     long ld_start) {
    for (long i = ld_start; i < num_writers + ld_start; i++) {
        if (lock_type == RW_LOCK) {
            pthread_create(&threads[i - ld_start], NULL, Read_first_writer, (void *)i
                );
        } else if (lock_type == PTHREAD_RWLOCK) {
            pthread_create(&threads[i - ld_start], NULL, pthread_writer, (void *)i);
        }
    }
}
```

3.2.4 测试方法集成函数

根据指定的测试方法和锁类型，创建相应的读线程和写线程。测试方法包括 READ_FIRST、WRITE_FIRST、ALTERNATE 和 RANDOM 四种，分别表示先申请所有读者再申请所有写者、先申请所有写者再申请所有读者、交替申请和随机申请。

```
// 测试方法
void test(TestMethod method, LockType lock_type) {
    pthread_t threads[NUM_READERS + NUM_WRITERS];
    long i;
    long Id_readr = 0; //记录printf不同reader的id
    long Id_writer = 0;//记录printf不同writer的id

    if (lock_type == PTHREAD_RWLOCK) {
        pthread_rwlock_init(&pthread_lock, NULL); // 初始化pthread读写锁，默认读者优先
        //pthread_rwlock_init(&pthread_lock, &attr); // 初始化pthread读写锁，写者优先
    } else if (lock_type == RW_LOCK) {
        rw_lock_init(&read_first_lock); // 初始化自定义读者优先读写锁
    }
    switch (method) {
        case READ_FIRST: //先创建所有读进程，再创建写进程
            create_reader_threads(threads, NUM_READERS, lock_type, 0);
            usleep(100);
            create_writer_threads(&threads[NUM_READERS], NUM_WRITERS, lock_type, 0);
            break;
        case WRITE_FIRST: //先创建所有写进程，再创建读进程：write write read read read
            create_writer_threads(threads, NUM_WRITERS, lock_type, 0);
            usleep(100);
            create_reader_threads(&threads[NUM_WRITERS], NUM_READERS, lock_type, 0);
            break;
        case ALTERNATE: //交替创建读写进程，按以下顺序：read，write，read，write...
            for (i = 0; i < NUM_READERS + NUM_WRITERS; i++) {
                if (i % 2 == 0) {
                    create_reader_threads(&threads[i], 1, lock_type, Id_readr);
                    Id_readr += 1;
                } else {
                    create_writer_threads(&threads[i], 1, lock_type, Id_writer);
                    Id_writer += 1;
                }

            }
            break;
        case RANDOM: //随机创建读写进程
            srand(time(NULL));
            for (i = 0; i < NUM_READERS + NUM_WRITERS; i++) {
                if (rand() % 2 == 0) {
                 create_reader_threads(&threads[i], 1, lock_type, Id_readr);
```

```
                    Id_readr += 1;
                } else {
                    create_writer_threads(&threads[i], 1, lock_type, Id_writer);
                    Id_writer += 1;
                }

            }
            break;
        default:
            printf("Invalid test method!\n");
            return;
    }
...
}
```

3.3 makefile 运行示例

```
do:
    gcc -o main rw_lock.c PA7_main.c -pthread
#READ_FIRST: 先创建所有读进程，再创建写进程
    ./main READ_FIRST rw_lock
    ./main READ_FIRST pthread_rwlock_t

#WRITE_FIRST: 先创建所有读进程，再创建写进程
    ./main WRITE_FIRST rw_lock
    ./main WRITE_FIRST pthread_rwlock_t

