第二次实例分析 第四部分

第七组组员：赵元浩　廖肇翊　胡羽昊　许卿茹

汇报：胡羽昊

2021.10.27

管道(pipe)	一
睡眠锁(sleeplock)	二
Linux 读写锁	三
Pthread 读写锁	四

PIPE

管道

01

管道 pipe 的数据结构是如何表示的?
其中重要字段的作用分别是什么?

PIPE

数据结构及重要字段作用

*pipe.c

```
#define PIPESIZE 512

struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;
  uint nwrite;
  int readopen;
  int writeopen;
};
```

- lock：自旋锁
- data：管道数据缓冲区
- nread：被读取字节数
- nwrite：被写入字节数
- readopen：记录读文件描述符是否依旧处于open状态
- writeopen：记录写文件描述符是否依旧处于open状态

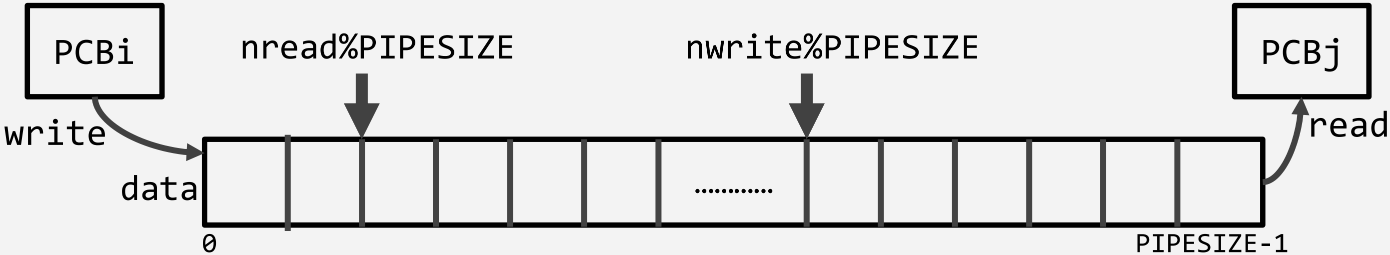
02 & 03

·Pipe 中的 nread 以及 nwrite，在使用过程中，如果超出了缓冲区大小，是否会进行取模回滚的操作?

·Pipe 中的缓冲区判满和判空条件分别是什么?

