The LockManager App Case Study: Server Structure & Functionality (Part 3)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

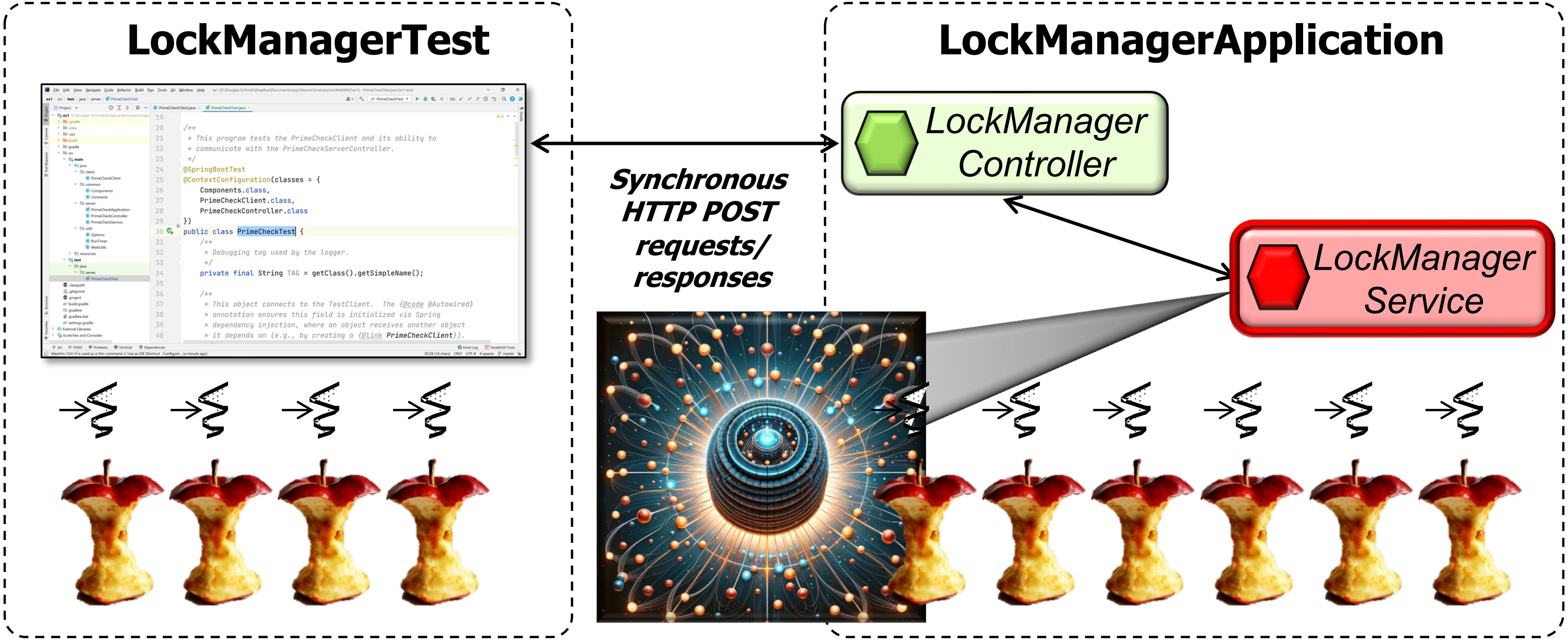
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- This lesson gives an overview of the semaphore algorithm implemented using Java ArrayBlockingQueue



```java

/**
 * This program tests the PrimeCheckClient and its ability to
 * communicate with the PrimeCheckServerController.
 */
@SpringBootTest
@ContextConfiguration(classes = {
    Components.class,
    PrimeCheckClient.class,
    PrimeCheckController.class
})
public class PrimeCheckTest {
    /**
     * Debugging tag used by the logger.
     */
    private final String TAG = getClass().getSimpleName();

    /**
     * This object connects to the TestClient.  The {@code @Autowired}
     * annotation ensures this field is initialized via Spring
     * dependency injection, where an object receives another object
     * it depends on (e.g., by creating a {@link PrimeCheckClient}).
```

