下面是一个简单的读优先的读写锁代码示例，使用 POSIX 标准下的互斥锁和条件变量实现：

c复制代码

#include <pthread.h>

typedef struct {

pthread\_mutex\_t lock; *// 互斥锁*

pthread\_cond\_t read\_cv; *// 读条件变量*

pthread\_cond\_t write\_cv; *// 写条件变量*

int active\_readers; *// 正在执行读操作的读者数量*

int waiting\_writers; *// 等待获取写锁的写者数量（包括正在执行写操作的写者）*

int writers; *// 当前写者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

pthread\_mutex\_init(&rwlock->lock, NULL);

pthread\_cond\_init(&rwlock->read\_cv, NULL);

pthread\_cond\_init(&rwlock->write\_cv, NULL);

rwlock->active\_readers = 0;

rwlock->waiting\_writers = 0;

rwlock->writers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->lock);

while (rwlock->waiting\_writers > 0 || rwlock->writers > 0) {

pthread\_cond\_wait(&rwlock->read\_cv, &rwlock->lock);

}

rwlock->active\_readers++;

pthread\_mutex\_unlock(&rwlock->lock);

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->lock);

rwlock->active\_readers--;

if (rwlock->active\_readers == 0 && rwlock->waiting\_writers > 0) {

pthread\_cond\_signal(&rwlock->write\_cv);

}

pthread\_mutex\_unlock(&rwlock->lock);

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->lock);

rwlock->waiting\_writers++;

while (rwlock->active\_readers > 0 || rwlock->writers > 0) {

pthread\_cond\_wait(&rwlock->write\_cv, &rwlock->lock);

}

rwlock->waiting\_writers--;

rwlock->writers++;

pthread\_mutex\_unlock(&rwlock->lock);

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->lock);

rwlock->writers--;

if (rwlock->waiting\_writers > 0) {

pthread\_cond\_signal(&rwlock->write\_cv);

} else {

pthread\_cond\_broadcast(&rwlock->read\_cv);

}

pthread\_mutex\_unlock(&rwlock->lock);

}

在上述代码中，读写锁包含了一个互斥锁和两个条件变量，通过这些同步机制实现对共享资源的互斥访问和读写操作的优化。具体来说：

* 在读操作时，读者首先获取互斥锁并检查是否有等待获取写锁的写者或正在执行写操作的写者，如果有则等待；否则可以直接进行读取。在读操作完成后，读者需要使用条件变量通知其他线程状态的改变。
* 在写操作时，写者首先获取互斥锁并将自己标记为等待获取写锁的写者，并检查是否有正在执行读操作的读者或正在执行写操作的写者，如果有则等待；否则可以直接进行写操作。在写操作完成后，写者需要使用条件变量通知其他线程状态的改变。

总之，这个读写锁实现了读优先，即在多个读者和写者同时存在时，更多的资源会分配给读操作，以保证系统的公平性和效率。

读写锁也可以使用信号量（Semaphore）来实现，下面是一个简单的读写锁代码示例，使用 POSIX 标准下的信号量实现：

c复制代码

#include <pthread.h>

#include <semaphore.h>

typedef struct {

sem\_t rw\_mutex; *// 读写互斥信号量*

sem\_t mutex; *// 访问计数互斥信号量*

int readers; *// 当前读者数量*

int writers; *// 当前写者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

sem\_init(&rwlock->rw\_mutex, 0, 1);

sem\_init(&rwlock->mutex, 0, 1);

rwlock->readers = 0;

rwlock->writers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex);

rwlock->readers++;

if (rwlock->readers == 1) {

sem\_wait(&rwlock->rw\_mutex);

}

sem\_post(&rwlock->mutex);

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex);

rwlock->readers--;

if (rwlock->readers == 0) {

sem\_post(&rwlock->rw\_mutex);

}

sem\_post(&rwlock->mutex);

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex);

rwlock->writers++;

if (rwlock->writers == 1) {

sem\_wait(&rwlock->rw\_mutex);

