Android Concurrency & Synchronization: Part 4

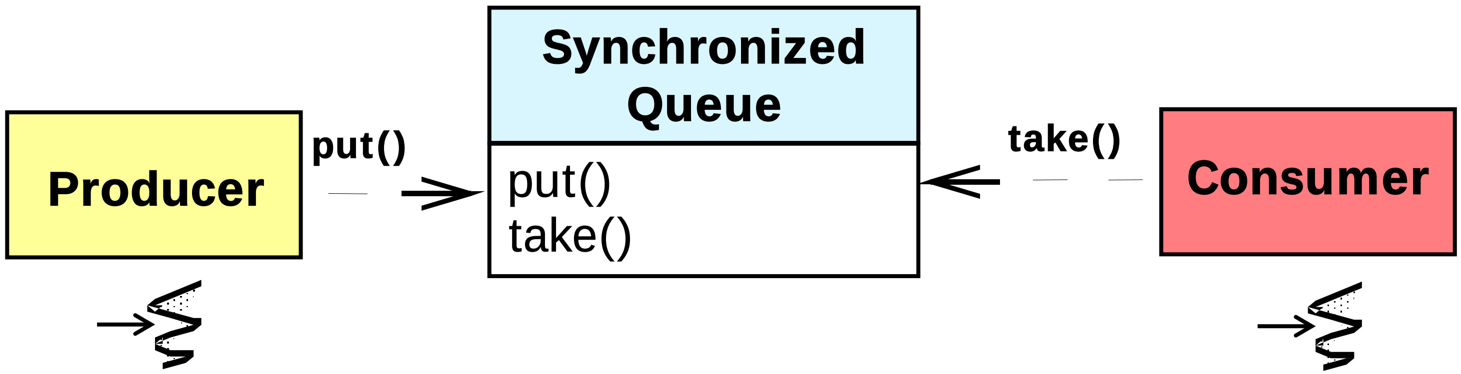
Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

CS 282 Principles of Operating Systems II
Systems Programming for Android

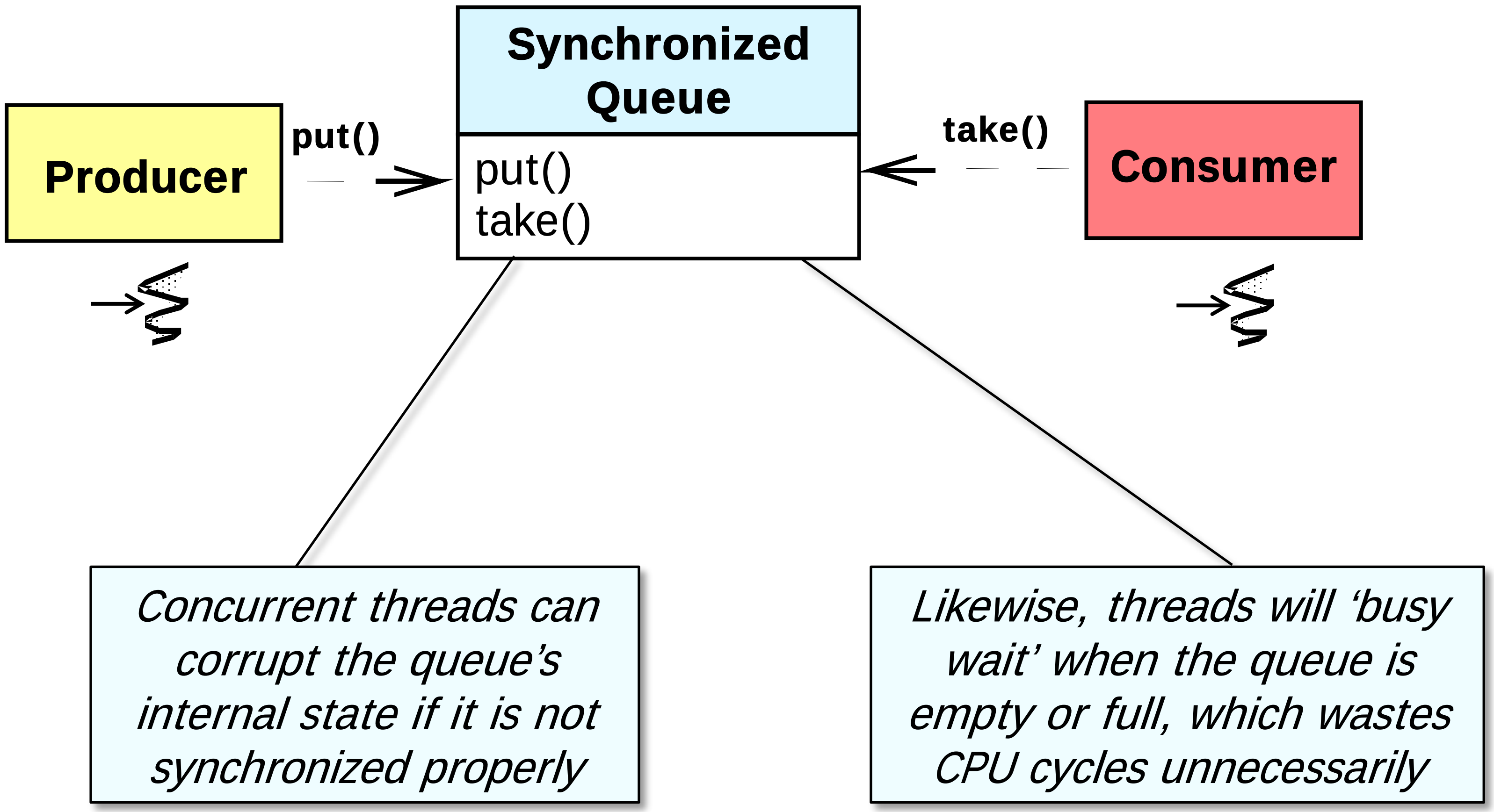
Learning Objectives in this Part of the Module

- Understand the Android mechanisms available to implement concurrent apps that *synchronize* & *schedule* their interactions



Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app



Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here's some example code that demonstrates the problem

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public void put(String msg){ q_.add(msg); }
  public String take(){ return q_.remove(0); }

  public static void main(String argv[]) {
     new Thread(new Runnable(){
        public void run(){
          for(int i = 0; i < 10; i++) put(Integer.toString(i));
        }).start();
     new Thread(new Runnable(){
        public void run(){
          while(true){ System.out.println(take());}
        }).start();
```

Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here’s some example code that demonstrates the problem

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public void put(String msg){ q_.add(msg); }
  public String take(){ return q_.remove(0); }

  public static void main(String argv[]) {
    new Thread(new Runnable(){
       public void run(){
         for(int i = 0; i < 10; i++) put(Integer.toString(i));
       }}).start();
    new Thread(new Runnable(){
       public void run(){
         while(true){ System.out.println(take());}
      }}).start();
```

Resizable-array implementation

Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here's some example code that demonstrates the problem

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public void put(String msg){ q_.add(msg); }
  public String take(){ return q_.remove(0); }

  public static void main(String argv[]) {
    new Thread(new Runnable(){
      public void run(){
        for(int i = 0; i < 10; i++) put(Integer.toString(i));
      }).start();
    new Thread(new Runnable(){
      public void run(){
        while(true){ System.out.println(take());}
      }).start();
```

Enqueue & dequeue strings into/from the queue

Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here's some example code that demonstrates the problem

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public void put(String msg){ q_.add(msg); }
  public String take(){ return q_.remove(0); }

  public static void main(String argv[]) {
    new Thread(new Runnable(){
      public void run(){
        for(int i = 0; i < 10; i++) put(Integer.toString(i));
      }}).start();
    new Thread(new Runnable(){
      public void run(){
        while(true){ System.out.println(take());}
      }}).start();
```

Spawn producer & consumer threads

Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here's some example code that demonstrates the problem

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public void put(String msg){ q_.add(msg); }
  public String take(){ return q_.remove(0); }

  public static void main(String argv[]) {
     new Thread(new Runnable(){
        public void run(){
          for(int i = 0; i < 10; i++) put(Integer.toString(i));
        }}).start();
     new Thread(new Runnable(){
        public void run(){
          while(true){ System.out.println(take());}
        }}).start();
```

What output will this code produce?

Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here's some example code that demonstrates the problem

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public void put(String msg){ q_.add(msg); }
  public String take(){ return q_.remove(0); }

  public static void main(String argv[]) {
     new Thread(new Runnable(){
        public void run(){
          for(int i = 0; i < 10; i++) put(Integer.toString(i));
        }}).start();
     new Thread(new Runnable(){
        public void run(){
          while(true){ System.out.println(take());}
        }}).start();
```

Must protect critical sections from being run by two threads concurrently

Partial Solution Using Java Synchronization

- Java provides the “synchronized” keyword to specify sections of code in an object that cannot be accessed concurrently by two threads

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public void synchronized put(String msg){ q_.add(msg); }
  public String synchronized take(){ return q_.remove(0); }

  public static void main(String argv[]) {
    new Thread(new Runnable(){
      public void run(){
        for(int i = 0; i < 10; i++) put(Integer.toString(i));
      }).start();
    new Thread(new Runnable(){
      public void run(){
        while(true){ System.out.println(take());}
      }).start();
```

Only one synchronized method can be active in any given object

There are still problems with this solution...

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
  - Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

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

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
  - Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword
  - Access to a synchronized method is serialized w/other synchronized methods

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
  - e.g.,

```
void put(String msg)
{
  ...
  synchronized (this) {
     q_.add(msg);
     notifyAll();
  }
```

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
  - e.g.,