PIPE

缓冲区工作原理与判断



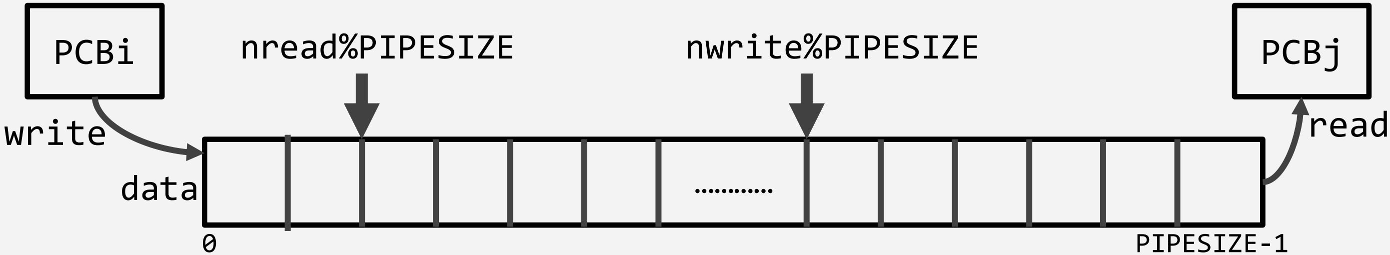
工作原理

管道的读取与写入均需要使用nread和nwrite以获悉管道状态；

在data[PIPESIZE-1]后写入的数据存放在data[0]中；

xv6中使用data[nread%PIPESIZE]和data[nwrite%PIPESIZE]来控制读写时的管道数据偏移，即循环读/写

PIPE
缓冲区工作原理与判断



工作原理

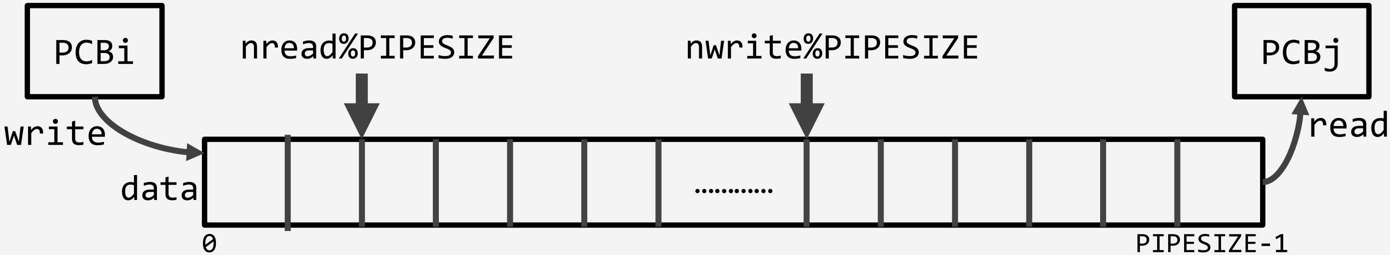
nwrite和nread不会循环，它们一直统计所有写入和读取的字节数

只有缓冲区是循环的，取模回滚

使用被读取字节数nread和被写入字节数nwrite的差值判断管道状态

PIPE

缓冲区工作原理与判断



工作原理

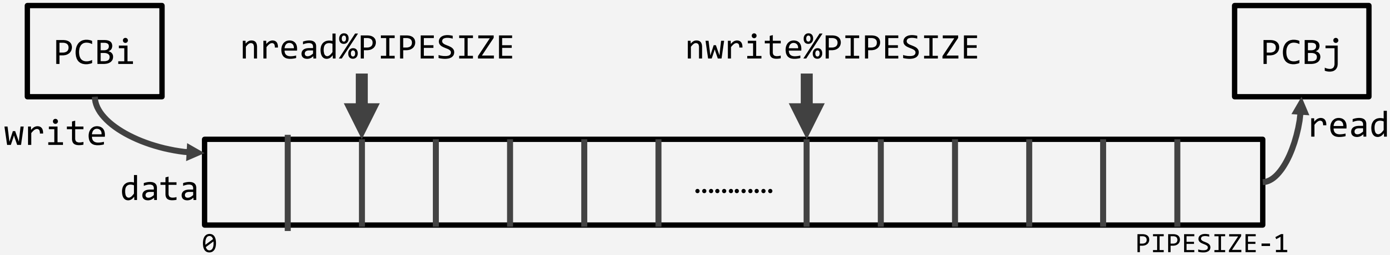
那么，管道空的判断条件是

nwrite == nread

此时数据未写入或被读取完毕

PIPE

缓冲区工作原理与判断



工作原理

那么，管道满的判断条件是

```
nwrite == nread + PIPESIZE
```

04

piperead 和 pipewrite 函数中，为什么要使用两个条件变量（ p->nread , p->nwrite ）？能否只使用一个？如果不能请举例说明。

PIPE

两个条件变量的作用

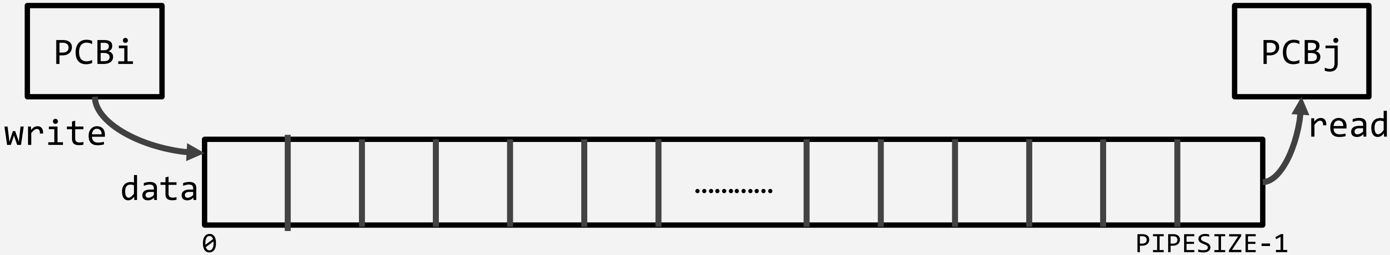
*pipe.c

```
pipe{lock, nread=0, nwite=0, data[512],
     readopen=1, wiriteopen=1};
waddr 被写数据地址  raddr 读取数据地址
n 读/写数据长度
```

```
Pipewirte(pipe, waddr, n){
  Acquire(lock);
  for(i = 0; i < n; i++){
    while(pipe is full){
      if(readopen==0 || PCBi is killed){
        Release(lock);
      }
      Wakeup(nread);
      Sleep(nwrite);
      Release(lock);
    }
    Write waddr[i] to data[nwrite++%512];
  }
  Wakeup(nread);
  Release(lock);
}
```

```
Piperead(pipe, raddr, n){
  Acquire(lock);
  while(pipe is empty && writeopen==1){
      if(PCBj is killed){
        Release(lock);
      }
      Sleep(nread);
      Release(lock);
    }
  for(i = 0; i < n; i++){
    if(pipe is empty) Break;
    Read data[nread++%512] to raddr[i];
  }
  Wakeup(nwrite);
  Release(lock);
}
```

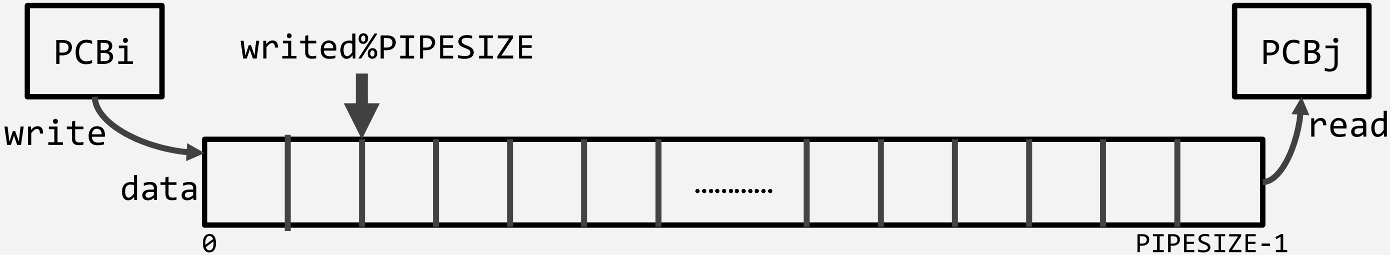
PIPE
两个条件变量的作用



如果只使用一个条件变量......

PIPE

两个条件变量的作用



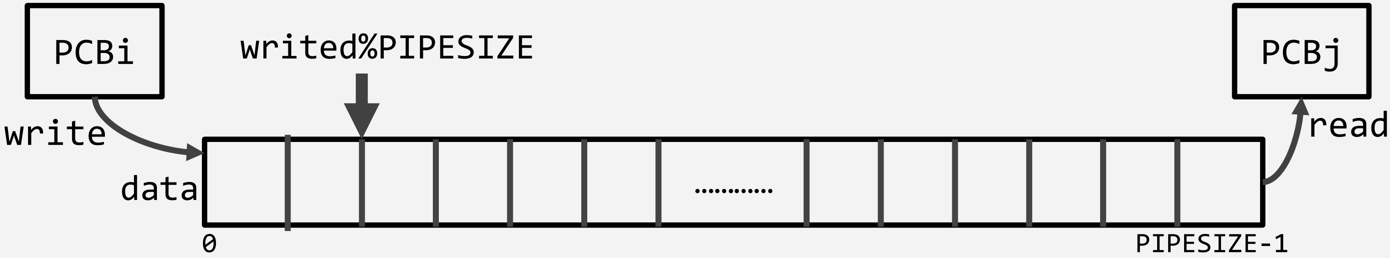
如果只使用一个条件变量......

假设这个条件变量叫writed,表示管道里被写入字节数

（必须有个变量来记录读或写的情况）

PIPE

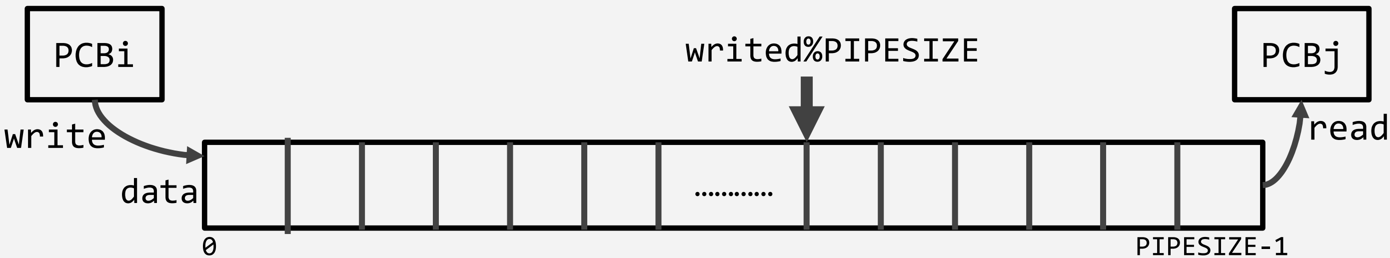
两个条件变量的作用



正常写入没有问题......

PIPE

两个条件变量的作用

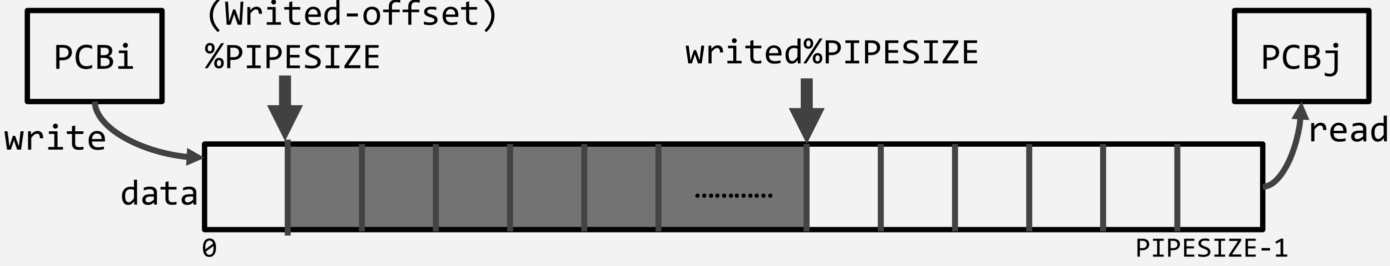


正常写入没有问题......