See WebMVC/ex5/src/main/java/edu/vandy/lockmanager/server

The ArrayBlockingQueue Semaphore Algorithm

The ArrayBlockingQueue Semaphore Algorithm

- LockManagerService uses ArrayBlockingQueue to manage a fixed # of permits/locks that mediate access to a shared resource

See en.wikipedia.org/wiki/Semaphore_(programming)

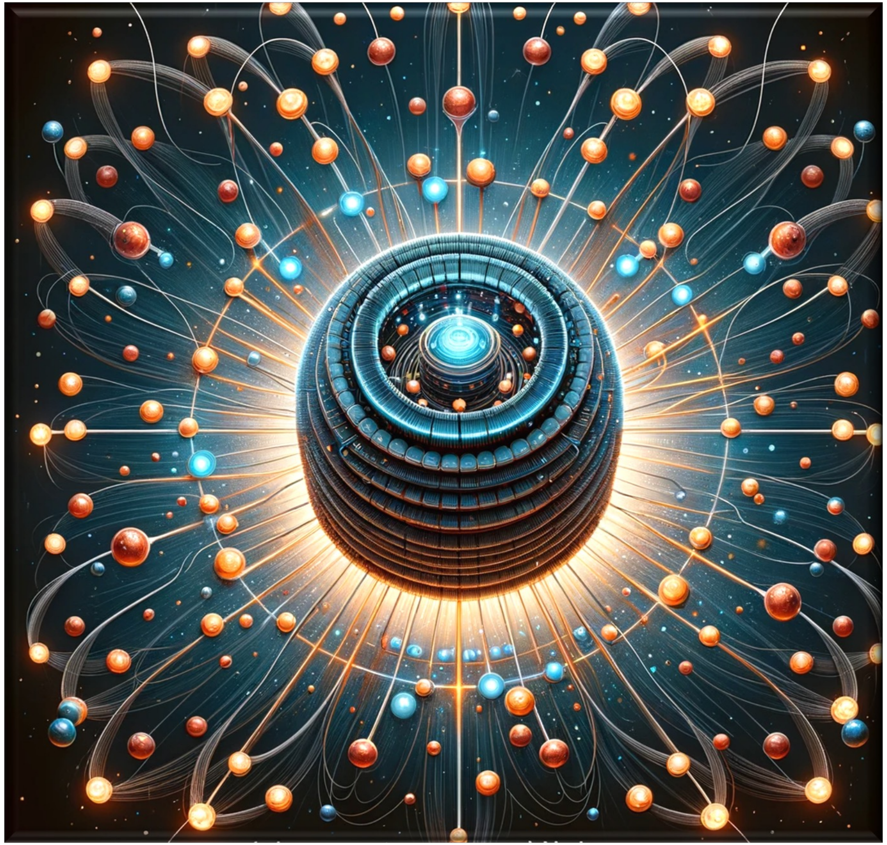
The ArrayBlockingQueue Semaphore Algorithm

- LockManagerService uses ArrayBlockingQueue to manage a fixed # of permits/locks that mediate access to a shared resource
  - This fixed capacity limits the # of concurrent accesses

See www.visitnewportbeach.com/beaches-and-parks/corona-del-mar-state-beach

The ArrayBlockingQueue Semaphore Algorithm

- LockManagerService uses Array BlockingQueue to manage a fixed # of permits/locks that mediate access to a shared resource
  - This fixed capacity limits the # of concurrent accesses
  - This queue is suitable for managing locks in multi-threaded programs
    - Ensures thread-safety & atomic acquire() & release() operations



The ArrayBlockingQueue Semaphore Algorithm

- **Initialization**
  - Create an ArrayBlockingQueue with a capacity equal to the # of permits

```
LockManager create(Integer permits) {
  var availableLocks = new
    ArrayBlockingQueue<Lock>
      (permits, true);

  availableLocks.addAll
    (makeLocks(permits));

  var lockManager = new LockManager
    (generateUniqueId(), permits);

  mLockManagerMap.put
    (lockManager, availableLocks);

  return lockManager; ...
```