```
void put(String msg)
{
  ...
  synchronized (this) {
     q_.add(msg);
     notifyAll();
  }
```

  - Synchronized blocks enable more fine-grained serialization

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html has more

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
- Java objects have wait() & notify()/notifyAll() methods that allow callers to wait for a condition to become true

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
- Java objects have wait() & notify()/notifyAll() methods that allow callers to wait for a condition to become true
  - Calling wait() on an object will suspend current thread until a notify*() call is made on the same object

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
- Java objects have wait() & notify()/notifyAll() methods that allow callers to wait for a condition to become true
  - Calling wait() on an object will suspend current thread until a notify*() call is made on the same object
  - Calling notifyAll() will wake up all waiting threads

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again

Detailed Analysis of wait() & notifyAll()

- Inside a synchronized method, you can request a thread "wait" for a condition, e.g.:
  - The synchronized take() method acquires the monitor lock, checks the queue size, & waits if the queue is empty

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

Detailed Analysis of wait() & notifyAll()

- Inside a synchronized method, you can request a thread "wait" for a condition, e.g.:
  - The synchronized take() method acquires the monitor lock, checks the queue size, & waits if the queue is empty

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

- *Always invoke wait() inside a loop that tests for the condition being waited for*
- *Don't assume the notification was for the particular condition being waited for or that the condition is still true*

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

Detailed Analysis of wait() & notifyAll()

- Inside a synchronized method, you can request a thread "wait" for a condition, e.g.:
  - The synchronized take() method acquires the monitor lock, checks the queue size, & waits if the queue is empty
  - The thread blocking on wait() doesn't continue until another thread notifies it that the queue has data to process

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

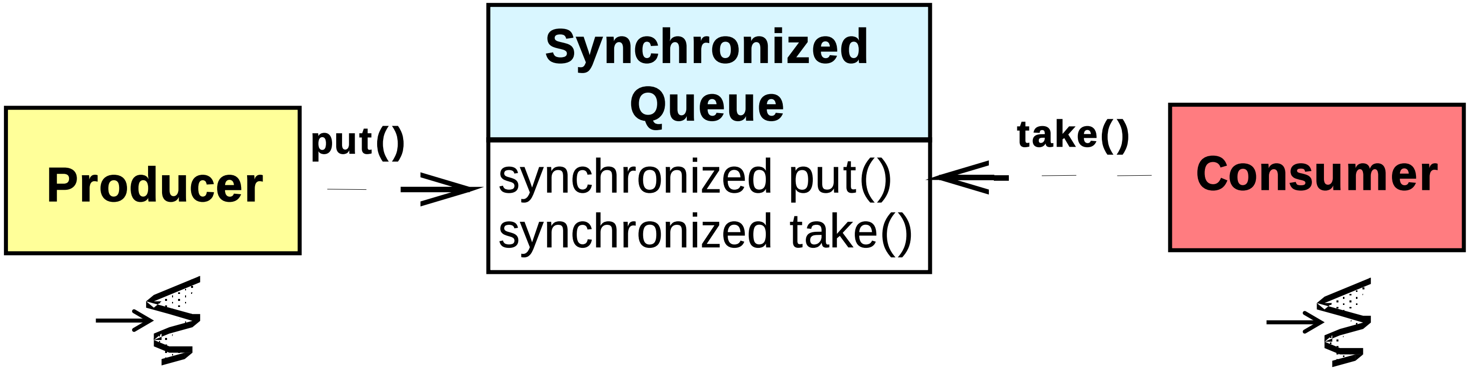
Detailed Analysis of wait() & notifyAll()

- Inside a synchronized method, you can request a thread "wait" for a condition, e.g.:
  - The synchronized take() method acquires the monitor lock, checks the queue size, & waits if the queue is empty
  - The thread blocking on wait() doesn't continue until another thread notifies it that the queue has data to process
  - When the thread is notified, it wakes up, obtains the monitor lock, continues after the wait() call, & releases the lock when the method returns