但当读取时，怎么知道该从哪里读呢......

PIPE

两个条件变量的作用



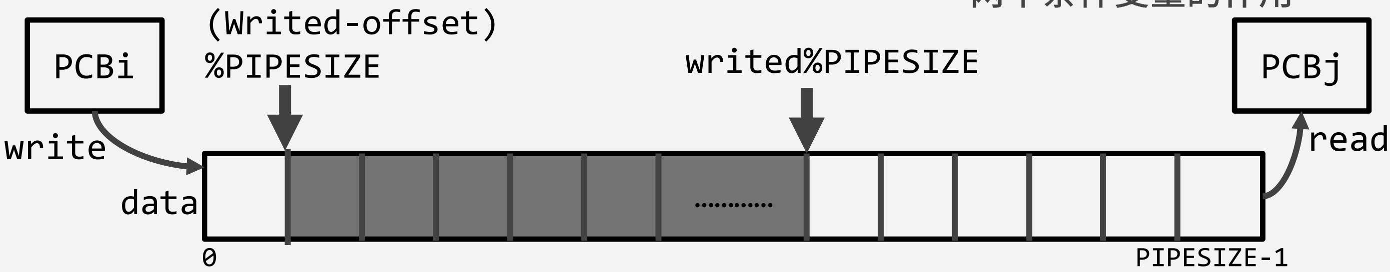
正常写入没有问题......

但当读取时，怎么知道该从哪里读呢......

认为需要添加一个偏移量offset，记录还要多少数据没有被读取

PIPE

两个条件变量的作用



而且在一个进程读取数据的时候不能让offset被其他进程修改

那其实......

本质上和使用nread和nwrite这两个条件变量没什么区别

所以认为不能只使用一个条件变量。

05

当只有一个读者和写者的时候， pipewrite 函数中以及 piperead 函数中的 while 语句能否使用if代替？多读者多写者的情况呢？

PIPE

两个条件变量的作用

*pipe.c

如果只有一个读者和写者的话

```
Pipewirte(pipe, waddr, n){
  Acquire(lock);
  for(i = 0; i < n; i++){
    if  (pipe is full){
      if(readopen==0 || PCBi is killed){
        Release(lock);
      }
      Wakeup(nread);
      Sleep(nwrite);
      Release(lock);
    }
    Write waddr[i] to data[nwrite++%512];
  }
  Wakeup(nread);
  Release(lock);
}
```

```
Piperead(pipe, raddr, n){
  Acquire(lock);
    if (pipe is empty && writeopen==1){
      if(PCBj is killed){
        Release(lock);
      }
      Sleep(nread);
      Release(lock);
    }
  for(i = 0; i < n; i++){
    if(pipe is empty) Break;
    Read data[nread++%512] to raddr[i];
  }
  Wakeup(nwrite);
  Release(lock);
}
```

PIPE

两个条件变量的作用

*pipe.c

用if判断没有问题，在获取lock的时候屏蔽了中断的影响，当写者写满，唤醒读者，若此时读者不读就等待，读者读就立即执行。

```
Pipewirte(pipe, waddr, n){
  Acquire(lock);
  for(i = 0; i < n; i++){
    if (pipe is full){
      if(readopen==0 || PCBi is killed){
        Release(lock);
      }
      Wakeup(nread);
      Sleep(nwrite);
      Release(lock);
    }
    Write waddr[i] to data[nwrite++%512];
  }
  Wakeup(nread);
  Release(lock);
}
```

```
Piperead(pipe, raddr, n){
  Acquire(lock);
    if (pipe is empty && writeopen==1){
      if(PCBj is killed){
        Release(lock);
      }
      Sleep(nread);
      Release(lock);
    }
  for(i = 0; i < n; i++){
    if(pipe is empty) Break;
    Read data[nread++%512] to raddr[i];
  }
  Wakeup(nwrite);
  Release(lock);
}
```

如果是多读者多写者呢

PIPE

两个条件变量的作用

*pipe.c

```
Pipewirte(pipe, waddr, n){
  Acquire(lock);
  for(i = 0; i < n; i++){
    if (pipe is full){
      if(readopen==0 || PCBi is killed){
        Release(lock);
      }
      Wakeup(nread);
      Sleep(nwrite);
      Release(lock);
    }
    Write waddr[i] to data[nwrite++%512];
  }
  Wakeup(nread);
  Release(lock);
}
```

```
Piperead(pipe, raddr, n){
  Acquire(lock);
  if (pipe is empty && writeopen==1){
      if(PCBj is killed){
        Release(lock);
      }
      Sleep(nread);
      Release(lock);
    }
  for(i = 0; i < n; i++){
    if(pipe is empty) Break;
    Read data[nread++%512] to raddr[i];
  }
  Wakeup(nwrite);
  Release(lock);
}
```

PIPE

两个条件变量的作用

*pipe.c

用if判断会出错的，只会对一个写者或一个读者进行判断，不会回来再检查一下其他写者/读者

```
Pipewirte(pipe, waddr, n){
  Acquire(lock);
  for(i = 0; i < n; i++){
    if   (pipe is full){
      if(readopen==0 || PCBi is killed){
        Release(lock);
      }
      Wakeup(nread);
      Sleep(nwrite);
      Release(lock);
    }
    Write waddr[i] to data[nwrite++%512];
  }
  Wakeup(nread);
  Release(lock);
}
```

```
Piperead(pipe, raddr, n){
  Acquire(lock);
    if (pipe is empty && writeopen==1){
      if(PCBj is killed){
        Release(lock);
      }
      Sleep(nread);
      Release(lock);
    }
  for(i = 0; i < n; i++){
    if(pipe is empty) Break;
    Read data[nread++%512] to raddr[i];
  }
  Wakeup(nwrite);
  Release(lock);
}
```

睡 眠 锁

SLEEPLOCK

01

xv6中 sleeplock 的数据结构及其中重要字段的作用

SLEEPLOCK

数据结构及重要字段作用

*sleeplock.h

```
struct sleeplock {
  uint locked;
  struct spinlock lk;

  // For debugging:
  char *name;
  int pid;
};
```

记录睡眠锁是否被锁住

自旋锁

睡眠锁的名字

持有睡眠锁的进程的pid

02

sleeplock 在xv6中的使用场景?

SLEEPLOCK
在xv6中的使用场景

- 有时xv6代码需要长时间持有锁。例如，文件系统在读取和写入磁盘上的文件时需要一直保持文件被锁住，而这些磁盘操作可能需要几十毫秒。
- 为了提高效率，我们希望在等待的过程中处理器可以被用来执行其他的进程，因此需要在上下文切换时还能使用的锁。

03

请简述xv6是如何利用自旋锁(spinlock)和条件变量(sleep & wakeup)实现 sleeplock 的?

```
Acquiresleep(sleeplock){
  Acquire(lock);
  while (sleeplock is locked)
{
    sleep(sleeplock);
    Release (lock);
  }
  Make sleeplock locked;
  pid = currentPCB’s pid;
  Release(lock);
}
```

SLEEPLOCK

实现原理

*sleeplock.c

获得睡眠锁:

- 获得它对应的自旋锁;
- 通过while判断locked域是否为1。如果为1说明睡眠锁被持有，调用sleep函数自动释放自旋锁，并切换进程;
- 如果locked域为0，则将locked域置1表示获得睡眠锁;
- 释放自旋锁。

SLEEPLOCK

实现原理

*sleeplock.c

```
Releasesleep(sleeplock){
  Acquire(lock);
  Make sleeplock unlocked;
  pid = 0;
  Wakeup(sleeplock);
  Release(lock);
}
```

释放睡眠锁：

- 获得它对应的自旋锁；
- 将locked域置0，表示释放睡眠锁；
- 调用wakeup函数唤醒所有申请该睡眠锁的进程；
- 释放自旋锁。

04

为什么 sleeplock 在获取到锁之后，直到释放锁的这段时间内，并没有像 spinlock 一样屏蔽中断?