The ArrayBlockingQueue Semaphore Algorithm

- **Initialization**
  - Create an ArrayBlockingQueue with a capacity equal to the # of permits
  - The queue is initialized with fairness set to true
    - Threads acquire locks in the order requested, preventing starvation

```
LockManager create(Integer permits) {
  var availableLocks = new
    ArrayBlockingQueue<Lock>
      (permits, true);

  availableLocks.addAll
    (makeLocks(permits));

  var lockManager = new LockManager
    (generateUniqueId(), permits);

  mLockManagerMap.put
    (lockManager, availableLocks);

  return lockManager; ...
```

See docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

The ArrayBlockingQueue Semaphore Algorithm

- **Initialization**
  - Create an ArrayBlockingQueue with a capacity equal to the # of permits
  - The queue is initialized with fairness set to true
  - This queue is filled with Lock objects
    - Each represents a permit

```
LockManager create(Integer permits) {
  var availableLocks = new
    ArrayBlockingQueue<Lock>
      (permits, true);

  availableLocks.addAll
    (makeLocks(permits));

  var lockManager = new LockManager
    (generateUniqueId(), permits);

  mLockManagerMap.put
    (lockManager, availableLocks);

  return lockManager; ...
```

The ArrayBlockingQueue Semaphore Algorithm

- **Initialization**
  - Create an ArrayBlockingQueue with a capacity equal to the # of permits
  - The queue is initialized with fairness set to true
  - This queue is filled with Lock objects
  - A LockManager keeps track of allocation ArrayBlockingQueue objects

```
LockManager create(Integer permits) {
  var availableLocks = new
    ArrayBlockingQueue<Lock>
      (permits, true);

  availableLocks.addAll
    (makeLocks(permits));

  var lockManager = new LockManager
    (generateUniqueId(), permits);

  mLockManagerMap.put
    (lockManager, availableLocks);

  return lockManager; ...
```

The ArrayBlockingQueue Semaphore Algorithm

- **Initialization**
  - Create an ArrayBlockingQueue with a capacity equal to the # of permits
  - The queue is initialized with fairness set to true
  - This queue is filled with Lock objects
  - A LockManager keeps track of allocation ArrayBlockingQueue objects
  - LockManager is returned to the client to differentiate each of the semaphore instances

```
LockManager create(Integer permits) {
  var availableLocks = new
    ArrayBlockingQueue<Lock>
      (permits, true);

  availableLocks.addAll
    (makeLocks(permits));

  var lockManager = new LockManager
    (generateUniqueId(), permits);

  mLockManagerMap.put
    (lockManager, availableLocks);

  return lockManager; ...
```

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (1)**

```
@Async public void acquire
   (LockManager lockManager,
    Callback callback) {
  var availableLocks =
    mLockManagerMap
       .get(lockManager);
  ...

  tryAcquire(callback,
                availableLocks);
  ...
}
```