}

sem\_post(&rwlock->mutex);

sem\_wait(&rwlock->rw\_mutex);

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

sem\_post(&rwlock->rw\_mutex);

sem\_wait(&rwlock->mutex);

rwlock->writers--;

if (rwlock->writers == 0) {

sem\_post(&rwlock->rw\_mutex);

}

sem\_post(&rwlock->mutex);

}

在上述代码中，读写锁包含了两个信号量，一个用来实现读写互斥，另一个用来实现访问计数的互斥。具体来说：

* 在读操作时，读者首先获取访问计数互斥信号量并增加读者数量，然后检查是否是第一个读者，如果是则获取读写互斥信号量，否则直接进行读取。在读操作完成后，读者需要减少读者数量并检查是否是最后一个读者，如果是则释放读写互斥信号量。
* 在写操作时，写者首先获取访问计数互斥信号量并增加写者数量，然后检查是否是第一个写者，如果是则获取读写互斥信号量，否则等待直到能够获取读写互斥信号量。在写操作完成后，写者需要减少写者数量并检查是否是最后一个写者，如果是则释放读写互斥信号量。

总之，读写锁使用信号量实现相对简单，但也需要考虑到互斥和同步的问题，以保证对共享资源的访问正确和高效。

下面是一个使用信号量实现读写公平的读写锁示例：

c复制代码

#include <pthread.h>

#include <semaphore.h>

typedef struct {

sem\_t wrt; *// 写者优先信号量*

sem\_t mutex; *// 访问计数互斥信号量*

int readers; *// 当前读者数量*

int writers; *// 当前写者数量*

int waiting\_readers;*// 等待获取读锁的读者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

sem\_init(&rwlock->wrt, 0, 1);

sem\_init(&rwlock->mutex, 0, 1);

rwlock->readers = 0;

rwlock->writers = 0;

rwlock->waiting\_readers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->wrt);

sem\_wait(&rwlock->mutex);

if (rwlock->waiting\_readers > 0 || rwlock->writers > 0) {

rwlock->waiting\_readers++;

sem\_post(&rwlock->mutex);

sem\_wait(&rwlock->wrt);

sem\_wait(&rwlock->mutex);

rwlock->waiting\_readers--;

}

rwlock->readers++;

sem\_post(&rwlock->mutex);

sem\_post(&rwlock->wrt);

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex);

rwlock->readers--;

if (rwlock->readers == 0 && rwlock->writers > 0) {

sem\_post(&rwlock->wrt);

}

sem\_post(&rwlock->mutex);

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex);

if (rwlock->readers > 0 || rwlock->writers > 0) {

rwlock->writers++;

sem\_post(&rwlock->mutex);

sem\_wait(&rwlock->wrt);

sem\_wait(&rwlock->mutex);

rwlock->writers--;

}

rwlock->writers++;

sem\_post(&rwlock->mutex);

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex);

rwlock->writers--;

if (rwlock->waiting\_readers > 0) {

sem\_post(&rwlock->wrt);

} else {

sem\_post(&rwlock->mutex);

}

}

在上述代码中，读写锁包含了两个信号量，一个用于实现写者优先，另一个用于实现访问计数的互斥。具体来说：

* 在读操作时，读者首先获取写者优先信号量并检查是否有等待获取读锁的读者或正在执行写操作的写者，如果有则加入等待队列并释放写者优先信号量，否则可以直接进行读取。在读操作完成后，读者需要减少读者数量并检查是否是最后一个读者以及是否有等待获取写锁的写者，如果是则唤醒等待队列中的一个写者获取锁。
* 在写操作时，写者首先获取访问计数互斥信号量并检查是否有正在执行读操作的读者或正在执行写操作的写者，如果有则加入等待队列并释放访问计数互斥信号量。然后获取写者优先信号量并检查是否有其他写者正在等待获取锁，如果有则加入等待队列并释放写者优先信号量，否则可以直接进行写操作。在写操作完成后，写者需要减少写者数量并检查是否有等待获取读锁的读者或等待获取写锁的写者，如果是则唤醒等待队列中的一个读者或写者获取锁。

