- **Read Access** – жодний потік не доступається до writing
- **Write Access** – no reading, no writing

```
public class ReadWriteLock{

  private int readers       = 0;
  private int writers       = 0;
  private int writeRequests = 0;

  public synchronized void lockRead() throws InterruptedException{
    while(writers > 0 || writeRequests > 0){
      wait();
    }
    readers++;
  }

  public synchronized void unlockRead(){
    readers--;
    notifyAll();
  }

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;

    while(readers > 0 || writers > 0){
      wait();
    }
    writeRequests--;
    writers++;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writers--;
    notifyAll();
  }
}
```

1 – Thread1 отримав один запит reading
2 – Thread2 отримав запит writing, але він відхилений, оскільки існує запит reading
3 – Thread1 отримав reading reentrance , який блокується через попередній

У випадку, якщо потік може отримати reading запит (без write-request) або вже має запит reading (незалежно від write-request), потік може гарантувати reading reentrance.

```
public class ReadWriteLock{

  private Map<Thread, Integer> readingThreads =
      new HashMap<Thread, Integer>();

  private int writers        = 0;
  private int writeRequests  = 0;

  public synchronized void lockRead() throws InterruptedException{
    Thread callingThread = Thread.currentThread();
    while(! canGrantReadAccess(callingThread)){
      wait();
    }

    readingThreads.put(callingThread,
       (getAccessCount(callingThread) + 1));
  }


  public synchronized void unlockRead(){
    Thread callingThread = Thread.currentThread();
    int accessCount = getAccessCount(callingThread);
    if(accessCount == 1){ readingThreads.remove(callingThread); }
    else { readingThreads.put(callingThread, (accessCount -1)); }
    notifyAll();
  }


  private boolean canGrantReadAccess(Thread callingThread){
    if(writers > 0)             return false;
    if(isReader(callingThread) return true;
    if(writeRequests > 0)       return false;
    return true;
  }
```

```
  private int getReadAccessCount(Thread callingThread){
    Integer accessCount = readingThreads.get(callingThread);
    if(accessCount == null) return 0;
    return accessCount.intValue();
  }

  private boolean isReader(Thread callingThread){
    return readingThreads.get(callingThread) != null;
  }

}
```

- Writing reentrance працює тільки у випадку вже існуючого writing доступу.

```
public class ReadWriteLock{

    private Map<Thread, Integer> readingThreads =
        new HashMap<Thread, Integer>();

    private int writeAccesses    = 0;
    private int writeRequests    = 0;
    private Thread writingThread = null;

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
    Thread callingThread = Thread.currentThread();
    while(! canGrantWriteAccess(callingThread)){
      wait();
    }
    writeRequests--;
    writeAccesses++;
    writingThread = callingThread;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writeAccesses--;
    if(writeAccesses == 0){
      writingThread = null;
    }
    notifyAll();
  }

  private boolean canGrantWriteAccess(Thread callingThread){
    if(hasReaders())             return false;
    if(writingThread == null)    return true;
    if(!isWriter(callingThread)) return false;
    return true;
  }
```

```
  private boolean hasReaders(){
    return readingThreads.size() > 0;
  }

  private boolean isWriter(Thread callingThread){
    return writingThread == callingThread;
  }
}
```

- Повна reentrance реалізація:

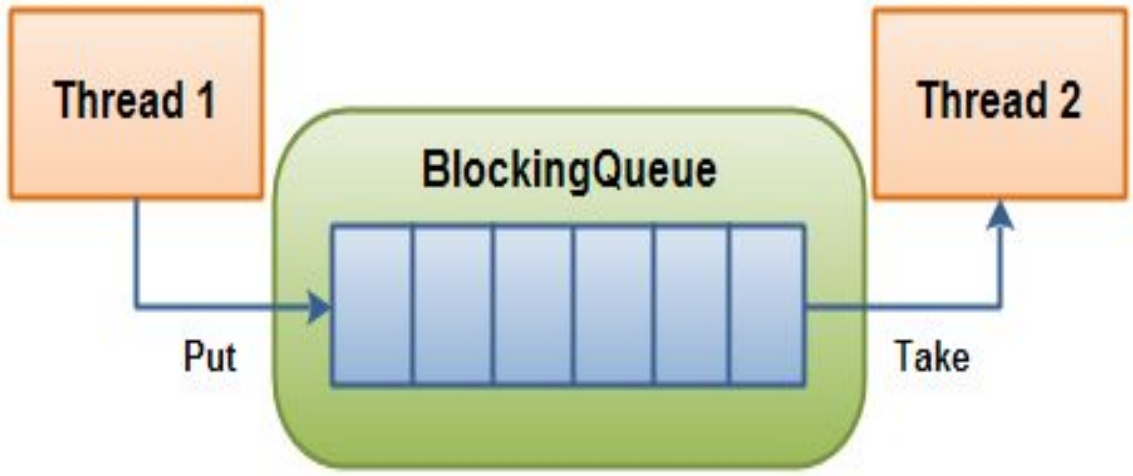
```
public class ReadWriteLock{

  private Map<Thread, Integer> readingThreads =
       new HashMap<Thread, Integer>();

   private int writeAccesses    = 0;
   private int writeRequests    = 0;
   private Thread writingThread = null;


  public synchronized void lockRead() throws InterruptedException{
    Thread callingThread = Thread.currentThread();
    while(! canGrantReadAccess(callingThread)){
      wait();
    }

    readingThreads.put(callingThread,
     (getReadAccessCount(callingThread) + 1));
  }

  private boolean canGrantReadAccess(Thread callingThread){
    if( isWriter(callingThread) ) return true;
    if( hasWriter()             ) return false;
    if( isReader(callingThread) ) return true;
    if( hasWriteRequests()      ) return false;
    return true;
  }


  public synchronized void unlockRead(){
    Thread callingThread = Thread.currentThread();
    if(!isReader(callingThread)){
      throw new IllegalMonitorStateException("Calling Thread does not" +
        " hold a read lock on this ReadWriteLock");
    }
```

```
  int accessCount = getReadAccessCount(callingThread);
  if(accessCount == 1){ readingThreads.remove(callingThread); }
  else { readingThreads.put(callingThread, (accessCount -1)); }
  notifyAll();
}

public synchronized void lockWrite() throws InterruptedException{
  writeRequests++;
  Thread callingThread = Thread.currentThread();
  while(! canGrantWriteAccess(callingThread)){
    wait();
  }
  writeRequests--;
  writeAccesses++;
  writingThread = callingThread;
}

public synchronized void unlockWrite() throws InterruptedException{
  if(!isWriter(Thread.currentThread()){
    throw new IllegalMonitorStateException("Calling Thread does not" +
      " hold the write lock on this ReadWriteLock");
  }
  writeAccesses--;
  if(writeAccesses == 0){
    writingThread = null;
  }
  notifyAll();
}

private boolean canGrantWriteAccess(Thread callingThread){
  if(isOnlyReader(callingThread))    return true;
  if(hasReaders())                   return false;
  if(writingThread == null)          return true;
  if(!isWriter(callingThread))       return false;
```

```
    return true;
  }


  private int getReadAccessCount(Thread callingThread){
    Integer accessCount = readingThreads.get(callingThread);
    if(accessCount == null) return 0;
    return accessCount.intValue();
  }


  private boolean hasReaders(){
    return readingThreads.size() > 0;
  }

  private boolean isReader(Thread callingThread){
    return readingThreads.get(callingThread) != null;
  }

  private boolean isOnlyReader(Thread callingThread){
    return readingThreads.size() == 1 &&
           readingThreads.get(callingThread) != null;
  }

  private boolean hasWriter(){
    return writingThread != null;
  }

  private boolean isWriter(Thread callingThread){
    return writingThread == callingThread;
  }
```

Blocking Queue

Це черга, яка блокується у випадку, коли ви надсилаєте запит вийти з неї, а вона порожня, або хочете долучитись до повної.



A BlockingQueue with one thread putting into it, and another thread taking from it.

```
public class BlockingQueue {

  private List queue = new LinkedList();
  private int  limit = 10;

  public BlockingQueue(int limit){
    this.limit = limit;
  }


  public synchronized void enqueue(Object item)
  throws InterruptedException  {
    while(this.queue.size() == this.limit) {
      wait();
    }
    if(this.queue.size() == 0) {
      notifyAll();
    }
    this.queue.add(item);
  }


  public synchronized Object dequeue()
  throws InterruptedException{
    while(this.queue.size() == 0){
      wait();
    }
    if(this.queue.size() == this.limit){
      notifyAll();
    }

    return this.queue.remove(0);
  }

}
```

Semaphores

Використовується у випадку для перевірки доступності перед використанням або для уникання пропущених запитів.

```
public class Semaphore {
  private boolean signal = false;

  public synchronized void take() {
    this.signal = true;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(!this.signal) wait();
    this.signal = false;
  }

}
```

Два потоки повідомляють один одного:

```
Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();
sender.start();
```

```
public class SendingThread {
  Semaphore semaphore = null;

  public SendingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      //do something, then signal
      this.semaphore.take();

    }
  }
}
```

```
public class RecevingThread {
  Semaphore semaphore = null;

  public ReceivingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      this.semaphore.release();
      //receive signal, then do something...
    }
  }
}
```

- Рахунок сигналів, відправлених методом take()

```
public class CountingSemaphore {
  private int signals = 0;

  public synchronized void take() {
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }

}
```