Called by LockManagerController to acquire just a single lock

The ArrayBlockingQueue Semaphore Algorithm

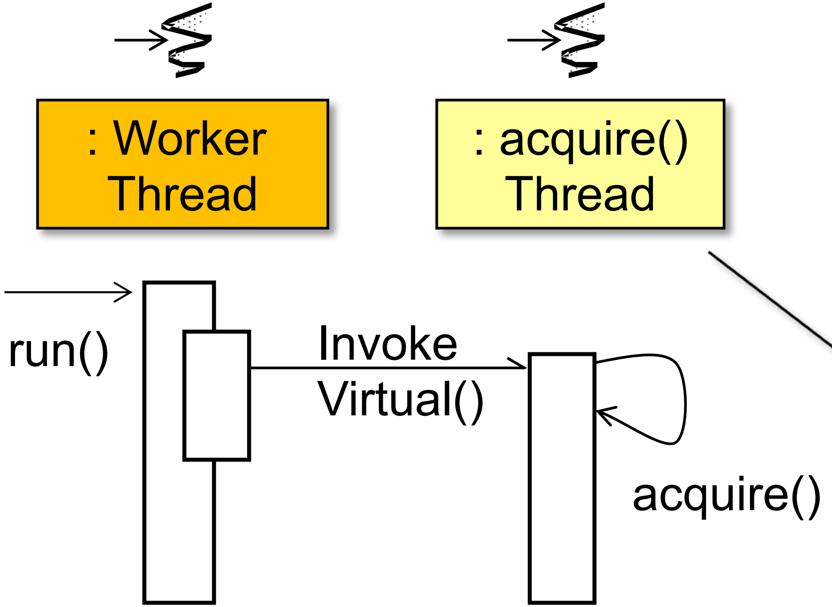
- **Async Acquire Operation (1)**
  - This acquire() method is marked with @Async

```
@Async public void acquire
   (LockManager lockManager,
    Callback callback) {
   var availableLocks =
      mLockManagerMap
         .get(lockManager);
   ...

   tryAcquire(callback,
              availableLocks);
   ...
}
```

See www.baeldung.com/spring-async

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (1)**
  - This acquire() method is marked with @Async



```
@Async public void acquire
   (LockManager lockManager,
    Callback callback) {
  var availableLocks =
    mLockManagerMap
       .get(lockManager);
  ...

  tryAcquire(callback,
             availableLocks);
  ...
}
```

@Async indicates it runs in a back ground (virtual) thread, separate from the HTTP worker thread

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (1)**
  - This acquire() method is marked with @Async
    - acquire() thus doesn't block the calling thread while waiting for a lock to become available

```
@Async public void acquire
   (LockManager lockManager,
    Callback callback) {
  var availableLocks =
     mLockManagerMap
        .get(lockManager);
  ...

  tryAcquire(callback,
                  availableLocks);
  ...
}
```

See www.baeldung.com/spring-async

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (1)**
  - This acquire() method is marked with @Async
  - The acquire() method first tries to obtain a lock by polling the ArrayBlockingQueue

```
void tryAcquire(Callback callback,
                ArrayBlockingQueue<Lock>
                  availableLocks) {
  var lock = availableLocks.poll();

  if (lock != null)
    ...
  else
    lock = availableLocks.take();

  callback.onSuccess(lock);
}
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html#poll

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (1)**
  - This acquire() method is marked with @Async
  - The acquire() method first tries to obtain a lock by polling the ArrayBlockingQueue
    - If a Lock is available, the non-blocking acquire is successful

```
void tryAcquire(Callback callback,
                ArrayBlockingQueue<Lock>
                  availableLocks) {
  var lock = availableLocks.poll();

  if (lock != null)
    ...
  else
    lock = availableLocks.take();

  callback.onSuccess(lock);
}
```

The ArrayBlockingQueue Semaphore Algorithm

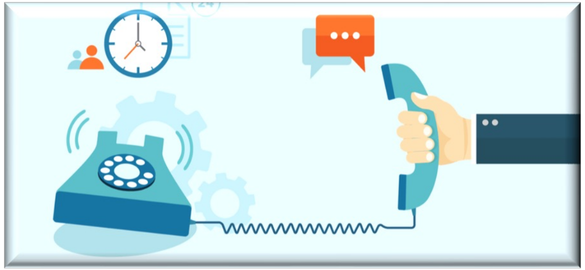
- **Async Acquire Operation (1)**
  - This acquire() method is marked with @Async
  - The acquire() method first tries to obtain a lock by polling the ArrayBlockingQueue
    - If a Lock is available, the non-blocking acquire is successful
    - If no Lock is available, the service blocks by calling take() to wait for a Lock

```
void tryAcquire(Callback callback,
                ArrayBlockingQueue<Lock>
                  availableLocks) {
  var lock = availableLocks.poll();

  if (lock != null)
    ...
  else
    lock = availableLocks.take();

  callback.onSuccess(lock);
}
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html#take