总之，这个读写锁实现了读写公平，即在多个读者和写者同时存在时，会通过优先级策略来分配资源，以保证系统的公平性和效率。

下面是一个使用信号量实现写者优先的读写锁示例：

c复制代码

#include <pthread.h>

#include <semaphore.h>

typedef struct {

sem\_t wrt; *// 写者优先信号量*

sem\_t mutex; *// 访问计数互斥信号量*

int readers; *// 当前读者数量*

int writers; *// 当前写者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

sem\_init(&rwlock->wrt, 0, 1);

sem\_init(&rwlock->mutex, 0, 1);

rwlock->readers = 0;

rwlock->writers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 读操作加锁保护计数器*

if (rwlock->writers > 0) { *// 如果有正在等待或执行的写操作，则等待*

sem\_post(&rwlock->mutex); *// 释放计数器锁，防止出现死锁*

sem\_wait(&rwlock->wrt); *// 等待写操作完成*

sem\_wait(&rwlock->mutex); *// 重新获取计数器锁*

}

rwlock->readers++; *// 增加读者计数器*

sem\_post(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 读操作加锁保护计数器*

rwlock->readers--; *// 减少读者计数器*

if (rwlock->readers == 0) { *// 如果当前没有任何读者，则唤醒等待的写操作*

sem\_post(&rwlock->wrt); *// 唤醒写者优先信号量*

}

sem\_post(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 写操作加锁保护计数器*

rwlock->writers++; *// 增加写者计数器*

if (rwlock->writers == 1) { *// 如果当前是第一个写者，则等待所有读者完成读操作*

sem\_wait(&rwlock->wrt); *// 等待写者优先信号量*

}

sem\_post(&rwlock->mutex); *// 释放计数器锁*

sem\_wait(&rwlock->wrt); *// 等待所有正在进行的读者完成读操作*

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 写操作加锁保护计数器*

rwlock->writers--; *// 减少写者计数器*

if (rwlock->writers == 0) { *// 如果当前没有任何写者，则唤醒等待的读操作或写操作*

sem\_post(&rwlock->wrt); *// 唤醒写者优先信号量*

} else {

sem\_post(&rwlock->wrt); *// 唤醒等待的写操作（如果有的话）*

}

sem\_post(&rwlock->mutex); *// 释放计数器锁*

}

在上述代码中，读写锁包含了两个信号量，一个用于实现写者优先，另一个用于实现访问计数的互斥。具体来说：

* 在读操作时，读者首先获取访问计数互斥信号量，然后检查是否有正在执行写操作的写者，如果是则加入等待队列并等待写者优先信号量唤醒，否则可以直接进行读取。在读操作完成后，读者需要减少读者数量并检查是否是最后一个读者，如果是则唤醒等待队列中的一个写者获取锁。
* 在写操作时，写者首先获取访问计数互斥信号量并将自己标记为正在执行写操作，然后检查是否有其他读者或写者正在执行操作，如果是则加入等待队列并等待写者优先信号量唤醒。在写操作完成后，写者需要减少写者数量并检查是否有等待获取读锁的读者或等待获取写锁的写者，如果是则唤醒等待队列中的一个读者或写者获取锁。
* 总之，这个读写锁实现了写者优先，即在多个读者和写者同时存在时，会优先分配资源给正在等待或执行的写操作，以提高系统的响应速度和效率。