```
public class SynchronizedQueue {
  private List<String> q_ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
```

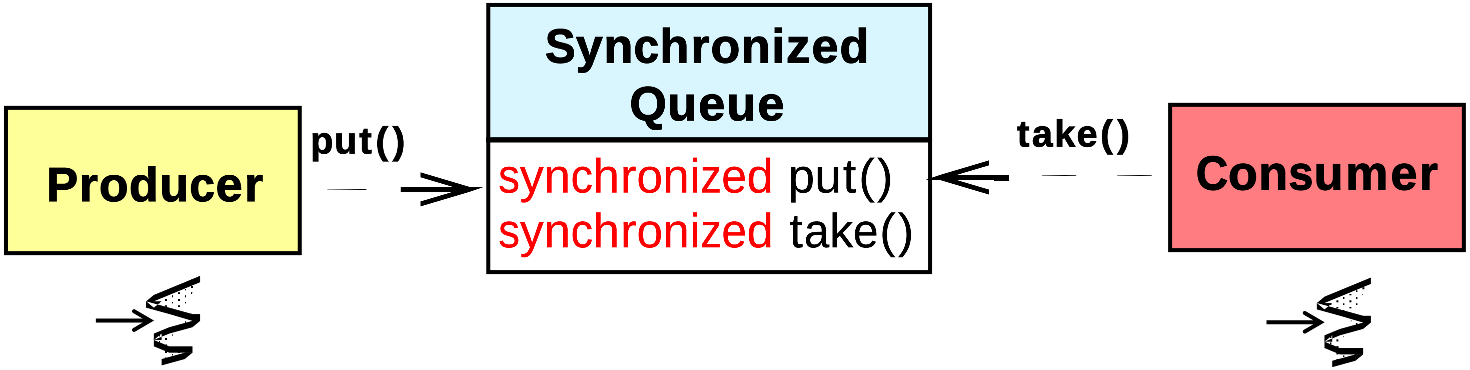
Summary

- Each Java object may be used as a monitor object



Summary

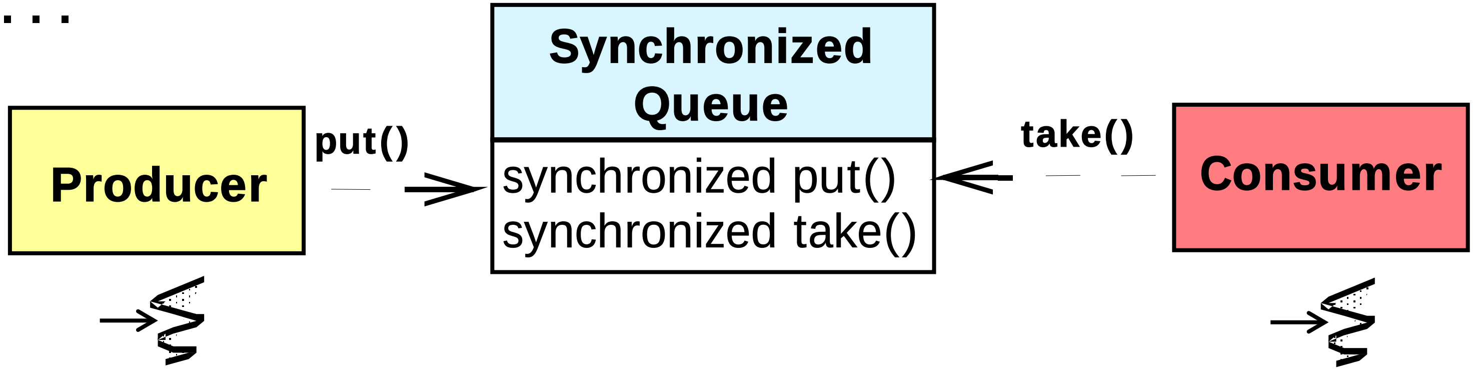
- Each Java object may be used as a monitor object
  - Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword



Summary

- Each Java object may be used as a monitor object
  - Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword
  - Blocks of code may also be marked by synchronized

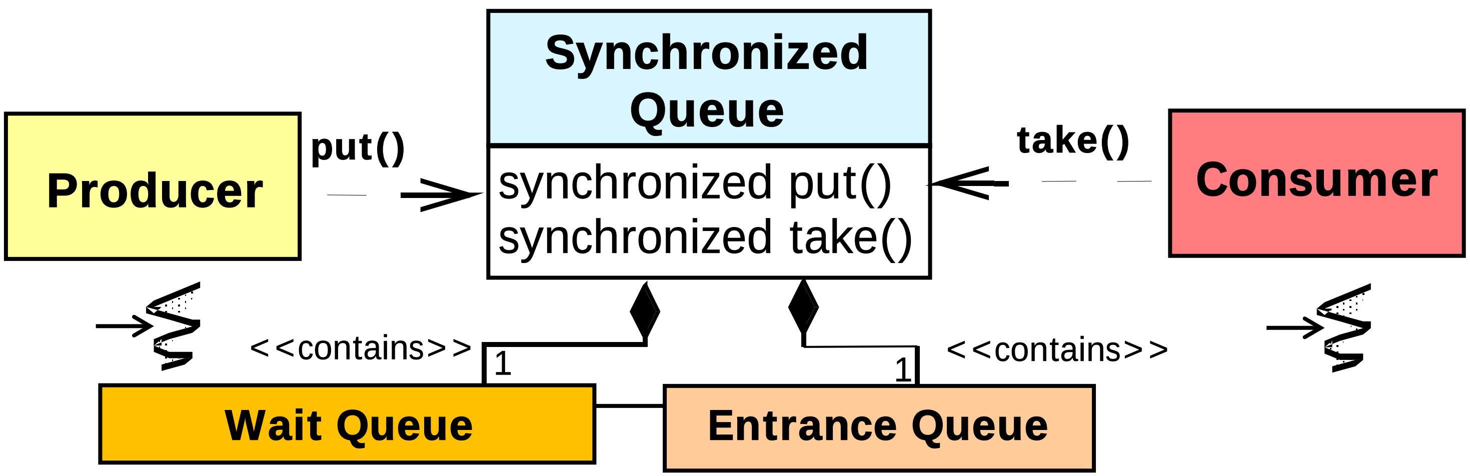
```
void put(String msg) {
   synchronized (this) {
     q_.add(msg);
     notifyAll();
   }
   ...
```



docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html has more

Summary

- Each Java object may be used as a monitor object
- Each monitor object in Java is equipped with a single wait queue in addition to its entrance queue
  - All waiting is done on this single wait queue & all notify() & notifyAll() operations apply to this queue



Summary

- Production Java apps may need more than the simply monitor mechanisms

Interfaces

Condition	`Condition` factors out the `Object` monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock` implementations.
Lock	`Lock` implementations provide more extensive locking operations than can be obtained using `synchronized` methods and statements.