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (1)**
  - This acquire() method is marked with @Async
  - The acquire() method first tries to obtain a lock by polling the ArrayBlockingQueue
  - Upon successfully acquiring a lock the service notifies the caller through a callback interface



```
void tryAcquire(Callback callback,
                ArrayBlockingQueue<Lock>
                  availableLocks) {
  var lock = availableLocks.poll();

  if (lock != null)
    ...
  else
    lock = availableLocks.take();

  callback.onSuccess(lock);
}
```

See en.wikipedia.org/wiki/Callback_(computer_programming)

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**

```
DeferredResult<List<Lock>> acquire
   (LockManager lockManager,
    int permits) {
   var result = new DeferredResult
     <List<Lock>>();
   ...

   mExecutor.submit
     (getRunnable(permits,
                  availableLocks,
                  result));
   ...

   return result;
}
```

Called by LockManagerController to acquire multiple lock permits

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
    - Holds the future result of the lock acquisition process

```
DeferredResult<List<Lock>> acquire
  (LockManager lockManager,
   int permits) {
  var result = new DeferredResult
    <List<Lock>>();
  ...

  mExecutor.submit
    (getRunnable(permits,
                 availableLocks,
                 result));
  ...

  return result;
}
```

See springframework/web/context/request/async/DeferredResult.html

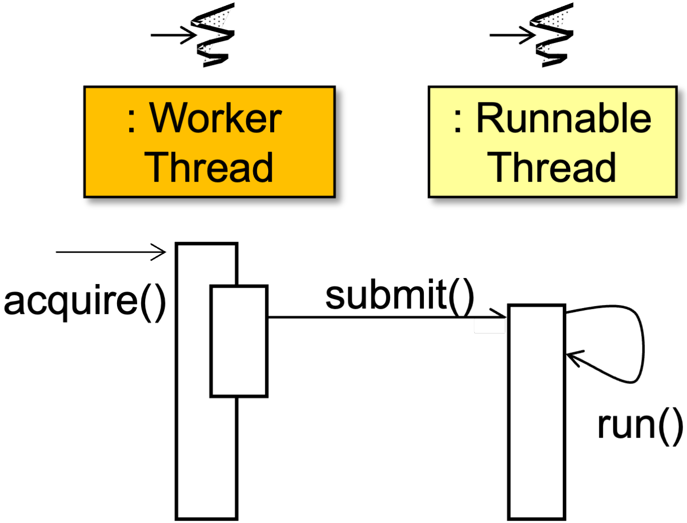
The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
    - Holds the future result of the lock acquisition process
    - Allow acquire() to return ASAP

```
DeferredResult<List<Lock>> acquire
   (LockManager lockManager,
    int permits) {
   var result = new DeferredResult
     <List<Lock>>();
   ...

   mExecutor.submit
     (getRunnable(permits,
                  availableLocks,
                  result));
   ...

   return result;
}
```

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
    - Holds the future result of the lock acquisition process
    - Allow acquire() to return ASAP



```
DeferredResult<List<Lock>> acquire
   (LockManager lockManager,
    int permits) {
  var result = new DeferredResult
    <List<Lock>>();
  ...

  mExecutor.submit
    (getRunnable(permits,
                 availableLocks,
                 result));

  ...

  return result;
}
```

Acquire permits in the background

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit

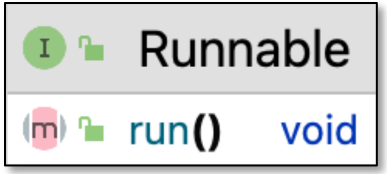
The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits

```
DeferredResult<List<Lock>> acquire
   (LockManager lockManager,
    int permits) {
  var result = new DeferredResult
     <List<Lock>>();
  ...

  mExecutor.submit
    (getRunnable(permits,
                 availableLocks,
                 result));
  ...

  return result;
}
```