- 如果获得睡眠锁的进程屏蔽中断，在持有锁的整个过程中CPU资源没有得到利用，效率低；

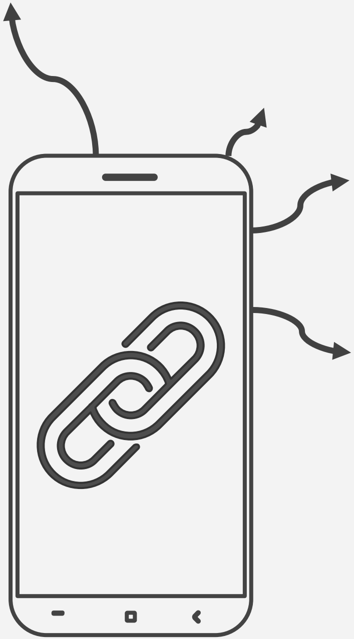
- 如果进程切换后屏蔽中断，且切换后的进程需要获得该睡眠锁，就会造成死锁的情况。

读 写 锁

01

读写锁的由来，基本原理，特性和使用场景

读写锁 Read-Write Lock



针对的是“读者-写者”问题

当存在多名读者时，没有任何理由拒绝其同时访问共享资源

核心思想就是将对共享资源的访问划分为读者和写者，读者不会修改数据，写者可能修改数据

因此多名读者可以同时访问该共享资源，写者必须确认直到没有读者或其他写者时才能够获得访问权

读写锁 Read-Write Lock 基本原理

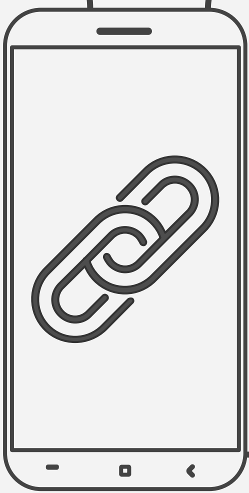
在申请对共享资源的访问权时

- 分别记录读者和写者的数量
- 需要额外的锁变量
- 在获取读写锁失败时将线程保存至某个等待队列

区分读者和写者两种身份
根据该记录对锁进行管理

保护读者/写者数量的记录

读写锁 Read-Write Lock



特性:“读-读共享、读-写互斥、写-写互斥”

- 多名读者能够同时获得对读写锁，实现对共享资源的访问
- 写者必须在没有其他读者或写者占有该锁时才能获得锁
- 读写锁在读加锁时，以读模式对它进行加锁的线程可以得到共享资源的访问权
- 读写锁在读加锁时，以写模式对它进行加锁的线程须等待直到所有读者释放锁
- 读写锁在写加锁时，以任何模式对它加锁的线程须等待直到当前写者释放锁

使用场景：任何对共享资源的读取频率远超过共享资源的修改频率的场景，例如数据库、订票系统等

02

spinlock类型相应的数据结构和支持的操作

Spinlock类型

数据结构

```
typedef struct {
  arch_rwlock_t raw_lock;
} rwlock_t;

typedef struct {
  u32 lock;
} arch_rwlock_t;
```

- 锁可能有三种状态：读者持有、写者持有、闲置
- 为0时表示没有进程持有它
- 为正表示有读者
- 为负表示有写者
- 此读写锁实现中没有等待队列，即使有一个写者正在等待读者释放此锁，后来的读者也会直接尝试获取此锁。

Spinlock类型支持的操作

1

void rwlock_init(rwlock_t)
初始化一个读写锁，其为闲置状态

2

读者获取一个读写锁，如果有写者正持有此锁，则忙等待直到锁被释放，如果锁闲置或者有其他读者持有此锁，则获取锁后返回；持有读写锁的过程中关闭内核抢占。
void read_lock(rwlock_t)

3

int read_trylock(rwlock_t)
读者尝试获取一个读写锁，如果有写者正持有此锁，则返回0；否则获取锁，并且返回1。

4

void read_unlock(rwlock_t)
读者释放一个读写锁，并且打开内核抢占。

Spinlock类型支持的操作

5

void write_lock(rwlock_t)
写者获取一个读写锁，如果锁正被人持有，则忙等待直到锁被释放，获取锁后返回；持有锁的过程中关闭内核抢占。

6

int write_trylock(rwlock_t)
写者尝试获取一个读写锁，如果锁正被人持有，则返回0，否则获取锁，并且返回1。

7

void write_unlock(rwlock_t)
写者释放一个读写锁，并且打开内核抢占。

Spinlock类型支持的操作

8

```
void write/read_lock_bh(rwlock_t)
void write/read_unlock_bh(rwlock_t)
```

写/读者持有锁的过程中，关闭软中断，解锁时打开软中断。

9

```
void write/read_lock_irq(rwlock_t)
void write/read_unlock_irq(rwlock_t)
```

写/读者持有锁的过程中，关闭硬中断，解锁时打开硬中断。

10

```
void write/read_lock_irqsave(rwlock_t, flags)
void write/read_unlock_irqrestore(rwlock_t, flags)
```

写/读者持有锁的过程中关闭硬中断并保存中断状态，开锁的时候将中断使能状态恢复到持有锁之前的状态。

03

信号量类型相应的数据结构和支持的操作

信号量类型

数据结构

```
struct rw_semaphore {
  atomic_long_t count;

  struct list_head wait_list;







  raw_spinlock_t wait_lock;
};
```

信号量

等待队列，若读者在抢锁时发现队列中有写者在自己前面等待，则直到其之前的写者拿到并释放锁之前都不会拿到锁。且此读写锁实现了写抢占，即后来的写者有可能比先来的写者先拿到读写锁

自旋锁，保护互斥访问

信号量类型支持的操作

1

`void down_read(struct rw_semaphore *sem)`

读者获取此读写锁。如果锁正由写者持有或者有写者先于自己申请抢锁，则此进程进入睡眠状态，直到获取锁后才被唤醒。

2

`void up_read(struct rw_semaphore *sem)`

读者释放此读写锁，如果释放此锁后此锁闲置，且正有写者在此读写锁上休眠，则唤醒此写者。

3

`void down_write(struct rw_semaphore *sem)`

写者获取此读写锁，如果此锁正由其他人持有，则休眠，被唤醒时会再次判断此锁是否由其他人持有，直到自己获得此锁为止。允许写抢占，无论此写者是否在队首，其都可能获得锁。

信号量类型支持的操作

4

`void down_write_killable(struct rw_semaphore *sem)`

写者获取此读写锁，相比于down_write接口，调用此接口表明的task可以被致命信号唤醒，如果被致命信号唤醒后锁仍然未闲置，其会唤醒下一个写者前的所有读者并退出。