#ALTERNATE: 交替创建读写进程
    ./main ALTERNATE rw_lock
    ./main ALTERNATE pthread_rwlock_t
#RANDOM : 随机创建读写进程
    ./main RANDOM rw_lock
    ./main RANDOM pthread_rwlock_t
```

4 实验结果

运行 makefile 文件，可以得到两种锁关于四种方法的所有运行结果，由于我们本次实验的研究目标之一是探究 pthread 读写锁与自实现的读者优先读写锁的区别，我们只着重关注 WRITE_FIRST 测试方法（按照 write write read read read 的顺序申请）和 ALTERNATE 测试方法（按照 read write read write read 的顺序申请）。

4.1 ALTERNATE 测试方法

由 Figure1 以及 Figure2 可以看出，pthread 读写锁（Figure2）与自实现的读者优先读写锁（Figure1）均为读者优先，体现在即使是交替申请，但是得到资源的顺序仍旧是 read 在得到权限后，直到最后一个 read 线程归还资源 write 才能得到权限，由此可以看出，默认情况下 pthread 读写锁是读者优先。

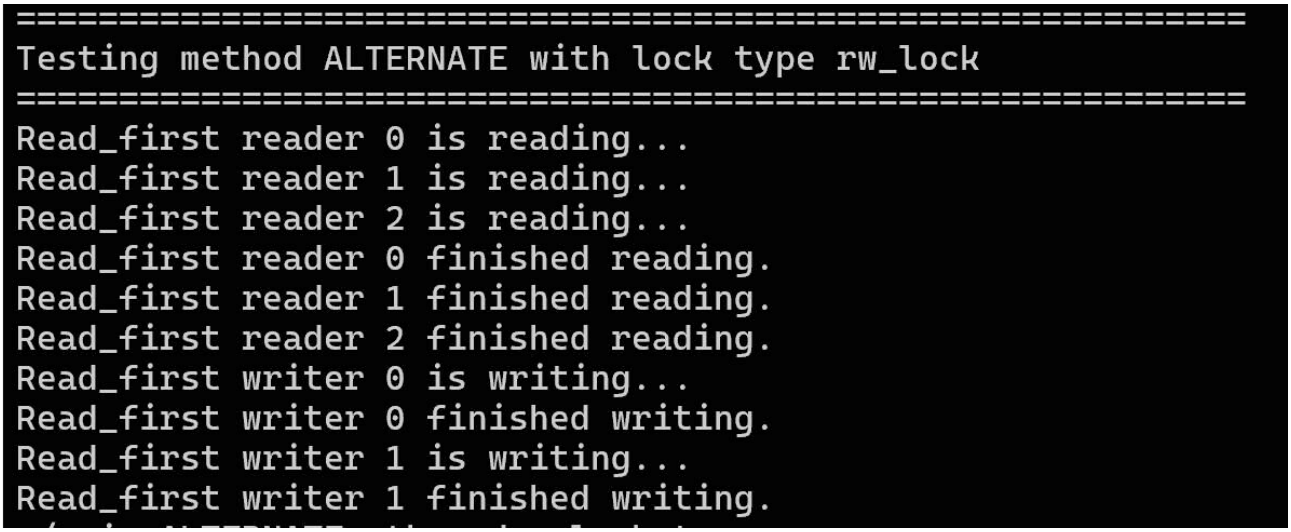

Figure 1: ALTERNATE 测试方法——rw_lock

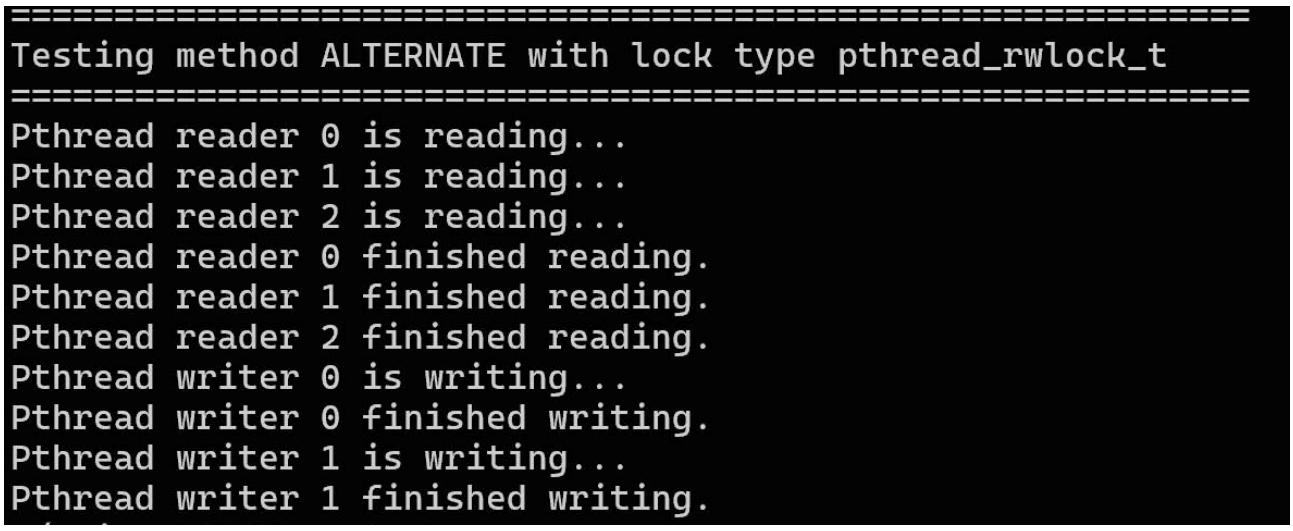

Figure 2: ALTERNATE 测试方法——pthread_rwlock_t

在进行实验前，曾经试图查询 pthread 读写锁的读写优先，在查询过程中，发现了 pthread_rwlockattr_t 的运用，继续查询，发现了 pthread 读写锁可自定义读写优先属性这一特点，通过以上实验可知，默认情况下 pthread 读写锁是读者优先，通过设置属性值 PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP 可以将 pthread 读写锁设置为写者优先，通过对比 pthread 读写锁同一其他配置下 ALTERNATE 测试方法的运行结果（Figure3）可以看出，在设置了属性后，pthread 读写锁变成了写者优先。(具体表现为在 write 得到权限后会执行完所有 write 线程)

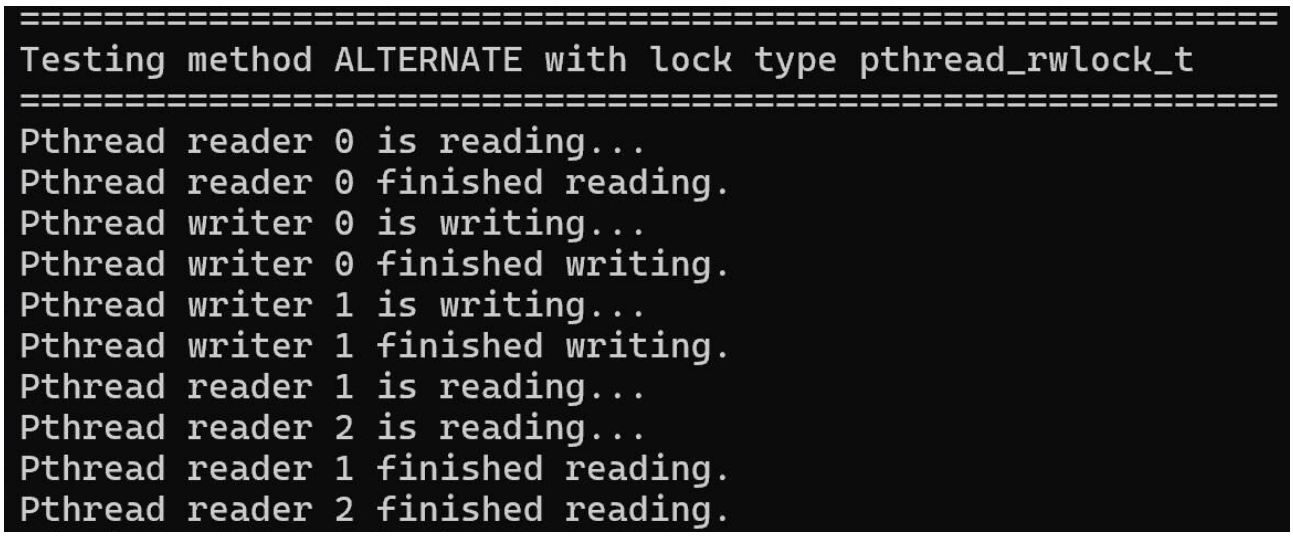

Figure 3: pthread_rwlock_t ALTERNATE 测试方法——写者优先