下面是一个使用信号量实现读者优先的读写锁示例：

c复制代码

#include <pthread.h>

#include <semaphore.h>

typedef struct {

sem\_t wrt; *// 写者优先信号量*

sem\_t mutex; *// 访问计数互斥信号量*

int readers; *// 当前读者数量*

int writers; *// 当前写者数量*

int waiting\_writers;*// 等待获取写锁的写者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

sem\_init(&rwlock->wrt, 0, 1);

sem\_init(&rwlock->mutex, 0, 1);

rwlock->readers = 0;

rwlock->writers = 0;

rwlock->waiting\_writers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 读操作加锁保护计数器*

if (rwlock->writers > 0 || rwlock->waiting\_writers > 0) { *// 如果有正在等待或执行的写操作，则等待*

sem\_post(&rwlock->mutex); *// 释放计数器锁，防止出现死锁*

sem\_wait(&rwlock->wrt); *// 等待写操作完成*

sem\_wait(&rwlock->mutex); *// 重新获取计数器锁*

}

rwlock->readers++; *// 增加读者计数器*

sem\_post(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 读操作加锁保护计数器*

rwlock->readers--; *// 减少读者计数器*

if (rwlock->writers > 0 || rwlock->waiting\_writers > 0) { *// 如果有正在等待的写操作，则唤醒所有等待的写者*

sem\_post(&rwlock->wrt); *// 唤醒写者优先信号量*

}

sem\_post(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 写操作加锁保护计数器*

if (rwlock->readers > 0 || rwlock->writers > 0) { *// 如果有正在执行或等待的读操作，则等待*

rwlock->waiting\_writers++; *// 记录等待获取写锁的写者数量*

sem\_post(&rwlock->mutex); *// 释放计数器锁，防止出现死锁*

sem\_wait(&rwlock->wrt); *// 等待写操作完成*

sem\_wait(&rwlock->mutex); *// 重新获取计数器锁*

rwlock->waiting\_writers--; *// 减少等待获取写锁的写者数量*

}

rwlock->writers++; *// 增加写者计数器*

sem\_post(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

sem\_wait(&rwlock->mutex); *// 写操作加锁保护计数器*

rwlock->writers--; *// 减少写者计数器*

if (rwlock->waiting\_writers > 0) { *// 如果有正在等待的写操作，则唤醒一个等待的写者*

sem\_post(&rwlock->wrt); *// 唤醒写者优先信号量*

} else if (rwlock->readers > 0) { *// 如果有正在执行的读操作，则唤醒所有等待的读者*

sem\_post(&rwlock->mutex); *// 先释放计数器锁，防止出现死锁*

sem\_post(&rwlock->wrt); *// 唤醒写者优先信号量（此时写者优先信号量已经被释放）*

sem \_wait(&rwlock->mutex); // 重新获取计数器锁

} else { // 如果没有正在执行的读操作或等待的写操作，则直接释放计数器锁

sem\_post(&rwlock->mutex);

}

}

在上述代码中，读写锁包含了两个信号量，一个用于实现写者优先，另一个用于实现访问计数的互斥。与写者优先不同的是，这里还需要增加一个等待获取写锁的写者数量的变量，以便在唤醒写者时判断是否还有其他写者在等待获取锁。

具体来说：

- 在读操作时，读者首先获取访问计数互斥信号量，然后检查是否有正在执行或等待的写操作，如果是则加入等待队列并等待写者优先信号量唤醒。否则可以直接进行读取。在读操作完成后，读者需要减少读者数量并检查是否有等待获取写锁的写者，如果是则唤醒其中一个写者获取锁。

- 在写操作时，写者首先检查是否有正在执行或等待的读操作或写操作，如果是则加入等待队列并等待写者优先信号量唤醒。在获取锁之前，写者需要先记录自己正在等待获取锁，并在之后的唤醒过程中根据需要减少等待获取锁的写者数量。在写操作完成后，写者需要减少写者数量并检查是否有等待获取锁的写者或正在执行的读操作，如果是则唤醒其中一个写者获取锁，否则唤醒所有等待的读者获取锁。