5

`void up_write(struct rw_semaphore *sem)`

写者释放此读写锁，释放后唤醒其后的一个写者或者下一个写者前的所有读者。

6

`void downgrade_write(struct rw_semaphore *sem)`

将一个正持有此读写锁的写者降级为读者，并且唤醒下一个写者前的所有读者。

信号量类型支持的操作

7

```
int down_read_trylock(struct rw_semaphore *sem)
```

读者获取此读写锁。如果锁正由写者持有或者有写者先于自己申请抢锁，则此进程退出并返回0，否则获取锁并返回1。

8

```
int down_write_trylock(struct rw_semaphore *sem)
```

写者获取此读写锁。如果锁未闲置，则此进程退出并返回0，否则获取锁并返回1。

读 写 锁

PTHREAD

01

Pthread库提供的读写锁API

Pthread库提供的读写锁常用API

1.
```
int pthread_rwlock_init(
  pthread_rwlock_t *lock, const pthread_rwlockattr_t *attr
);  // Initialize a read/write lock object
```

2.
```
int pthread_rwlock_destroy(pthread_rwlock_t *lock);
// Destroy a read/write lock object
```

3.
```
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
// Lock a read/write lock for reading
```

4.
```
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
// Lock a read/write lock for writing
```

5.
```
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
// Unlock a read/write lock
```

Pthread库提供的读写锁其他API

1
```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
// Attempt to lock a read/write lock for reading
```

2
```
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
// Attempt to lock a read/write lock for writing
```

3
```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
// Initialize a read/write lock attribute object
```

4
```
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
 // Destroy a read/write lock attribute object
```

5
```
int pthread_rwlockattr_getpshared(
 const pthread_rwlockattr_t *attr, int *pshared
);//Retrieve process shared setting for rwlock attribute object
```

6
```
Int pthread_rwlockattr_setpshared(
  pthread_rwlockattr_t *attr, int pshared
); // Set process shared setting for the rwlock attribute object
```

02

benchmark程序比较 pthread_rwlock 和 pthread_mutex 的性能差异

多 读 者 场 景　　多 写 者 场 景　　读写事件驱动

场景A——众多读者访问共享资源

```
static char *newspaper_headlines[]={
    "5G, Your Next Big Upgrade",
    "Generation China",
    "Your Face, Your Password",
    "Blockchain Decoded",
    ......
};  // Pieces of news for today
```

假设这群读者均想看今天的报纸
但很遗憾，报纸只有1份

要么通过竞争抢夺报纸的阅读权（对应互斥锁）
要么平等共享抱团取暖（对应读写锁）

模拟读者完整读报
=>确定今日发生的热点新闻

同时假设读者可能需要重复该过程多次
=>模拟需要从共享资源中读取大量数据的情况，例如数据库

```
for (int t = 0; t < REPETITION_TIME; t++) {
  for (int i = 0; i < number_of_headlines; i++){
    char *headline = newspaper_headlines[i];
    for (int j = 0; headline[j] != '\0'; j++);
  }   // Read the headline
}   // Read for many times
```

场景A—众多读者访问共享资源

```
void *specific_type_of_reader(void *lock) {
  pthread_corresponding_type_of_lock(lock);
  // Read the headlines
  pthread_corresponding_type_of_unlock(lock);
  pthread_exit(NULL);
}   // Mutex sharing newspaper


clock_gettime(CLOCK_REALTIME, &start_sharing);
for (int id = 0; id < number_of_readers; id++){
  pthread_create(
    &readers[id], NULL, reader_type, lock_shared
  );  // Some news, please
}   // Some readers want to read the newspaper

for (int id = 0; id < number_of_readers; id++){
  pthread_join(readers[id], NULL);
}   // Wait for readers to finish reading
clock_gettime(CLOCK_REALTIME, &stop_sharing);
```

读者阅读的过程大致为：
获取锁→读报→释放锁→退出

在使用互斥锁和使用读写锁两种场景下，可分别对特定数量读者完成特定阅读量的时间进行时间测定

场景A 测试分析

（1）当阅读量较小时，程序性能主要由操作系统的调度方式所影响
互斥锁和读写锁所消耗的时间波动较大

```
500 mutex lock readers spent     56.6888 ms to read the headlines
500  r/w  lock readers spent     60.8139 ms to read the headlines
550 mutex lock readers spent     52.0219 ms to read the headlines
550  r/w  lock readers spent     49.2964 ms to read the headlines
600 mutex lock readers spent     44.2497 ms to read the headlines
600  r/w  lock readers spent     52.2068 ms to read the headlines
```

（2）当阅读量较大时，读写锁相对互斥锁由明显的性能优势

```
500 mutex lock readers spent  18665.6438 ms to read the headlines
500  r/w  lock readers spent   5239.9788 ms to read the headlines
550 mutex lock readers spent  20707.2398 ms to read the headlines
550  r/w  lock readers spent   5826.9530 ms to read the headlines
600 mutex lock readers spent  22783.9364 ms to read the headlines
600  r/w  lock readers spent   6434.9800 ms to read the headlines
```

（3）在两种阅读量设置下，读者数量均不是主要影响因素，至少线程数量在普通内存容量允许的范围内时不是

场景B—众多写者访问共享资源

众多写者与众多读者的测试思路类似

设想他们都想为未来的几部影片拟定Punchline
并假设每个写者可能会反复修改多次以获得较为满意的效果
用来模拟写大量数据的情景

```
for (int t = 0; t < REPETITION_TIME; t++) {
    for (int i = 0; i < number_of_punchlines; i++) {
        char *punchline = film_punchlines[i];
        for (int j = 0; j < BLANK_LETTERS; j++) {
            punchline[j] = (char)j;
        }   // Write punchlines
    }   // Write the punchline
}   // Write for many times
```

场景B 测试分析

(1) 写入数据量较大时，互斥锁性能比读写锁更加优越
因为读写锁需要额外的管理开销

```
# Massive Writing Test: REPETITION_TIME = 1000
     500 mutex lock writers spent    1595.7766 ms to write the punchlines
     500   r/w  lock writers spent    1706.2355 ms to write the punchlines
     520 mutex lock writers spent    1656.0960 ms to write the punchlines
     520   r/w  lock writers spent    1735.3913 ms to write the punchlines
     540 mutex lock writers spent    1720.4837 ms to write the punchlines
     540   r/w  lock writers spent    1814.6616 ms to write the punchlines
     560 mutex lock writers spent    1766.9189 ms to write the punchlines
     560   r/w  lock writers spent    1797.5816 ms to write the punchlines
     580 mutex lock writers spent    1805.9362 ms to write the punchlines
     580   r/w  lock writers spent    1869.4536 ms to write the punchlines
     600 mutex lock writers spent    1876.8174 ms to write the punchlines
     600   r/w  lock writers spent    1907.3053 ms to write the punchlines
```

场景B 测试分析

(2) 写入数据量较小时，互斥锁性能比读写锁更加优越

```
# Massive Writing Test: REPETITION_TIME = 10
     500 mutex lock writers spent    180.0902 ms to write the punchlines
     500  r/w  lock writers spent    201.8125 ms to write the punchlines
     520 mutex lock writers spent    167.7855 ms to write the punchlines
     520  r/w  lock writers spent    217.1113 ms to write the punchlines
     540 mutex lock writers spent    124.2517 ms to write the punchlines
     540  r/w  lock writers spent    155.8252 ms to write the punchlines
     560 mutex lock writers spent     79.9842 ms to write the punchlines
     560  r/w  lock writers spent    121.7671 ms to write the punchlines
     580 mutex lock writers spent    108.9668 ms to write the punchlines
     580  r/w  lock writers spent    325.3362 ms to write the punchlines
     600 mutex lock writers spent    339.1219 ms to write the punchlines
     600  r/w  lock writers spent    174.6084 ms to write the punchlines
```