ReadWriteLock	A `ReadWriteLock` maintains a pair of associated `locks`, one for read-only operations and one for writing.

Classes

AbstractOwnableSynchronizer	A synchronizer that may be exclusively owned by a thread.
AbstractQueuedLongSynchronizer	A version of `AbstractQueuedSynchronizer` in which synchronization state is maintained as a `long`.
AbstractQueuedLongSynchronizer.ConditionObject	Condition implementation for a `AbstractQueuedLongSynchronizer` serving as the basis of a `Lock` implementation.
AbstractQueuedSynchronizer	Provides a framework for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on first-in-first-out (FIFO) wait queues.
AbstractQueuedSynchronizer.ConditionObject	Condition implementation for a `AbstractQueuedSynchronizer` serving as the basis of a `Lock` implementation.
LockSupport	Basic thread blocking primitives for creating locks and other synchronization classes.
ReentrantLock	A reentrant mutual exclusion `Lock` with the same basic behavior and semantics as the implicit monitor lock accessed using `synchronized` methods and statements, but with extended capabilities.
ReentrantReadWriteLock	An implementation of `ReadWriteLock` supporting similar semantics to `ReentrantLock`.
ReentrantReadWriteLock.ReadLock	The lock returned by method `readLock()`.
ReentrantReadWriteLock.WriteLock	The lock returned by method `writeLock()`.

developer.android.com/reference/java/util/concurrent/locks/package-summary.html