总之，这个读写锁实现了读者优先，即在多个读者和写者同时存在时，会优先分配资源给正在等待或执行的读操作，以提高系统的并发度和吞吐量。

下面是一个使用mutex实现写者优先的读写锁示例：

c复制代码

#include <pthread.h>

typedef struct {

pthread\_mutex\_t rw\_mutex; *// 读写锁互斥量*

pthread\_mutex\_t mutex; *// 访问计数互斥量*

int readers; *// 当前读者数量*

int writers; *// 当前写者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

pthread\_mutex\_init(&rwlock->rw\_mutex, NULL);

pthread\_mutex\_init(&rwlock->mutex, NULL);

rwlock->readers = 0;

rwlock->writers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 读操作加锁保护计数器*

while (rwlock->writers > 0) { *// 如果有正在等待或执行的写操作，则等待*

pthread\_mutex\_unlock(&rwlock->mutex); *// 先释放计数器锁，防止出现死锁*

pthread\_mutex\_lock(&rwlock->rw\_mutex); *// 等待写操作完成*

pthread\_mutex\_lock(&rwlock->mutex); *// 重新获取计数器锁*

}

rwlock->readers++; *// 增加读者计数器*

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 读操作加锁保护计数器*

rwlock->readers--; *// 减少读者计数器*

if (rwlock->readers == 0) { *// 如果当前没有任何读者，则唤醒等待的写操作*

pthread\_mutex\_unlock(&rwlock->rw\_mutex);*// 唤醒写者优先互斥量*

}

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 写操作加锁保护计数器*

rwlock->writers++; *// 增加写者计数器*

while (rwlock->readers > 0 || rwlock->writers > 1) { *// 如果有正在执行或等待的读操作或写操作，则等待*

pthread\_mutex\_unlock(&rwlock->mutex); *// 先释放计数器锁，防止出现死锁*

pthread\_mutex\_lock(&rwlock->rw\_mutex); *// 等待写操作完成*

pthread\_mutex\_lock(&rwlock->mutex); *// 重新获取计数器锁*

}

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 写操作加锁保护计数器*

rwlock->writers--; *// 减少写者计数器*

if (rwlock->writers == 0) { *// 如果当前没有任何写者，则唤醒等待的读操作或写操作*

pthread\_mutex\_unlock(&rwlock->rw\_mutex);*// 唤醒写者优先互斥量*

} else {

pthread\_mutex\_unlock(&rwlock->rw\_mutex);*// 唤醒等待的写操作（如果有的话）*

}

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

在上述代码中，读写锁包含了两个互斥量，一个用于实现写者优先，另一个用于实现访问计数的互斥。具体来说：

* 在读操作时，读者首先获取访问计数互斥量，然后检查是否有正在执行写操作的写者，如果是则加入等待队列并等待写者优先互斥量唤醒，否则可以直接进行读取。在读操作完成后，读者需要减少读者数量并检查是否

下面是一个使用mutex实现读者优先的读写锁示例：

c复制代码

#include <pthread.h>

typedef struct {

pthread\_mutex\_t rw\_mutex; *// 读写锁互斥量*

pthread\_mutex\_t mutex; *// 访问计数互斥量*

int readers; *// 当前读者数量*

int writers; *// 当前写者数量*

int waiting\_writers; *// 等待获取写锁的写者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

pthread\_mutex\_init(&rwlock->rw\_mutex, NULL);

pthread\_mutex\_init(&rwlock->mutex, NULL);

rwlock->readers = 0;

rwlock->writers = 0;

rwlock->waiting\_writers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 读操作加锁保护计数器*

while (rwlock->writers > 0 || rwlock->waiting\_writers > 0) { *// 如果有正在等待或执行的写操作，则等待*

pthread\_mutex\_unlock(&rwlock->mutex); *// 先释放计数器锁，防止出现死锁*

pthread\_mutex\_lock(&rwlock->rw\_mutex); *// 等待写操作完成*

pthread\_mutex\_lock(&rwlock->mutex); *// 重新获取计数器锁*

}

rwlock->readers++; *// 增加读者计数器*

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 读操作加锁保护计数器*

rwlock->readers--; *// 减少读者计数器*

if (rwlock->readers == 0 && rwlock->waiting\_writers > 0) { *// 如果当前没有任何读者并且有等待获取写锁的写者，则唤醒其中一个写者*