*但因**调度因**素存在且工作负载较少，实际的运行时间通常比较接近，有时读写锁的实际运行速度会稍稍超过互斥锁的运行速度，但这和锁无关

场景C——事件驱动获取拿锁时间

测量互斥锁和读写锁的性能，如果回归本质站在线程的角度观察，需要测量获取锁的等待时间

沿用场景A和B中的思路，但现在读者和写者会随机出现，其出现概率随机控制

例如可以设置读者出现的概率为reader_possibility，则依此概率可以模拟在不同**读者-写者比例**情况下，读者和写者获取锁需要的时间。

```
for (int i = 0; i < NUMBER_OF_EVENTS; i++) {
       bool there_comes_a_reader = frand() < reader_possibility;
       if (there_comes_a_reader) number_of_reader += 1;
       else                      number_of_writer += 1;
       // Record the arrive of reader/writer
       pthread_create(
           &table[i].thread_id, NULL, there_comes_a_reader ?
           unnamed_reader : unnamed_writer, &table[i]
       );  // Generate corresponding events
   }   // Generate read/write events
```

场景C——事件驱动获取拿锁时间

```
void *unnamed_person(void *entry) {
    // Local variables goes here

    clock_gettime(CLOCK_MONOTONIC,
&start_waiting);
    // for mutex : pthread_mutex_lock
(&lock_shared);
    // for reader:
pthread_rwlock_rdlock(&lock_shared);
    // for writer:
pthread_rwlock_wrlock(&lock_shared);
    clock_gettime(CLOCK_MONOTONIC, &stop_waiting);

    // Read the shared resources

    // for mutex : pthread_mutex_unlock
(&lock_shared);
    // for rwlock:
pthread_rwlock_unlock(&lock_shared);

    // Calculate and record time spent

    pthread_exit(NULL);
}   // Unamed reader
```

在当前场景中
读者所需要做的
是随机读取共享资源中的信息
而写者在获取锁后
需要像大猩猩一样随机改变共享资源中的数据

在访问共享资源前
任何人均需要获取相应的锁
此时记录获取锁的时间。

场景C 测试分析

(1) 互斥锁在不同读者-写者比例下，各线程获取锁的时间均相近

```
Using mutex lock to protect shared buffer...
   110 readers waited for    2.542e+04 ms in total for lock (     231.1 ms/reader)
   890 writers waited for    1.955e+05 ms in total for lock (     219.6 ms/writer)
Using mutex lock to protect shared buffer...
   305 readers waited for    5.998e+04 ms in total for lock (     196.6 ms/reader)
   695 writers waited for    1.289e+05 ms in total for lock (     185.5 ms/writer)
Using mutex lock to protect shared buffer...
   503 readers waited for     9.05e+04 ms in total for lock (     179.9 ms/reader)
   497 writers waited for    9.063e+04 ms in total for lock (     182.4 ms/writer)
Using mutex lock to protect shared buffer...
   709 readers waited for    1.111e+05 ms in total for lock (     156.7 ms/reader)
   291 writers waited for    4.617e+04 ms in total for lock (     158.7 ms/writer)
Using mutex lock to protect shared buffer...
   891 readers waited for    1.205e+05 ms in total for lock (     135.3 ms/reader)
   109 writers waited for    1.324e+04 ms in total for lock (     121.4 ms/writer)
```

场景C 测试分析

（2） 读写锁在不同读者-写者比例下，读者获取锁的时间显著降低，而写者需要额外进行等待（读友好）

（3） 随着读者比例的增加，读者获取读写锁的时间逐渐降低，写者获取锁的时间逐渐增加

```
Using read write lock to protect shared buffer...
    96 readers waited for         4.824 ms in total for lock (   0.05025 ms/reader)
   904 writers waited for     3.054e+05 ms in total for lock (     337.8 ms/writer)
Using read write lock to protect shared buffer...
   298 readers waited for         12.35 ms in total for lock (   0.04143 ms/reader)
   702 writers waited for     2.769e+05 ms in total for lock (     394.4 ms/writer)
Using read write lock to protect shared buffer...
   507 readers waited for       0.05437 ms in total for lock ( 0.0001072 ms/reader)
   493 writers waited for     1.883e+05 ms in total for lock (       382 ms/writer)
Using read write lock to protect shared buffer...
   729 readers waited for        0.1081 ms in total for lock ( 0.0001483 ms/reader)
   271 writers waited for     9.413e+04 ms in total for lock (     347.4 ms/writer)
Using read write lock to protect shared buffer...
   904 readers waited for           0.1 ms in total for lock ( 0.0001107 ms/reader)
    96 writers waited for     3.261e+04 ms in total for lock (     339.7 ms/writer)
```

附1：场景A测试代码

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#include <unistd.h>
#include <time.h>
#include <pthread.h>

#define REPETITION_TIME 100000

#define READER_COUNT_LB 100
#define READER_COUNT_SP 100
#define READER_COUNT_UB 500

#if ((READER_COUNT_LB <= 0) \
  || (READER_COUNT_SP <= 0) \
  || (READER_COUNT_UB <= 0))
    #error "Fatal Error: Please Check"
#endif

typedef struct Benchmark {
    char lock_type[24];
    int lock_num_type;
    size_t number_of_readers;
    void *lock_held;
    time_t total_time_spent;
} Benchmark;

Benchmark myBenchmarks[] = {
    {"mutex lock", 'M', 100, NULL, 0},
    {" r/w  lock", 'R', 100, NULL, 0}
};  // Benchmarks

struct timespec start_sharing;
struct timespec stop_sharing;

static char *newspaper_headlines[] = {
    "5G, Your Next Big Upgrade",
    "Generation China",
    "Hacking the Apocalypse",
    "The Future of Funerals",
    "Tech Enabled",
    "Beyond Passwords",
    "Your Face, Your Password",
    "Follow the Money",
    "Blockchain Decoded",
    "Apple Watch Is Leading New Ideas"
};  // Pieces of news for today

const size_t number_of_headlines = \
    sizeof(newspaper_headlines) / sizeof(char *);

size_t number_of_benchmarks = \
    sizeof(myBenchmarks) / sizeof(Benchmark);
void *mutex_sharing_reader(void *mutex) {
    pthread_mutex_lock(mutex);

    for (int t = 0; t < REPETITION_TIME; t++) {
        for (int i = 0; i < number_of_headlines; i++) {
            char *headline = newspaper_headlines[i];
            for (int j = 0; headline[j] != '\0'; j++) ;
        }   // Read the headline
    }   // Read for 100 times

    pthread_mutex_unlock(mutex);

    pthread_exit(NULL);
}   // Mutex sharing newspaper

void *rw_sharing_reader(void *rwlock) {
    pthread_rwlock_rdlock(rwlock);