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
    - Factory returns a Runnable



Runnable

run() void

```
Runnable getRunnable(int permits,
                 ArrayBlockingQueue<Lock>
                      availLocks,
                 DeferredResult<List<Lock>>
                   result) {
  return () -> {
    var locks = new ArrayList
      <Lock>(permits);
    while (tryAcquire
      (availLocks, locks)
         != permits)
      continue;
    result.setResult(locks);
  }
}
```

See docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
    - Factory returns a Runnable
    - A loop tries acquiring required # of permits by polling queue

```
Runnable getRunnable(int permits,
         ArrayBlockingQueue<Lock>
              availLocks,
         DeferredResult<List<Lock>>
           result) {
  return () -> {
    var locks = new ArrayList
      <Lock>(permits);
    while (tryAcquire
      (availLocks, locks)
         != permits)
      continue;
    result.setResult(locks);
  }
}
```

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
    - Factory returns a Runnable
    - A loop tries acquiring required # of permits by polling queue

```
Integer tryAcquire
    (ArrayBlockingQueue<Lock>
     availLocks, List<Lock> locks){
  var lock = availLocks.poll();

  if (lock != null) {
    locks.add(lock);
    return locks.size();
  } else {
    locks.forEach(locks::offer);
    locks.clear();
    return 0;
  }
```

Ensure that task either acquires all required permits or none, preventing partial acquisitions that could lead to deadlocks or resource starvation

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
    - Factory returns a Runnable
    - A loop tries acquiring required # of permits by polling queue

Each successful poll adds a lock to the list of acquired locks & return current size of the locks

```
Integer tryAcquire
    (ArrayBlockingQueue<Lock>
     availLocks, List<Lock> locks){
  var lock = availLocks.poll();

  if (lock != null) {
    locks.add(lock);
    return locks.size();
  } else {
    locks.forEach(locks::offer);
    locks.clear();
    return 0;
  }
}
```

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
    - Factory returns a Runnable
    - A loop tries acquiring required # of permits by polling queue

```
Integer tryAcquire
      (ArrayBlockingQueue<Lock>
       availLocks, List<Lock> locks){
   var lock = availLocks.poll();

   if (lock != null) {
      locks.add(lock);
      return locks.size();
   } else {
      locks.forEach(locks::offer);
      locks.clear();
      return 0;
   }
}
```

If lock can't be acquired, all locks already acquired are returned to queue, the list of locks is cleared, & the caller will then try again

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
    - Factory returns a Runnable
    - A loop tries acquiring required # of permits by polling queue

```
Runnable getRunnable(int permits,
                     ArrayBlockingQueue<Lock>
                          availLocks,
                     DeferredResult<List<Lock>>
                       result) {
  return () -> {
    var locks = new ArrayList
      <Lock>(permits);
    while (tryAcquire
      (availLocks, locks)
         != permits)
      continue;
    result.setResult(locks);
  }
}
```

Loop continues until all permits are acquired!

See en.wikipedia.org/wiki/Non-blocking_algorithm

The ArrayBlockingQueue Semaphore Algorithm

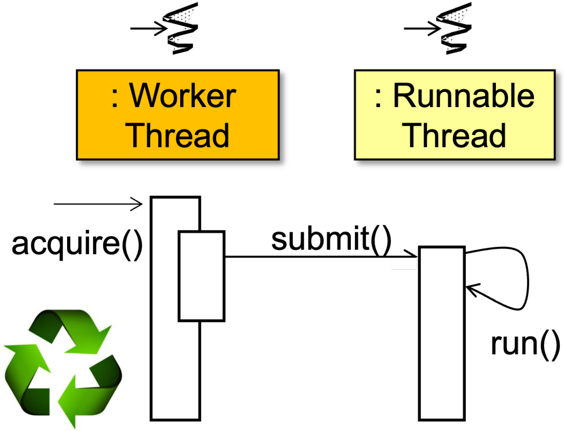
- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
    - Factory returns a Runnable
    - A loop tries acquiring required # of permits by polling queue

```
Runnable getRunnable(int permits,
                     ArrayBlockingQueue<Lock>
                          availLocks,
                     DeferredResult<List<Lock>>
                       result) {
  return () -> {
    var locks = new ArrayList
      <Lock>(permits);
    while (tryAcquire
      (availLocks, locks)
         != permits)
      continue;
    result.setResult(locks);
  }
}
```