```
pthread_rwlockattr_t attr; //用于决定读写优先
...

pthread_rwlockattr_init(&attr);
    // 设置读写锁的属性PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP 表示写优先
pthread_rwlockattr_setkind_np(&attr,PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);
...

pthread_rwlock_init(&pthread_lock, &attr); // 初始化pthread读写锁，写者优先
```

4.2 WRITE——FIRST 测试方法

虽然通过以上方法的分析，可以确定 pthread 读写锁是读者优先。但是按照自实现的读者优先算法的思路，WRITE——FIRST 测试方法（按照 write write read read read 的顺序申请）的结果应该是 write write read read read，显然，自实现的读者优先锁 rw_lock 的输出结果确实如此（Figure 4），但是却发现 pthread 读写锁的输出却是 write read read read write（Figure 5），后又构造了多个类似的测试，结果仍旧是如此，推测 Linux 下的 pthread 读写锁为读者锁在等待序列中提升了优先级，即在第一个 write 未结束前，第二个 write 和第一个 read 申请并进入了等待序列，虽然 read 是后进入的，但是因为优先级高，所以在第一个 write 结束后，read 优先申请得到资源，在增加第二个 write 和第一个 read 线程之间的创建时间间隔后，得到了该有的输出，验证了这一点。

```
============================================================
Testing method WRITE_FIRST with lock type rw_lock
============================================================
Read_first writer 0 is writing...
Read_first writer 0 finished writing.
Read_first writer 1 is writing...
Read_first writer 1 finished writing.
Read_first reader 0 is reading...
Read_first reader 1 is reading...
Read_first reader 2 is reading...
Read_first reader 0 finished reading.
Read_first reader 1 finished reading.
Read_first reader 2 finished reading.
```

Figure 4: pthread_rwlock_t WRITE——FIRST 测试方法——rw_lock 锁

```
============================================================
Testing method WRITE_FIRST with lock type pthread_rwlock_t
============================================================
Pthread writer 0 is writing...
Pthread writer 0 finished writing.
Pthread reader 2 is reading...
Pthread reader 0 is reading...
Pthread reader 1 is reading...
Pthread reader 0 finished reading.
Pthread reader 1 finished reading.
Pthread reader 2 finished reading.
Pthread writer 1 is writing...
Pthread writer 1 finished writing.
```

Figure 5: pthread_rwlock_t WRITE——FIRST 测试方法——pthread_rwlock_t 锁

5 实验中的困难与解决

在实验过程中，由于错误使用 sleep（即在每次创建线程后 sleep 一段较长的时间），导致在 ALTERNATE 测试方法中一度出现 read 和 write 也交替执行的情况，导致了错误的判断（以为 pthread 读写锁是读写公平的），后来在修改 sleep 的时间时发现实验结果出现不同，从而发现这个问题。思考原因是在第一个 read 线程执行即将完成前由于 sleep 来不及等到 read 的第二个线程的申请，导致 read 的权限提前结束，出现了读写公平的假象。删除了每次创建进程后的 sleep 后得到正确的实验结果。