    for (int t = 0; t < REPETITION_TIME; t++) {
        for (int i = 0; i < number_of_headlines; i++) {
            char *headline = newspaper_headlines[i];
            for (int j = 0; headline[j] != '\0'; j++) ;
        }   // Read the headline
    }   // Read for 100 times

    pthread_rwlock_unlock(rwlock);

    pthread_exit(NULL);
}   // Generously sharing newapaper

void all_readers_start(int bench, void *(reader_type)(void *)) {
    int number_of_readers = myBenchmarks[bench].number_of_readers;
    void *lock_shared = myBenchmarks[bench].lock_held;

    pthread_t readers[number_of_readers];

    clock_gettime(CLOCK_REALTIME, &start_sharing);

    for (int id = 0; id < number_of_readers; id++) {
        pthread_create(
            &readers[id], NULL,
            reader_type, lock_shared
        );  // Some news, please
    }   // Some readers want to read the newspaper

    for (int id = 0; id < number_of_readers; id++) {
        pthread_join(readers[id], NULL);
    }   // Wait for readers to finish reading

    clock_gettime(CLOCK_REALTIME, &stop_sharing);

    myBenchmarks[bench].total_time_spent = \
        (stop_sharing.tv_nsec - start_sharing.tv_nsec)
      + (stop_sharing.tv_sec  - start_sharing.tv_sec )*1000000000;
}   // Helper function

int main(int argc, char **argv) {

    for (int reader_count = READER_COUNT_LB;
        reader_count <= READER_COUNT_UB; reader_count += READER_COUNT_SP) {

        for (int bench = 0; bench < number_of_benchmarks; bench++) {

            myBenchmarks[bench].number_of_readers = reader_count;

            switch (myBenchmarks[bench].lock_num_type) {
                case 'M': {

                    myBenchmarks[bench].lock_held = malloc(sizeof(pthread_mutex_t));
                    pthread_mutex_init(myBenchmarks[bench].lock_held, NULL);
                    all_readers_start(bench, mutex_sharing_reader);
                    pthread_mutex_destroy(myBenchmarks[bench].lock_held);

                }   break;
                case 'R': {

                    myBenchmarks[bench].lock_held = malloc(sizeof(pthread_rwlock_t));
                    pthread_rwlock_init(myBenchmarks[bench].lock_held, NULL);
                    all_readers_start(bench, rw_sharing_reader);
                    pthread_rwlock_destroy(myBenchmarks[bench].lock_held);

                }   break;
                default: break;
            }   // Mutex / Read-Write

            printf("%8lu %s readers spent %12.4lf ms to read the headlines\n",
                myBenchmarks[bench].number_of_readers,
                myBenchmarks[bench].lock_type,
                myBenchmarks[bench].total_time_spent/1000000.0
            );  // Print benchmark result

        }   // Test all benchmarks

    }   // Loop through a range of reader numbers

    return EXIT_SUCCESS;
}   // Simulate the scenario
```

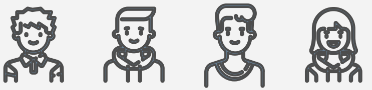
附2：场景B测试代码

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#include <unistd.h>
#include <time.h>
#include <pthread.h>

#define REPETITION_TIME 100

#define WRITER_COUNT_LB 500
#define WRITER_COUNT_SP 500
#define WRITER_COUNT_UB 2000

#define BLANK_LETTERS 128

#if ((WRITER_COUNT_LB <= 0) \
  || (WRITER_COUNT_SP <= 0) \
  || (WRITER_COUNT_UB <= 0) \
  || (BLANK_LETTERS   <= 0))
    #error "Fatal Error: Please Check"
#endif

typedef struct Benchmark {
    char lock_type[24];
    int lock_num_type;
    size_t number_of_writers;
    void *lock_held;
    time_t total_time_spent;
} Benchmark;

Benchmark myBenchmarks[] = {
    {"mutex lock", 'M', 100, NULL, 0},
    {" r/w  lock", 'R', 100, NULL, 0}
};  // Benchmarks

struct timespec start_sharing;
struct timespec stop_sharing;

const size_t number_of_punchlines = 10;
static char **film_punchlines;

size_t number_of_benchmarks = \
    sizeof(myBenchmarks) / sizeof(Benchmark);

void *mutex_sharing_writer(void *mutex) {
    pthread_mutex_lock(mutex);

    for (int t = 0; t < REPETITION_TIME; t++) {
        for (int i = 0; i < number_of_punchlines; i++) {
            char *punchline = film_punchlines[i];
            for (int j = 0; j < BLANK_LETTERS; j++) {
                punchline[j] = (char)j;
            }  // Write punchlines
        }  // Write the punchline
    }  // Write for 100 times

    pthread_mutex_unlock(mutex);

    pthread_exit(NULL);
}  // Mutex sharing newspaper

void *rw_sharing_writer(void *rwlock) {
    pthread_rwlock_wrlock(rwlock);

    for (int t = 0; t < REPETITION_TIME; t++) {
        for (int i = 0; i < number_of_punchlines; i++) {
            char *punchline = film_punchlines[i];
            for (int j = 0; j < BLANK_LETTERS; j++) {
                punchline[j] = (char)j;
            }  // Write punchlines
        }  // Write the punchline
    }  // Write for 100 times

    pthread_rwlock_unlock(rwlock);

    pthread_exit(NULL);
}  // Generously sharing newapaper

void all_writers_start(int bench, void *(writer_type)(void *)) {
    int number_of_writers = myBenchmarks[bench].number_of_writers;
    void *lock_shared = myBenchmarks[bench].lock_held;

    pthread_t writers[number_of_writers];

    clock_gettime(CLOCK_REALTIME, &start_sharing);

    for (int id = 0; id < number_of_writers; id++) {
        pthread_create(
            &writers[id], NULL,
            writer_type, lock_shared
        );  // Some news, please
    }  // Some writers want to write the punchline

    for (int id = 0; id < number_of_writers; id++) {
        pthread_join(writers[id], NULL);
    }  // Wait for writers to finish writing

    clock_gettime(CLOCK_REALTIME, &stop_sharing);

    myBenchmarks[bench].total_time_spent = \
        (stop_sharing.tv_nsec - start_sharing.tv_nsec)
      + (stop_sharing.tv_sec  - start_sharing.tv_sec )*1000000000;
}  // Helper function

int main(int argc, char **argv) {

    char *the_film_punchlines[number_of_punchlines];
    film_punchlines = (char **)&the_film_punchlines;

    for (int i = 0; i < number_of_punchlines; i++) {
        the_film_punchlines[i] = malloc(BLANK_LETTERS*sizeof(char));
        if (the_film_punchlines[i] == NULL) goto CLEAN_UP;
    }  // Initialize

    for (int writer_count = WRITER_COUNT_LB;
        writer_count <= WRITER_COUNT_UB; writer_count += WRITER_COUNT_SP) {

        for (int bench = 0; bench < number_of_benchmarks; bench++) {

            myBenchmarks[bench].number_of_writers = writer_count;

            switch (myBenchmarks[bench].lock_num_type) {
                case 'M': {

                    myBenchmarks[bench].lock_held = malloc(sizeof(pthread_mutex_t));
                    pthread_mutex_init(myBenchmarks[bench].lock_held, NULL);
                    all_writers_start(bench, mutex_sharing_writer);
                    pthread_mutex_destroy(myBenchmarks[bench].lock_held);

                }   break;
                case 'R': {

                    myBenchmarks[bench].lock_held = malloc(sizeof(pthread_rwlock_t));
                    pthread_rwlock_init(myBenchmarks[bench].lock_held, NULL);
                    all_writers_start(bench, rw_sharing_writer);
                    pthread_rwlock_destroy(myBenchmarks[bench].lock_held);

                }   break;
                default: break;
            }  // Mutex / Read-Write

            printf("%8lu %s writers spent %12.4lf ms to write the punchlines\n",
                myBenchmarks[bench].number_of_writers,
                myBenchmarks[bench].lock_type,
                myBenchmarks[bench].total_time_spent/1000000.0
            );  // Print benchmark result

        }  // Test all benchmarks

    }  // Loop through a range of writer numbers

CLEAN_UP:
    for (int i = 0; i < number_of_punchlines; i++) {
        free(the_film_punchlines[i]);
    }  // Free memory

    return EXIT_SUCCESS;
}  // Simulate the scenario
```