pthread\_mutex\_unlock(&rwlock->rw\_mutex);*// 唤醒写者优先互斥量*

rwlock->writers++; *// 增加写者计数器*

rwlock->waiting\_writers--; *// 减少等待获取写锁的写者数量*

}

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 写操作加锁保护计数器*

while (rwlock->readers > 0 || rwlock->writers > 0) { *// 如果有正在执行或等待的读操作或写操作，则等待*

rwlock->waiting\_writers++; *// 记录等待获取写锁的写者数量*

pthread\_mutex\_unlock(&rwlock->mutex); *// 先释放计数器锁，防止出现死锁*

pthread\_mutex\_lock(&rwlock->rw\_mutex); *// 等待写操作完成*

pthread\_mutex\_lock(&rwlock->mutex); *// 重新获取计数器锁*

rwlock->waiting\_writers--; *// 减少等待获取写锁的写者数量*

}

rwlock->writers++; *// 增加写者计数器*

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 写操作加锁保护计数器*

rwlock->writers--; *// 减少写者计数器*

if (rwlock->waiting\_writers > 0) { *// 如果有正在等待的写操作，则唤醒其中一个写者*

pthread\_mutex\_unlock(&rwlock->rw\_mutex);*// 唤醒写者优先互斥量*

rwlock->writers++; *// 增加写者计数器*

rwlock->waiting\_writers--; *// 减少等待获取写锁的写者数量*

} else if (rwlock->read ers > 0) { // 如果当前没有等待的写操作并且有正在执行的读操作，则唤醒所有等待的读者

pthread\_mutex\_unlock(&rwlock->rw\_mutex);// 唤醒所有读者优先互斥量等待者

}

pthread\_mutex\_unlock(&rwlock->mutex); // 释放计数器锁

}

在上述代码中，读写锁包含了两个互斥量，一个用于实现读者优先，另一个用于实现访问计数的互斥。与写者优先不同的是，这里还需要增加一个等待获取写锁的写者数量的变量，以便在唤醒写者时判断是否还有其他写者在等待。

具体来说：

- 在读操作时，读者首先获取访问计数互斥量，然后检查是否有正在执行或等待的写操作，如果是则加入等待队列并等待读者优先互斥量唤醒。否则可以直接进行读取。在读操作完成后，读者需要减少读者数量并检查是否有等待获取写锁的写者，如果是则唤醒其中一个写者获取锁。

- 在写操作时，写者首先检查是否有正在执行或等待的读操作或写操作，如果是则加入等待队列并等待读者优先互斥量唤醒。在获取锁之前，写者需要记录自己正在等待获取锁，并在之后的唤醒过程中根据需要减少等待获取锁的写者数量。在写操作完成后，写者需要减少写者数量并检查是否有等待获取锁的写者或正在执行的读操作，如果是则唤醒其中一个写者获取锁，否则唤醒所有等待的读者获取锁。

总之，这个读写锁实现了读者优先，即在多个读者和写者同时存在时，会优先分配资源给正在等待或执行的读操作，以提高系统的并发度和吞吐量。

下面是一个使用mutex实现读写公平的读写锁示例：

c复制代码

#include <pthread.h>

typedef struct {

pthread\_mutex\_t rw\_mutex; *// 读写锁互斥量*

pthread\_mutex\_t mutex; *// 访问计数互斥量*

pthread\_cond\_t read\_cond; *// 等待读取操作的条件变量*

pthread\_cond\_t write\_cond; *// 等待写入操作的条件变量*

int readers; *// 当前读者数量*

int writers; *// 当前写者数量*

int waiting\_readers; *// 等待获取读锁的读者数量*

int waiting\_writers; *// 等待获取写锁的写者数量*

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rwlock) {

pthread\_mutex\_init(&rwlock->rw\_mutex, NULL);

pthread\_mutex\_init(&rwlock->mutex, NULL);

pthread\_cond\_init(&rwlock->read\_cond, NULL);

pthread\_cond\_init(&rwlock->write\_cond, NULL);

rwlock->readers = 0;

rwlock->writers = 0;

rwlock->waiting\_readers = 0;

rwlock->waiting\_writers = 0;