Trigger the DeferredResult to return to the locks list to client

The ArrayBlockingQueue Semaphore Algorithm

- **Async Acquire Operation (2)**
  - Create a DeferredResult object
  - A Runnable task is submitted to AsyncTaskExecutor to acquire the specified # of permits
  - Return after submitting the acquisition task



```
DeferredResult<List<Lock>> acquire
   (LockManager lockManager,
    int permits) {
   var result = new DeferredResult
     <List<Lock>>();
   ...

   mExecutor.submit
     (getRunnable(permits,
                  availableLocks,
                  result));
   ...

   return result;
}
```

The HTTP worker thread can be recycled after acquire() returns

The ArrayBlockingQueue Semaphore Algorithm

- **Release Operation (1)**
  - When a lock is released, the release() method tries to put the Lock object back into the ArrayBlockingQueue

```
Boolean release
   (LockManager lockManager) {
  var availableLocks =
    mLockManagerMap
      .get(lockManager);

  if (availableLocks == null)
    return false;
  else
    return availableLocks
                 .offer(lock);
}
```

The ArrayBlockingQueue Semaphore Algorithm

- **Release Operation (1)**
  - When a lock is released, the release() method tries to put the Lock object back into the ArrayBlockingQueue

```
Boolean release
   (LockManager lockManager) {
  var availableLocks =
    mLockManagerMap
      .get(lockManager);

  if (availableLocks == null)
    return false;
  else
    return availableLocks
                .offer(lock);
}
```

Get the ArrayBlockingQueue associated with the LockManager

The ArrayBlockingQueue Semaphore Algorithm

- **Release Operation (1)**
  - When a lock is released, the release() method tries to put the Lock object back into the ArrayBlockingQueue

```
Boolean release
   (LockManager lockManager) {
  var availableLocks =
    mLockManagerMap
      .get(lockManager);

  if (availableLocks == null)
    return false;
  else
    return availableLocks
             .offer(lock);
}
```

This operation is non-blocking & immediately returns a Boolean indicating whether the Lock was successfully returned to the queue

The ArrayBlockingQueue Semaphore Algorithm

- **Release Operation (2)**
  - release() also supports releasing multiple locks at once

```
Boolean release
  (LockManager lockManager,
   List<Lock> locks) {
  var availableLocks =
    mLockManagerMap
      .get(lockManager);

  if (availableLocks == null)
    return false;
  else {
    return locks
      .stream()
      .allMatch
         (availableLocks::offer);
  } ...
```

The ArrayBlockingQueue Semaphore Algorithm

- **Release Operation (2)**
  - release() also supports releasing multiple locks at once

```
Boolean release
  (LockManager lockManager,
   List<Lock> locks) {
  var availableLocks =
    mLockManagerMap
      .get(lockManager);

  if (availableLocks == null)
    return false;
  else {
    return locks
      .stream()
      .allMatch
         (availableLocks::offer);
  } ...
```

Get the ArrayBlockingQueue associated with the LockManager

The ArrayBlockingQueue Semaphore Algorithm

- **Release Operation (2)**
  - release() also supports releasing multiple locks at once

```
Boolean release
   (LockManager lockManager,
    List<Lock> locks) {
  var availableLocks =
    mLockManagerMap
       .get(lockManager);

  if (availableLocks == null)
    return false;
  else {
    return locks
       .stream()
       .allMatch
          (availableLocks::offer);
  } ...
```

Iterate thru the Lock object List, trying to return each one to the queue without blocking

End of the LockManager App Case Study: Server Structure & Functionality (Part 3)