附3：场景C测试代码

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#include <unistd.h>
#include <time.h>
#include <pthread.h>

// #define USE_PTHREAD_RWLOCK_TEST

#define frand() ((rand() % 10000)/10000.0)

#define NUMBER_OF_EVENTS 1000
#define SIZE_OF_BUFFER   1024
#define REPETITION_TIME 10

#define READER_ODDS_LB 0.10f
#define READER_ODDS_SP 0.10f
#define READER_ODDS_UB 1.00f

static size_t number_of_reader;
static size_t number_of_writer;
static char *all_shared_buffer;

typedef struct table_entry {
    pthread_t thread_id;
    time_t time_waiting_for_lock;
    int thread_num_type;
} table_entry_t;

#ifndef USE_PTHREAD_RWLOCK_TEST
    pthread_mutex_t lock_shared;
#else
    pthread_rwlock_t lock_shared;
#endif

void *unnamed_reader(void *entry) {
    struct timespec start_waiting, stop_waiting;
    table_entry_t *my_entry = entry; char ch;

    clock_gettime(CLOCK_MONOTONIC, &start_waiting);

    #ifndef USE_PTHREAD_RWLOCK_TEST
        pthread_mutex_lock(&lock_shared);
    #else
        pthread_rwlock_rdlock(&lock_shared);
    #endif

    clock_gettime(CLOCK_MONOTONIC, &stop_waiting);

    for (int t = 0; t < REPETITION_TIME; t++) {
        for (int i = 0; i < SIZE_OF_BUFFER; i++) {
            ch = all_shared_buffer[rand() % SIZE_OF_BUFFER];
        }   // Read characters in the buffer
    }   // Read for many times

    #ifndef USE_PTHREAD_RWLOCK_TEST
        pthread_mutex_unlock(&lock_shared);
    #else
        pthread_rwlock_unlock(&lock_shared);
    #endif

    my_entry->time_waiting_for_lock = \
        (stop_waiting.tv_nsec - start_waiting.tv_nsec)
      + (stop_waiting.tv_sec  - start_waiting.tv_sec )*1000000000;

    pthread_exit(NULL);
}   // Unamed reader

void *unnamed_writer(void *entry) {
    struct timespec start_waiting, stop_waiting;
    table_entry_t *my_entry = entry;

    clock_gettime(CLOCK_MONOTONIC, &start_waiting);

    #ifndef USE_PTHREAD_RWLOCK_TEST
        pthread_mutex_lock(&lock_shared);
    #else
        pthread_rwlock_wrlock(&lock_shared);
    #endif

    clock_gettime(CLOCK_MONOTONIC, &stop_waiting);

    for (int t = 0; t < REPETITION_TIME; t++) {
        for (int i = 0; i < SIZE_OF_BUFFER; i++) {
            all_shared_buffer[rand() % SIZE_OF_BUFFER] = rand() % 57 + 65;
        }   // Write characters in the buffer
    }   // Write for many times

    #ifndef USE_PTHREAD_RWLOCK_TEST
        pthread_mutex_unlock(&lock_shared);
    #else
        pthread_rwlock_unlock(&lock_shared);
    #endif

    my_entry->time_waiting_for_lock = \
        (stop_waiting.tv_nsec - start_waiting.tv_nsec)
      + (stop_waiting.tv_sec  - start_waiting.tv_sec )*1000000000;

    pthread_exit(NULL);
}   // Unamed writer

int main(int argc, char **argv) {
    srand(time(NULL) + 42*getpid());

    float reader_possibility;
    double readers_waiting, writers_waiting;

    all_shared_buffer = malloc(SIZE_OF_BUFFER*sizeof(char));
    if (all_shared_buffer == NULL) exit(EXIT_FAILURE);

    table_entry_t table[NUMBER_OF_EVENTS];

    for (reader_possibility = READER_ODDS_LB;
        reader_possibility < READER_ODDS_UB;
        reader_possibility += READER_ODDS_SP) {

        #ifndef USE_PTHREAD_RWLOCK_TEST
            pthread_mutex_init(&lock_shared, NULL);
        #else
            pthread_rwlock_init(&lock_shared, NULL);
        #endif

        number_of_reader = number_of_writer = 0;
        readers_waiting = writers_waiting = 0.0f;

        for (int i = 0; i < NUMBER_OF_EVENTS; i++) {
            bool there_comes_a_reader = frand() < reader_possibility;

            if (there_comes_a_reader) number_of_reader += 1;
            else                      number_of_writer += 1;

            table[i].thread_num_type = there_comes_a_reader;

            pthread_create(
                &table[i].thread_id, NULL, there_comes_a_reader ?
                unnamed_reader : unnamed_writer, &table[i]
            );  // Generate corresponding events

        }   // Generate events

        for (int i = 0; i < NUMBER_OF_EVENTS; i++) {
            pthread_join(table[i].thread_id, NULL);
        }   // Wait for events to complete

        #ifndef USE_PTHREAD_RWLOCK_TEST
            pthread_mutex_destroy(&lock_shared);
        #else
            pthread_rwlock_destroy(&lock_shared);
        #endif

        for (int i = 0; i < NUMBER_OF_EVENTS; i++) {
            if (table[i].thread_num_type)
                readers_waiting += table[i].time_waiting_for_lock/1000000.0;
            else writers_waiting += table[i].time_waiting_for_lock/1000000.0;
        }   // Wait for events to complete

        #ifndef USE_PTHREAD_RWLOCK_TEST
            printf("Using mutex lock to protect shared buffer...\n");
        #else
            printf("Using read write lock to protect shared buffer...\n");
        #endif

        printf("%6lu readers waited for %12.4g ms in total for lock (%10.4g ms/reader)\n",
            number_of_reader, readers_waiting, readers_waiting/number_of_reader);
        printf("%6lu writers waited for %12.4g ms in total for lock (%10.4g ms/writer)\n",
            number_of_writer, writers_waiting, writers_waiting/number_of_writer);
    }   // Traverse reader event possibility

    free(all_shared_buffer);
    return EXIT_SUCCESS;
}   // Timing lock acquisition
```

THANKS!



参　考

- ✓ 《xv6配套讲义》
- ✓ https://rootreturn0.github.io/2019/08/31/xv6-管道/
- ✓ 关于内存屏障：https://blog.csdn.net/xujianqun/article/details/7800813
- ✓ 关于内核抢占与本地中断：https://blog.csdn.net/woshijidutu/article/details/68952702