Android Concurrency & Synchronization: Part 5

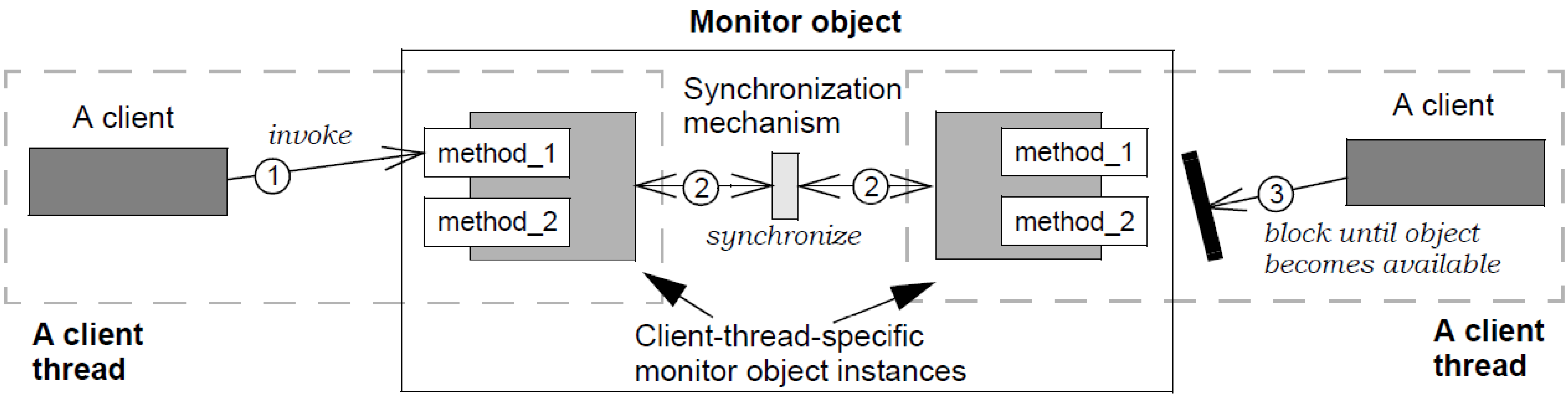
Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

CS 282 Principles of Operating Systems II
Systems Programming for Android

Learning Objectives in this Part of the Module

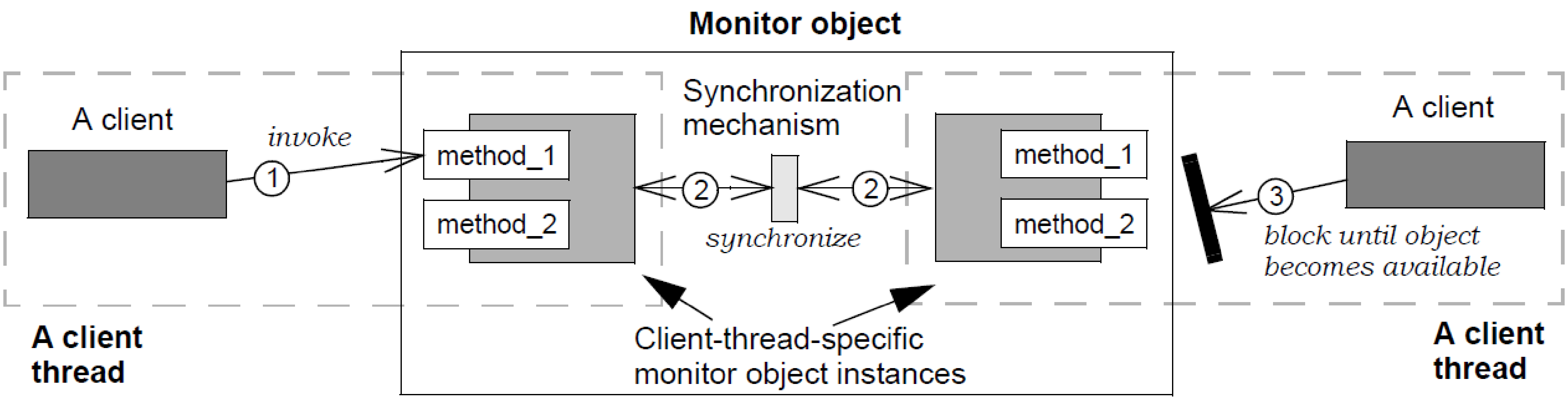
- Understand the *Monitor Object* pattern & how it can be used to synchronize & schedule concurrent Android programs



Monitor Object POSA2 Concurrency

Intent

- Synchronizes concurrent method execution to ensure only one method at a time runs within an object
- Allows an object's methods to cooperatively schedule their execution sequences

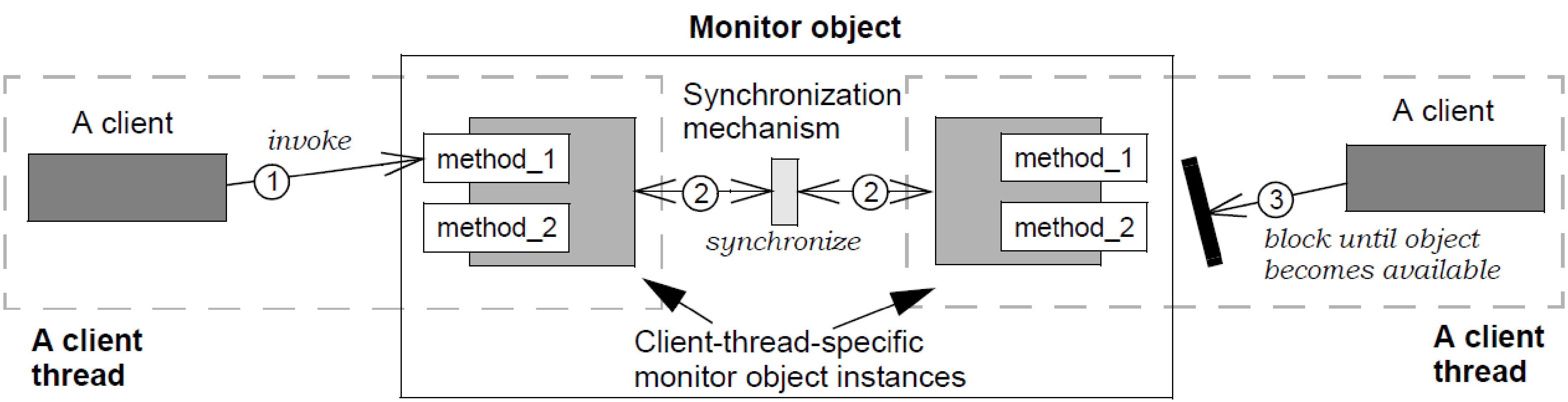


See www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf for *Monitor Object*

Monitor Object POSA2 Concurrency

Applicability

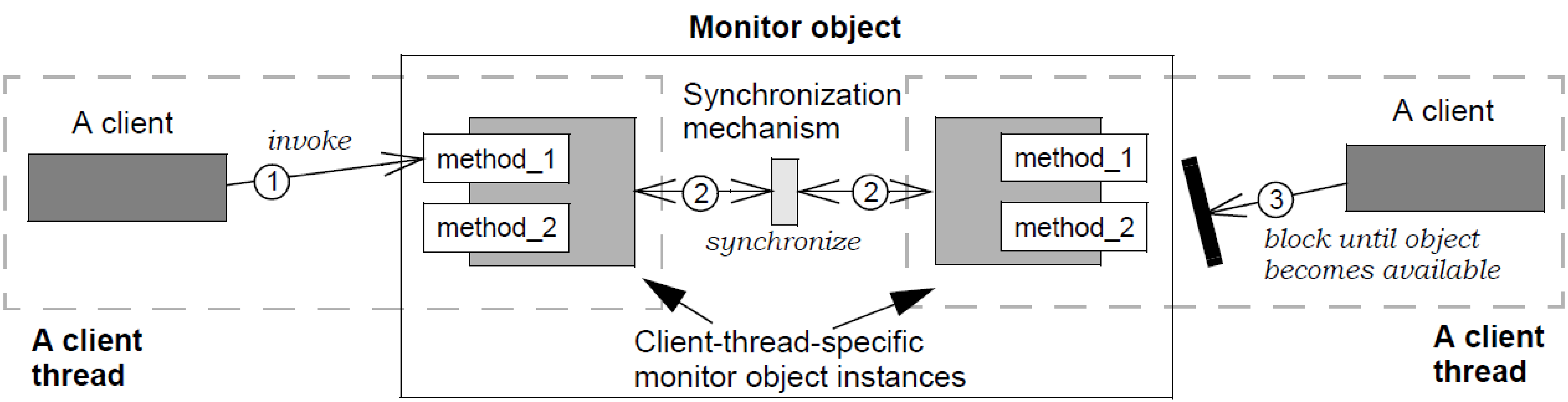
- When an object's interface methods should define its synchronization boundaries



Monitor Object POSA2 Concurrency

Applicability

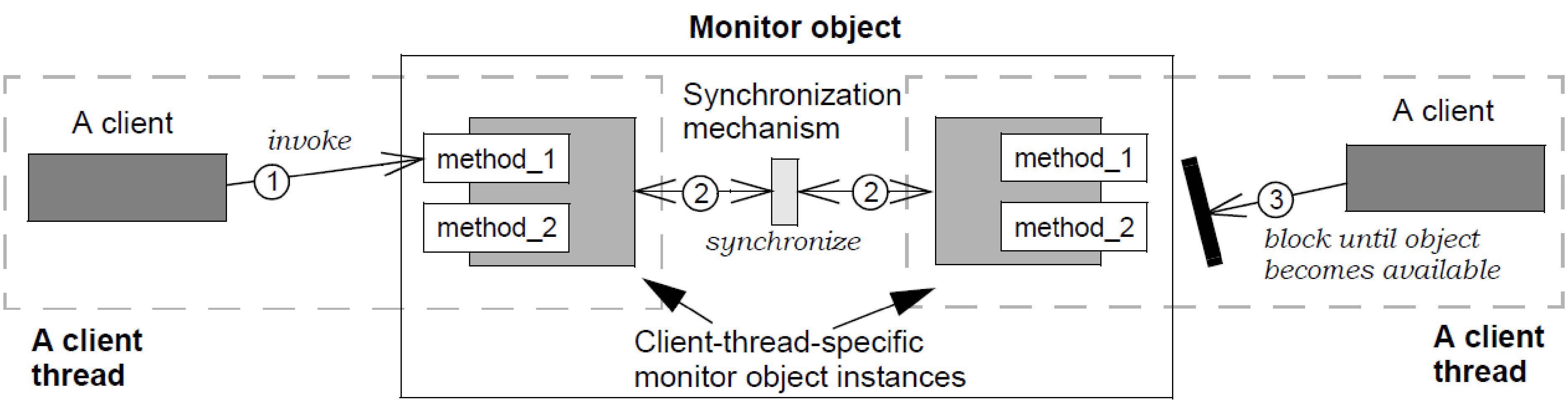
- When an object's interface methods should define its synchronization boundaries
- When only one method at a time should be active within the same object



Monitor Object POSA2 Concurrency

Applicability

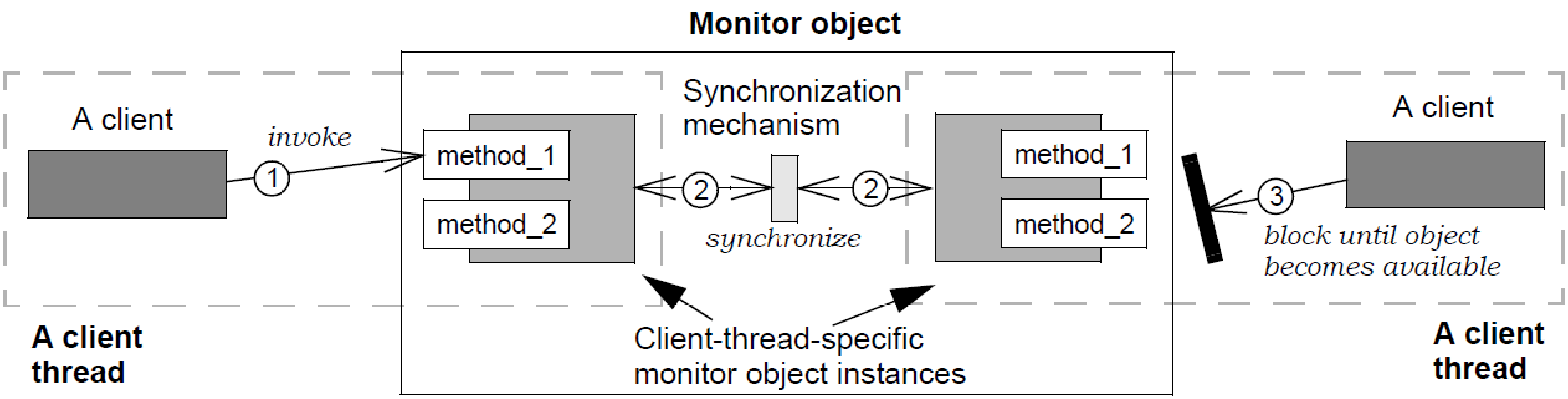
- When an object's interface methods should define its synchronization boundaries
- When only one method at a time should be active within the same object