}

void rwlock\_read\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 读操作加锁保护计数器*

while (rwlock->writers > 0 || rwlock->waiting\_writers > 0) { *// 如果有正在等待或执行的写操作，则等待*

rwlock->waiting\_readers++; *// 记录等待获取读锁的读者数量*

pthread\_cond\_wait(&rwlock->read\_cond, &rwlock->mutex); *// 等待读取操作条件变量唤醒*

rwlock->waiting\_readers--; *// 减少等待获取读锁的读者数量*

}

rwlock->readers++; *// 增加读者计数器*

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_read\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 读操作加锁保护计数器*

rwlock->readers--; *// 减少读者计数器*

if (rwlock->readers == 0 && rwlock->waiting\_writers > 0) { *// 如果当前没有任何读者并且有等待获取写锁的写者，则唤醒其中一个写者*

pthread\_cond\_signal(&rwlock->write\_cond);*// 唤醒等待获取写锁的写者*

rwlock->writers++; *// 增加写者计数器*

rwlock->waiting\_writers--; *// 减少等待获取写锁的写者数量*

}

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_lock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 写操作加锁保护计数器*

while (rwlock->readers > 0 || rwlock->writers > 0) { *// 如果有正在执行或等待的读操作或写操作，则等待*

rwlock->waiting\_writers++; *// 记录等待获取写锁的写者数量*

pthread\_cond\_wait(&rwlock->write\_cond, &rwlock->mutex);*// 等待写入操作条件变量唤醒*

rwlock->waiting\_writers--; *// 减少等待获取写锁的写者数量*

}

rwlock->writers++; *// 增加写者计数器*

pthread\_mutex\_unlock(&rwlock->mutex); *// 释放计数器锁*

}

void rwlock\_write\_unlock(rwlock\_t \*rwlock) {

pthread\_mutex\_lock(&rwlock->mutex); *// 写操作加锁保护计数器*

rwlock->writers--; *// 减少写者计数器*

if (rwlock->waiting\_readers > 0) { *// 如果有正在等待的读操作，则唤醒所有读者*

pthread\_cond\_broadcast(&rwlock->read\_cond);*//* *唤醒所有等待获取读锁的读者*

*} else if (rwlock->waiting\_writers > 0) { // 如果有正在等待的写操作，则唤醒其中一个写者*

*pthread\_cond\_signal(&rwlock->write\_cond);// 唤醒等待获取写锁的写者*

*rwlock->writers++; // 增加写者计数器*

*rwlock->waiting\_writers--; // 减少等待获取写锁的写者数量*

*}*

*pthread\_mutex\_unlock(&rwlock->mutex); // 释放计数器锁*

*}*

在上述代码中，读写锁同样包含了两个互斥量，但是与读者优先不同的是，这里使用了条件变量来实现读写公平。具体来说：

- 在读操作时，读者首先获取访问计数互斥量，然后检查是否有正在执行或等待的写操作，如果是则加入等待队列并等待写入操作条件变量唤醒。否则可以直接进行读取。在读操作完成后，读者需要减少读者数量并检查是否有等待获取写锁的写者，如果是则唤醒其中一个写者获取锁。

- 在写操作时，写者首先检查是否有正在执行或等待的读操作或写操作，如果是则加入等待队列并等待读取操作条件变量唤醒。在获取锁之前，写者需要记录自己正在等待获取锁，并在之后的唤醒过程中根据需要减少等待获取锁的写者数量。在写操作完成后，写者需要减少写者数量并检查是否有正在等待的读操作或写操作，如果有则唤醒所有等待获取读锁的读者或其中一个等待获取写锁的写者。

总之，这个读写锁实现了读写公平，即在多个读者和写者同时存在时，会尽量避免某个操作一直占用资源而导致其他操作长时间无法获得锁，以提高系统的效率和稳定性。