- When objects should be responsible for method synchronization transparently, without requiring explicit client intervention



Monitor Object POSA2 Concurrency

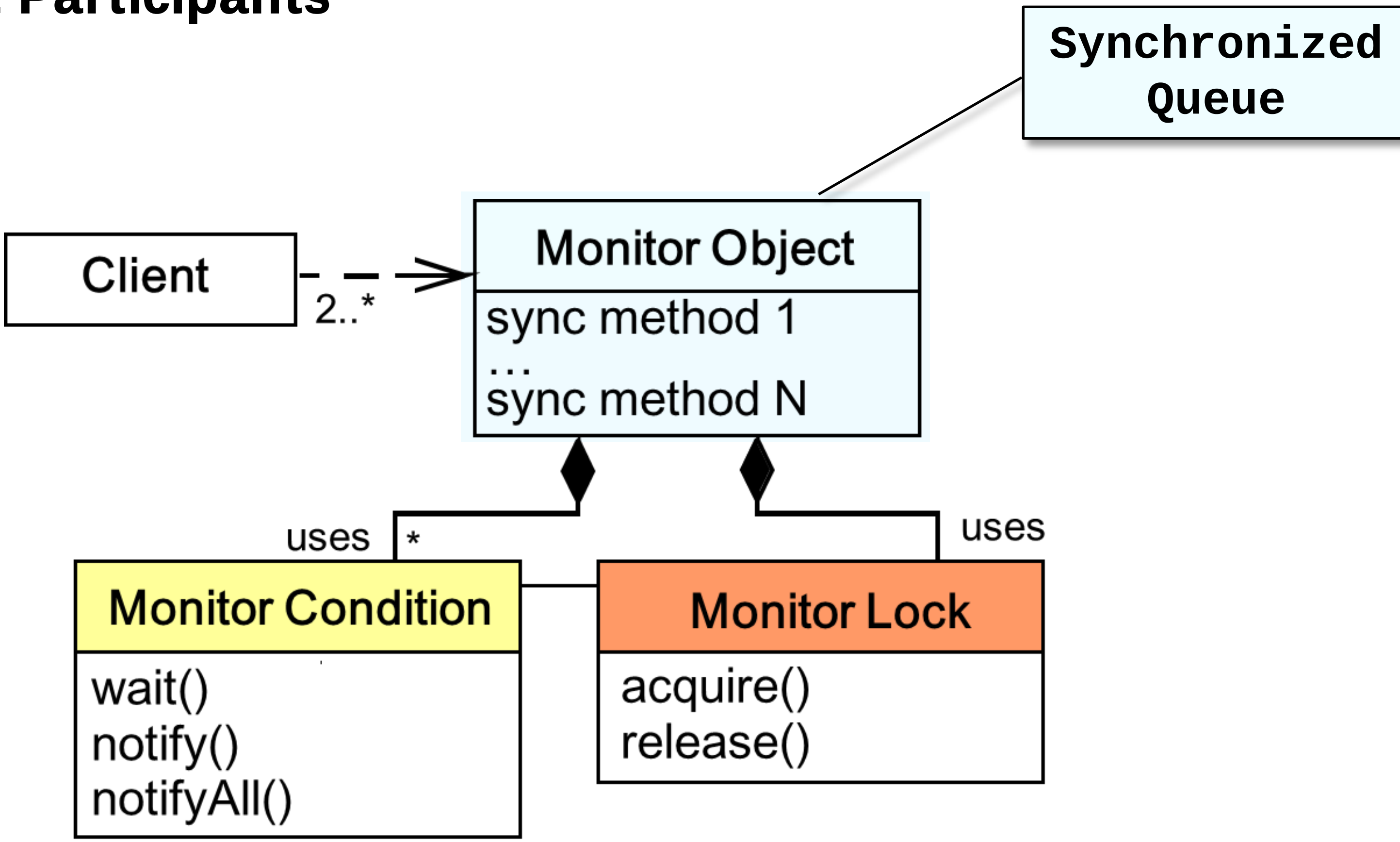
Applicability

- When an object's interface methods should define its synchronization boundaries
- When only one method at a time should be active within the same object
- When objects should be responsible for method synchronization transparently, without requiring explicit client intervention
- When an object's methods may block during their execution



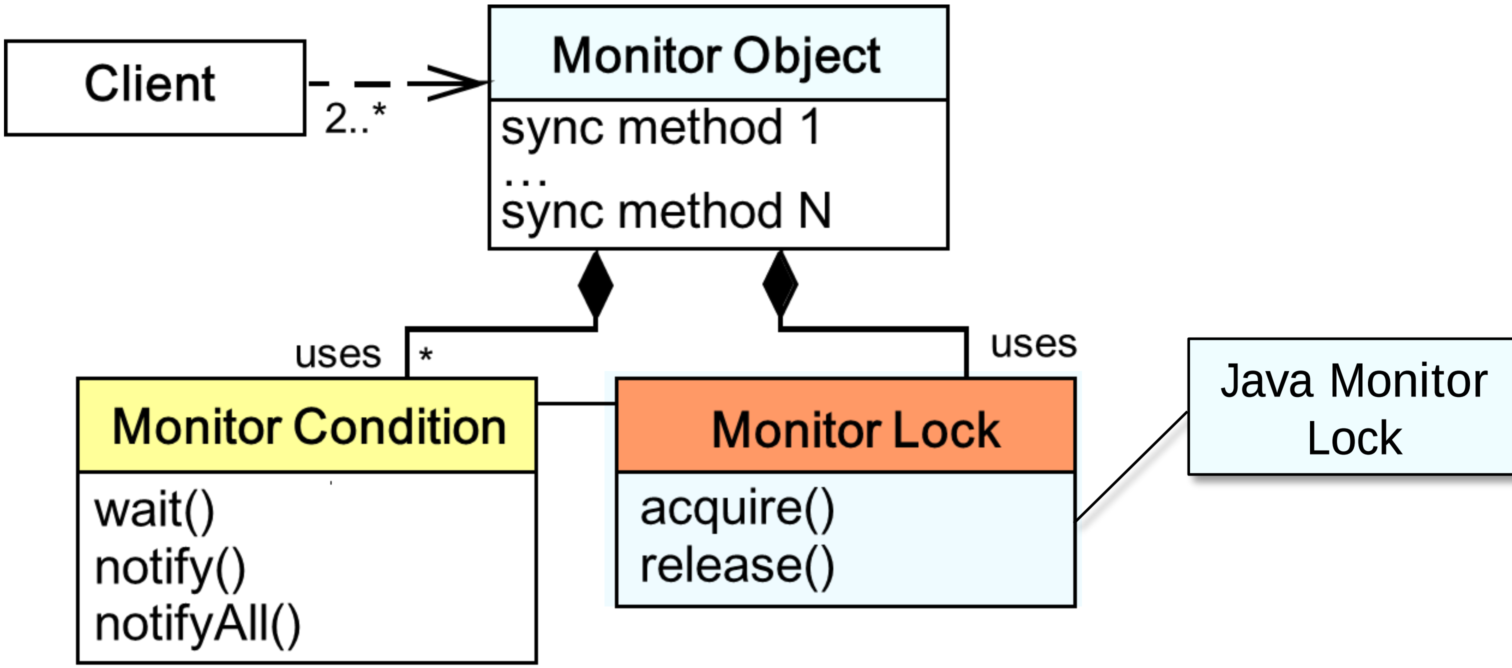
Monitor Object POSA2 Concurrency

Structure & Participants



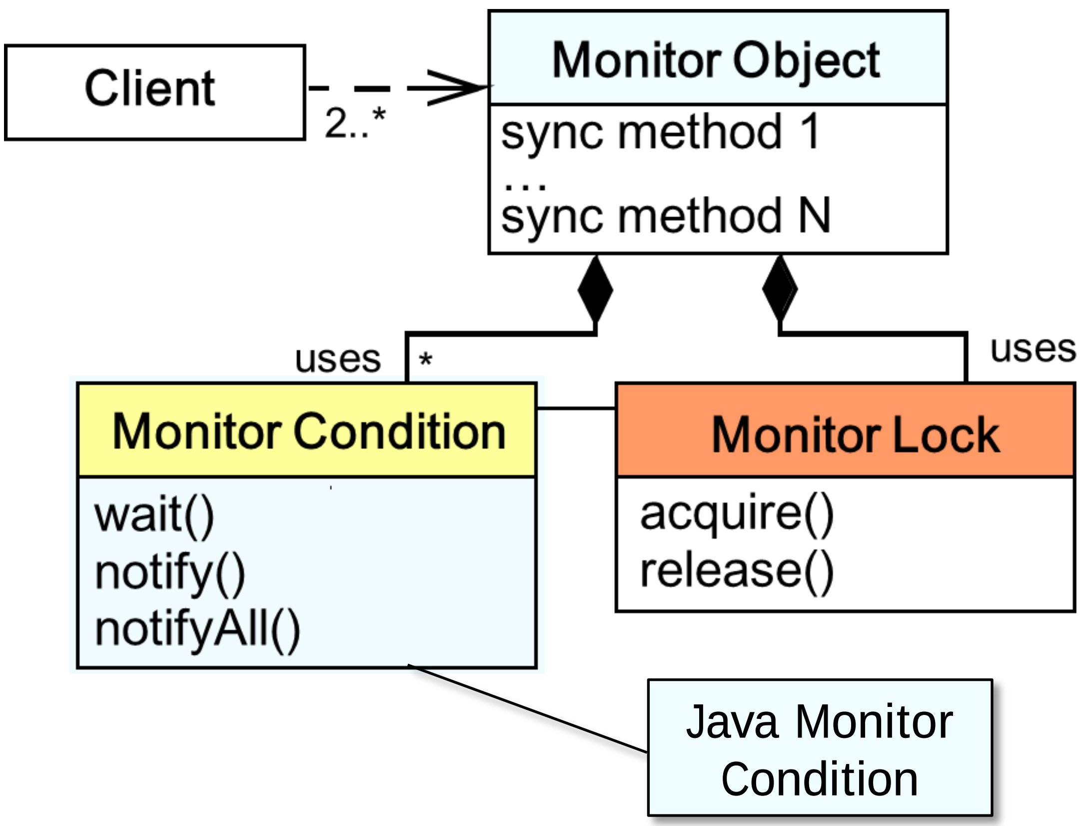
Monitor Object POSA2 Concurrency

Structure & Participants



Monitor Object POSA2 Concurrency

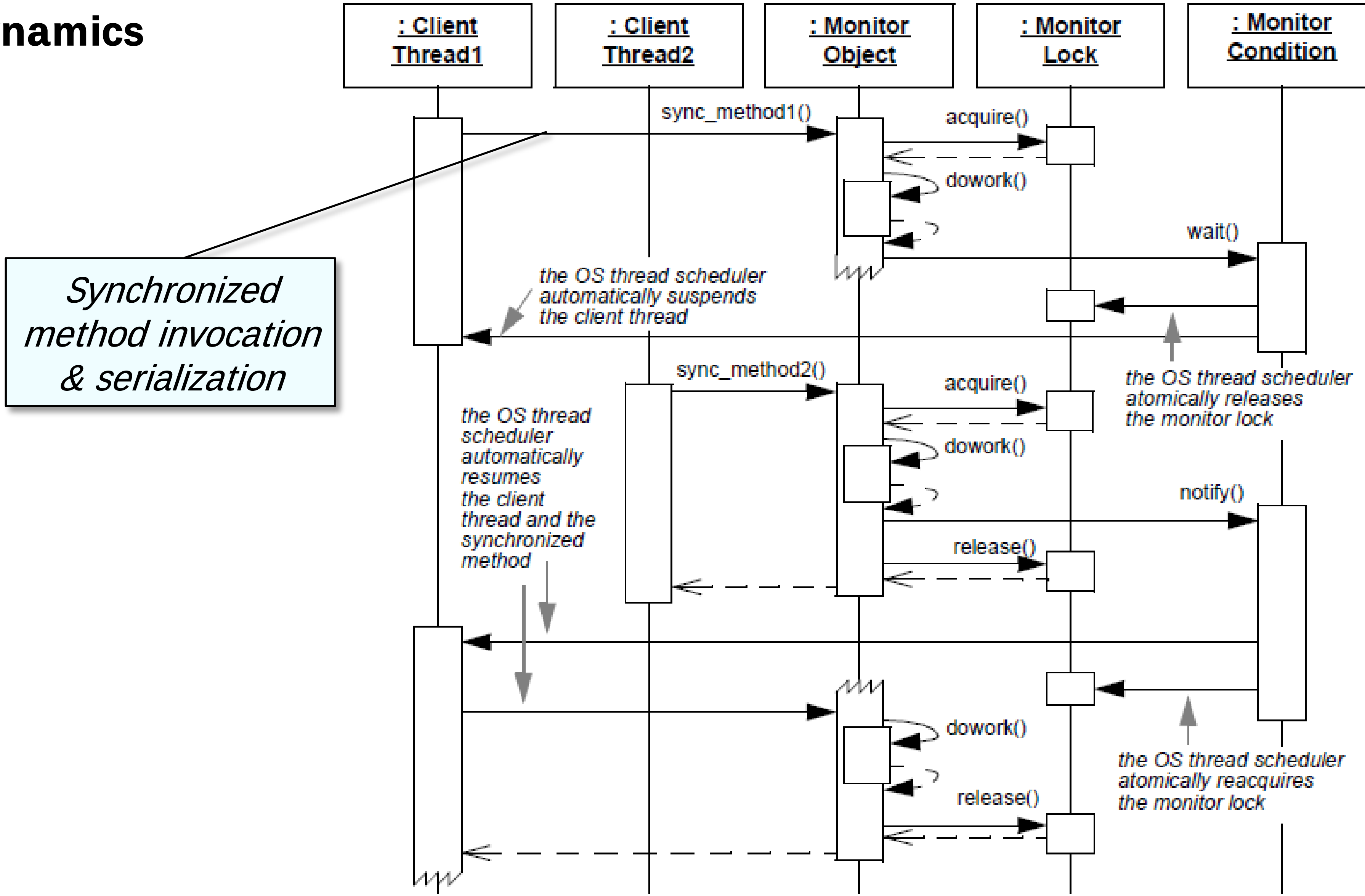
Structure & Participants



Note that Java monitor objects only have a single (implicit) monitor condition

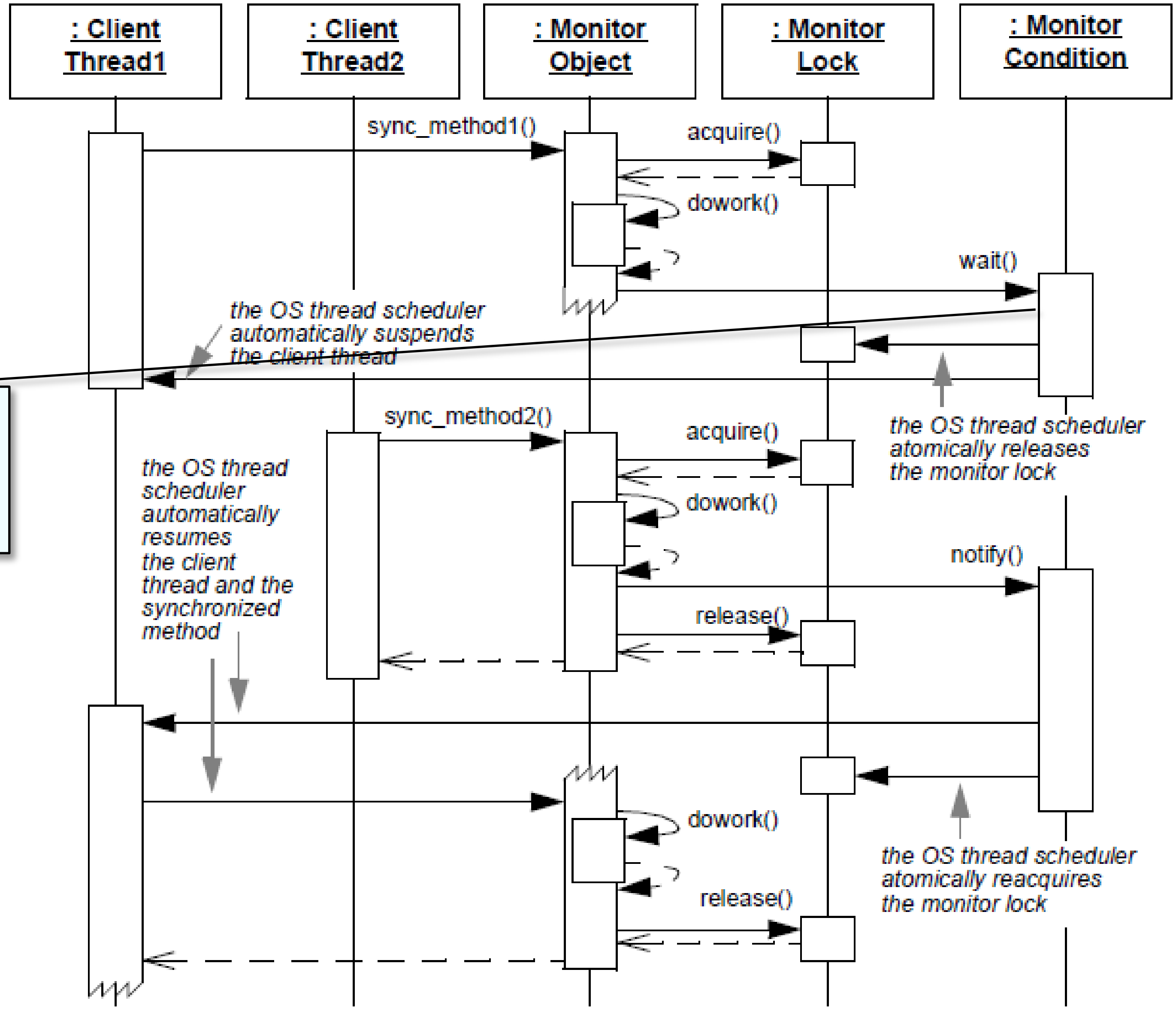
Monitor Object POSA2 Concurrency

Dynamics



Monitor Object POSA2 Concurrency

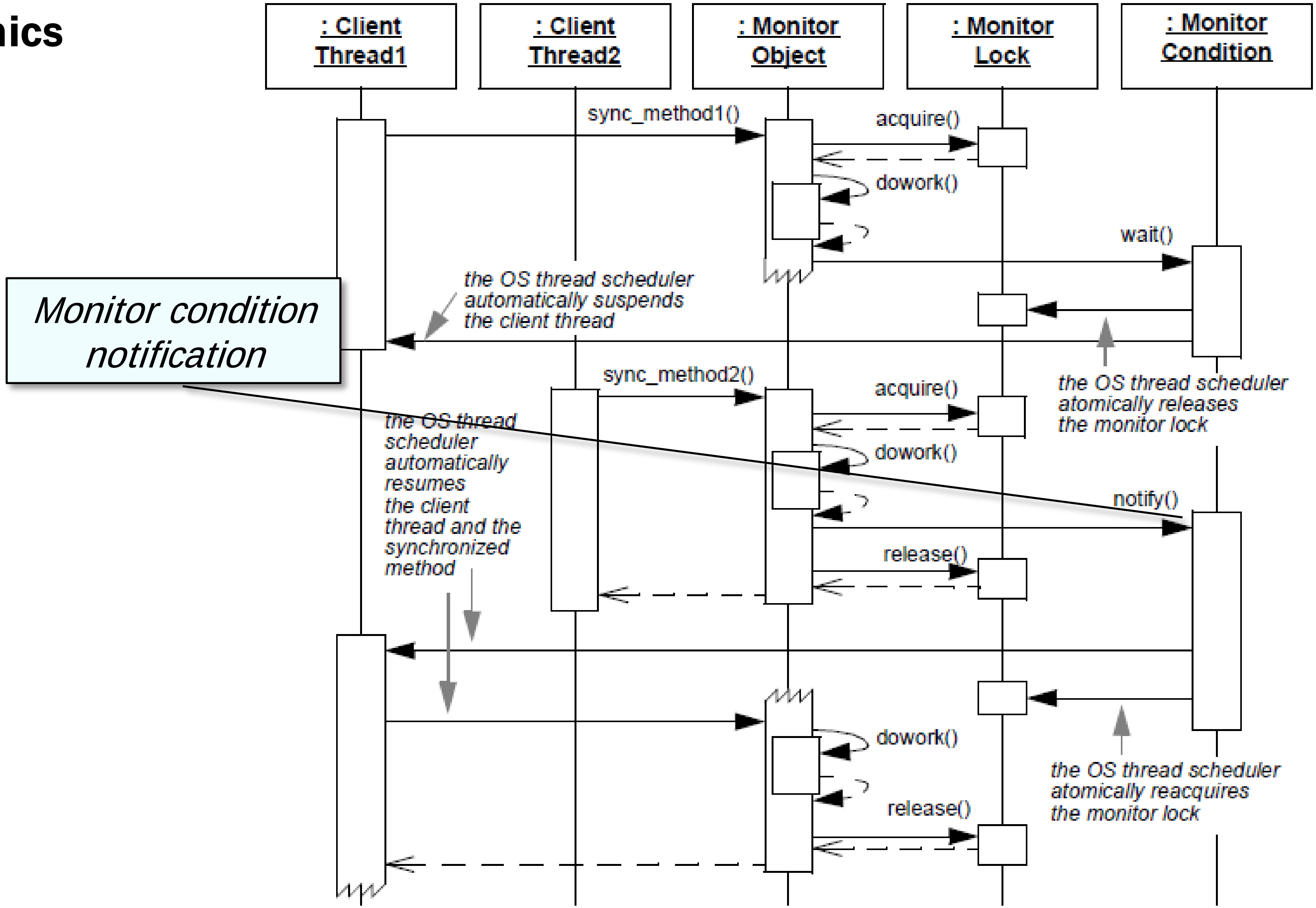
Dynamics



Synchronized method thread suspension

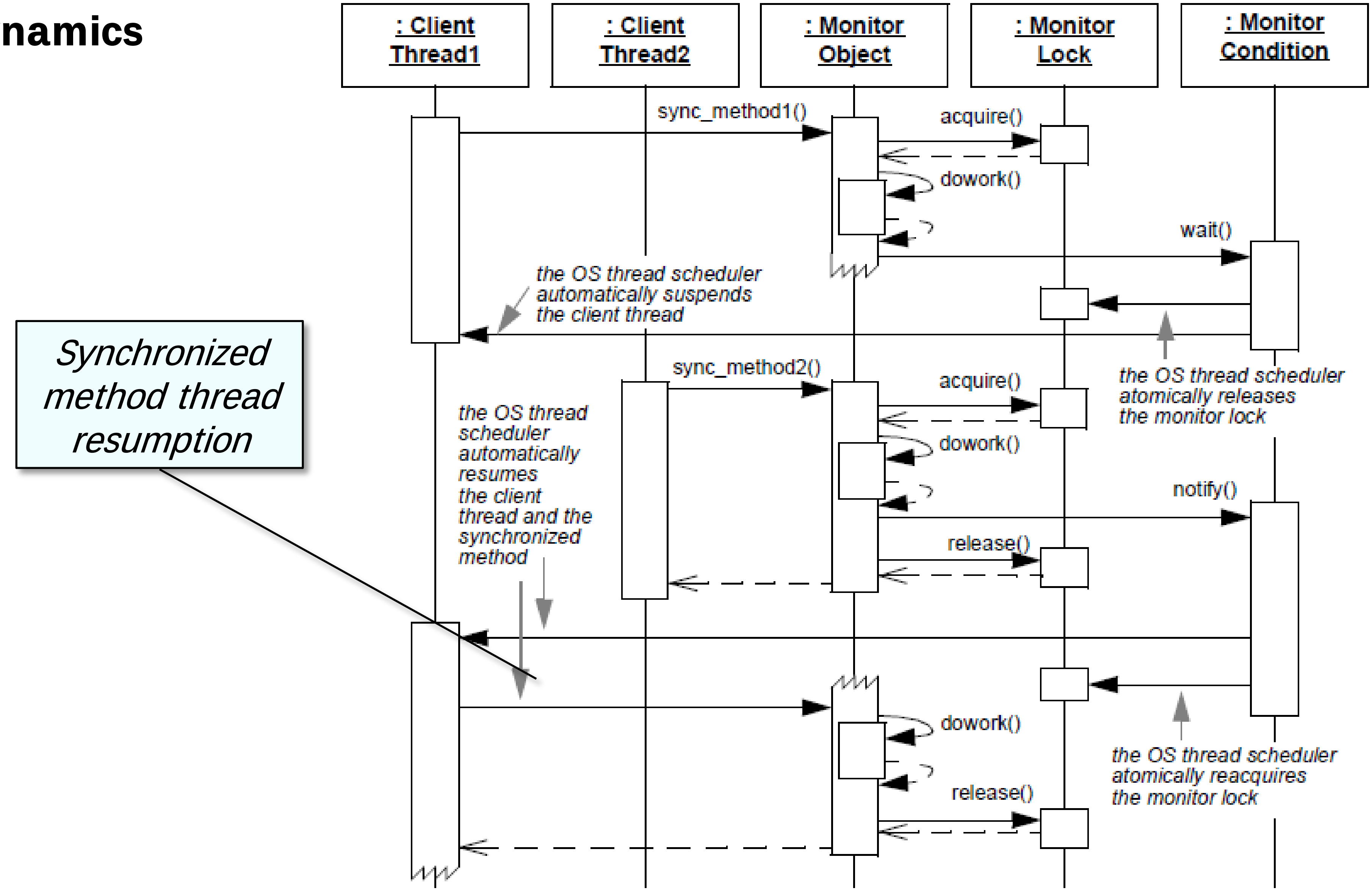
Monitor Object POSA2 Concurrency

Dynamics



Monitor Object POSA2 Concurrency

Dynamics



Monitor Object POSA2 Concurrency

Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that's in progress

```
public final class CancellationSignal
{
    ...
    private boolean
            mCancelInProgress;

    public void setOnCancelListener
      (OnCancelListener listener) {
      ...
    }

    public void cancel() {
      ...
    }
    ...
}
```

See developer.android.com/reference/android/os/CancellationSignal.html

Monitor Object POSA2 Concurrency

Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that's in progress
  - Used for long-running operations like ContentResolver.query()

```
public final class CancellationSignal
{
    ...
    private boolean
            mCancelInProgress;

    public void setOnCancelListener
      (OnCancelListener listener) {
      ...
    }

    public void cancel() {
      ...
    }
    ...
```

See developer.android.com/reference/android/content/ContentResolver.html #query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String, android.os.CancellationSignal)

Monitor Object POSA2 Concurrency

Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that's in progress
- Several method are used to implement *Monitor Object*
  - setOnCancelListener() – Sets the cancellation listener whose onCancel() hook will be called when an operation is cancelled

```
public final class CancellationSignal
{
    ...
    private boolean
            mCancelInProgress;

    public void setOnCancelListener
      (OnCancelListener listener) {
      synchronized (this) {
        while (mCancelInProgress) {
          try { wait(); } catch
          (InterruptedException ex)
          {}
        }
        ...
        mOnCancelListener = listener;
        ...
```

frameworks/base/core/java/android/os/CancellationSignal.java has the code

Monitor Object POSA2 Concurrency

Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that's in progress
- Several method are used to implement *Monitor Object*
  - setOnCancelListener() – Sets the cancellation listener whose onCancel() hook will be called when an operation is cancelled
  - cancel() – Cancels operation & signals cancellation listener

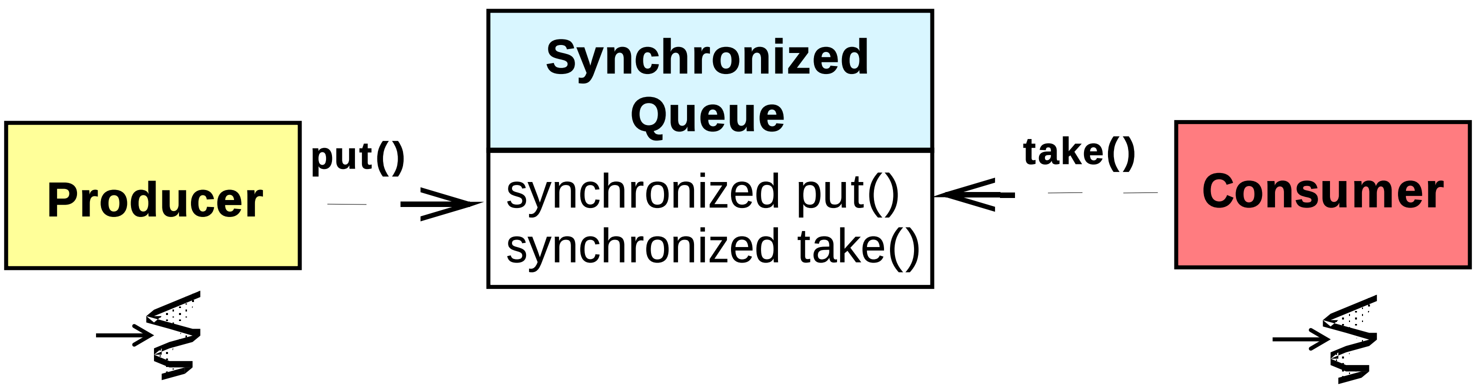
```
public final class CancellationSignal
{
    ...
    public void cancel() {
      synchronized (this) {
        mCancelInProgress = true;
        ...
      }
      try {
        ...
        listener.onCancel();
        ...
      } finally {
        synchronized (this) {
          mCancelInProgress = false;
          notifyAll();
        }
```

frameworks/base/core/java/android/os/CancellationSignal.java has the code

Monitor Object POSA2 Concurrency

Consequences

+ Simplification of concurrency control
  - Presents a concise programming model for sharing an object among cooperating threads where object synchronization corresponds to method invocations



Monitor Object POSA2 Concurrency

Consequences

+ Simplification of concurrency control

+ Simplification of scheduling method execution

- Synchronized methods use their monitor conditions to determine the circumstances under which they should suspend or resume their execution & those of collaborating monitor objects

```
public synchronized
  void put(String msg){
  ...
  q_.add(msg);
  notifyAll();
}
```

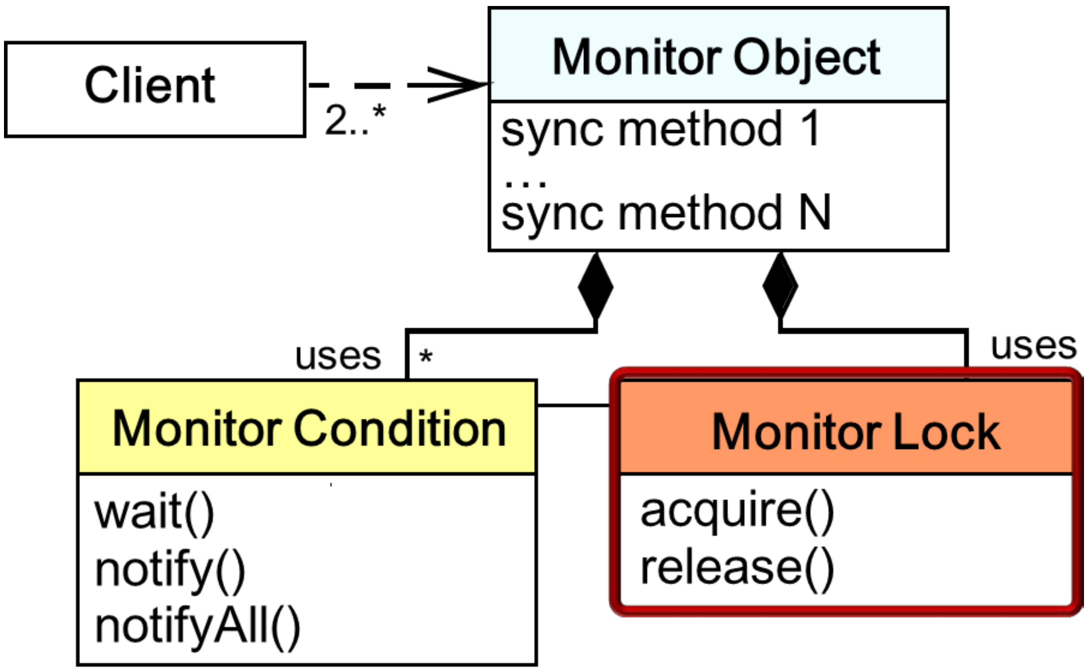
```
public synchronized
  String take(){
  while (q_.isEmpty()) {
    wait();
  }
  ...
  return q_.remove(0);
  }
}
```

Monitor Object POSA2 Concurrency

Consequences

– Limited Scalability

- A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object



Monitor Object POSA2 Concurrency

Consequences

– Limited Scalability

– Complicated extensibility semantics

- These result from the coupling between a monitor object's functionality & its synchronization mechanisms

```
public synchronized
  void put(String msg){
  ...
  q_.add(msg);
  notifyAll();
}
```

```
public synchronized
  String take(){
  while (q_.isEmpty()) {
    wait();
  }
  ...
  return q_.remove(0);
}
}
```

wisnesky.net/anomaly.pdf has info on the "inheritance anomaly problem"

Monitor Object POSA2 Concurrency

Consequences

– Limited Scalability

– Complicated extensibility semantics

– Nested monitor lockout

- This problem can occur when monitor objects are nested

```
class Inner {
  protected boolean cond_ = false;
  public synchronized void
     awaitCondition() {
   while (!cond)
   try { wait (); } catch
   (InterruptedException e) {}
  }
  public synchronized void
    signalCondition(boolean c) {
      cond_ = c; notifyAll ();
  }
}
```

```
class Outer {
  protected Inner inner_ =
    new Inner();
  public synchronized void
    process() {
    inner_.awaitCondition ();
  }
  public synchronized void
    set(boolean c) {
    inner_.signalCondition(c);
  }
}
```

Holds the monitor lock

Method won't execute!

Monitor Object POSA2 Concurrency

Implementation

- Define the monitor object's interface methods
  - e.g., ArrayBlockingQueue is a bounded BlockingQueue backed by an array that queues elements in FIFO order

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
        implements BlockingQueue<E>,
        java.io.Serializable {

  public void put(E e) ...

  public E take() ...
}
```

Monitor Object POSA2 Concurrency

Implementation

- Define the monitor object's interface methods
- Define the monitor object's implementation methods
  - See the *Thread-Safe Interface* pattern for design rationale

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
        implements BlockingQueue<E>,
        java.io.Serializable {

  public void put(E e) ...

  public E take() ...

  private void insert(E x) ...

  private E extract() ...
}
```

Monitor Object POSA2 Concurrency

Implementation

- Define the monitor object's interface methods
- Define the monitor object's implementation methods
- Define the monitor object's internal state & synchronization mechanisms
  - Can use classes defined in the java.util.concurrent package

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
        implements BlockingQueue<E>,
        java.io.Serializable {

  ...

  final Object[] items;
  int takeIndex;
  int putIndex;
  int count;

  final ReentrantLock lock;
  private final Condition notEmpty;
  private final Condition notFull;
}
```

See Lock.html & Condition.html at developer.android.com/reference/java/util/concurrent/locks

Monitor Object POSA2 Concurrency

Implementation

- Define the monitor object's interface methods
- Define the monitor object's implementation methods
- Define the monitor object's internal state & synchronization mechanisms
- Implement all the monitor object's methods & data members
  - Note the Java synchronized keyword isn't used here!

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
        implements BlockingQueue<E>,
        java.io.Serializable {

  public void put(E e) throws
               InterruptedException {
    final ReentrantLock lock =
                           this.lock;
    lock.lockInterruptibly();
    try {
      while (count == items.length)
         notFull.await();
      insert(e);
    } finally {
       lock.unlock();
  }
  ...
```

libcore/luni/src/main/java/java/util/concurrent/ArrayBlockingQueue.java

Monitor Object POSA2 Concurrency

Implementation

- Define the monitor object's interface methods
- Define the monitor object's implementation methods
- Define the monitor object's internal state & synchronization mechanisms
- Implement all the monitor object's methods & data members
  - Note the Java synchronized keyword isn't used here!

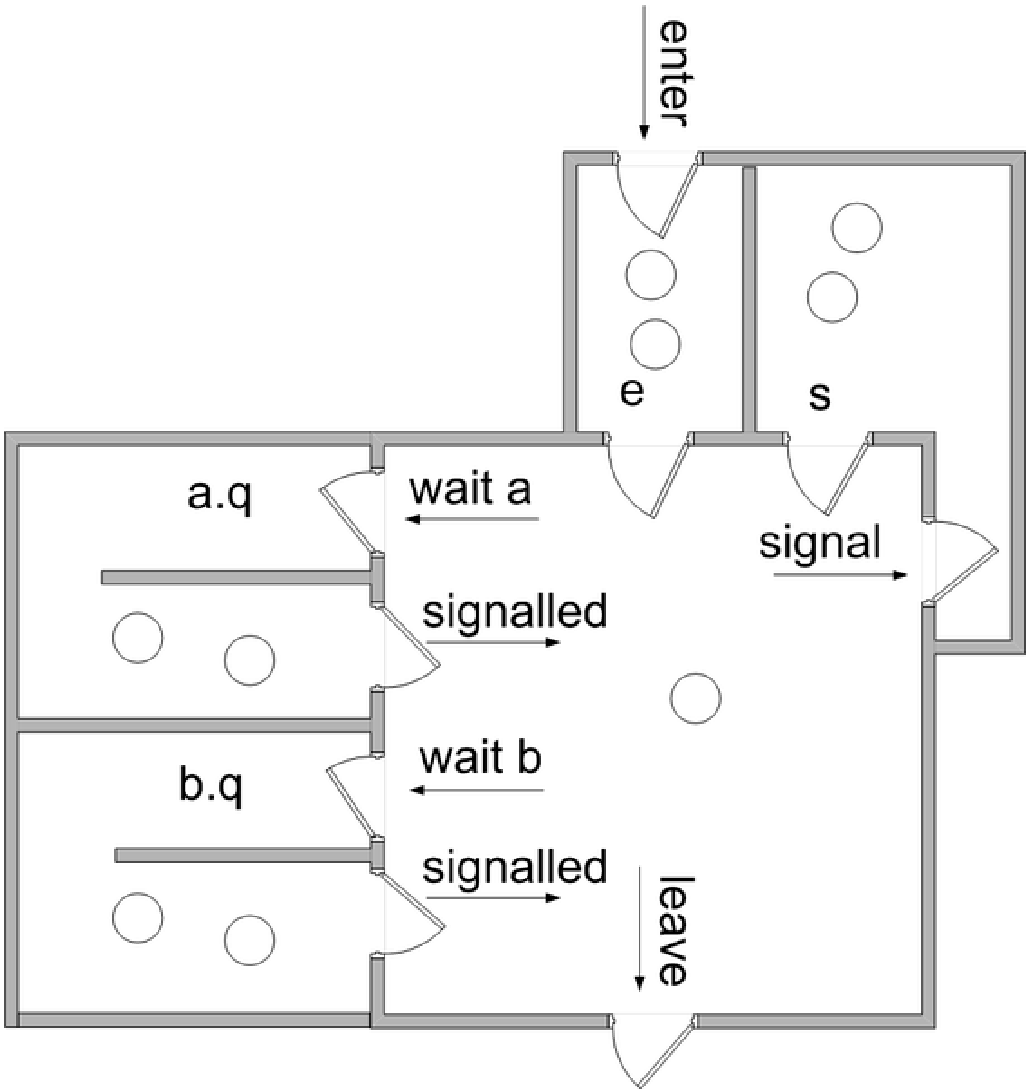
```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
        implements BlockingQueue<E>,
        java.io.Serializable {

  public E take() throws
                 InterruptedException {
    final ReentrantLock lock =
                            this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
       lock.unlock();
  }
  ...
```

libcore/luni/src/main/java/java/util/concurrent/ArrayBlockingQueue.java

Monitor Object POSA2 Concurrency

Known Uses

- Dijkstra & Hoare-style Monitors



Monitor Object POSA2 Concurrency

Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
  - Note how few synchronized methods/blocks are used in java.util.concurrency, yet this pattern is still widely applied

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
        implements BlockingQueue<E>,
        java.io.Serializable {

  public E take() throws
               InterruptedException {
    final ReentrantLock lock =
                          this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
  }
  ...
```

docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html has more

Monitor Object POSA2 Concurrency

Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- Android CancellationSignal

```
public final class CancellationSignal
{
    ...
    private boolean
            mCancelInProgress;

    public void setOnCancelListener
      (OnCancelListener listener) {
      ...
    }

    public void cancel() {
      ...
    }
}
```

frameworks/base/core/java/android/os/CancellationSignal.java has the code

Monitor Object POSA2 Concurrency

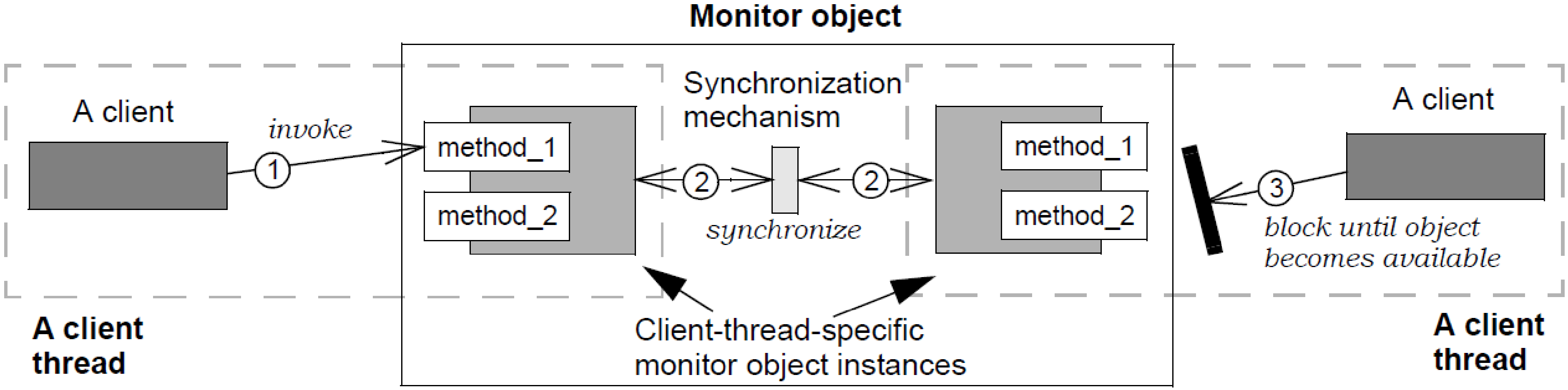
Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- Android CancellationSignal
- ACE provides portable C++ building blocks for implementing monitor objects

ACE Class
ACE_Guard ACE_Read_Guard ACE_Write_Guard
ACE_Thread_Mutex ACE_Process_Mutex ACE_Null_Mutex
ACE_RW_Thread_Mutex ACE_RW_Process_Mutex
ACE_Thread_Semaphore ACE_Process_Semaphore ACE_Null_Semaphore
ACE_Condition_Thread_Mutex ACE_Null_Condition

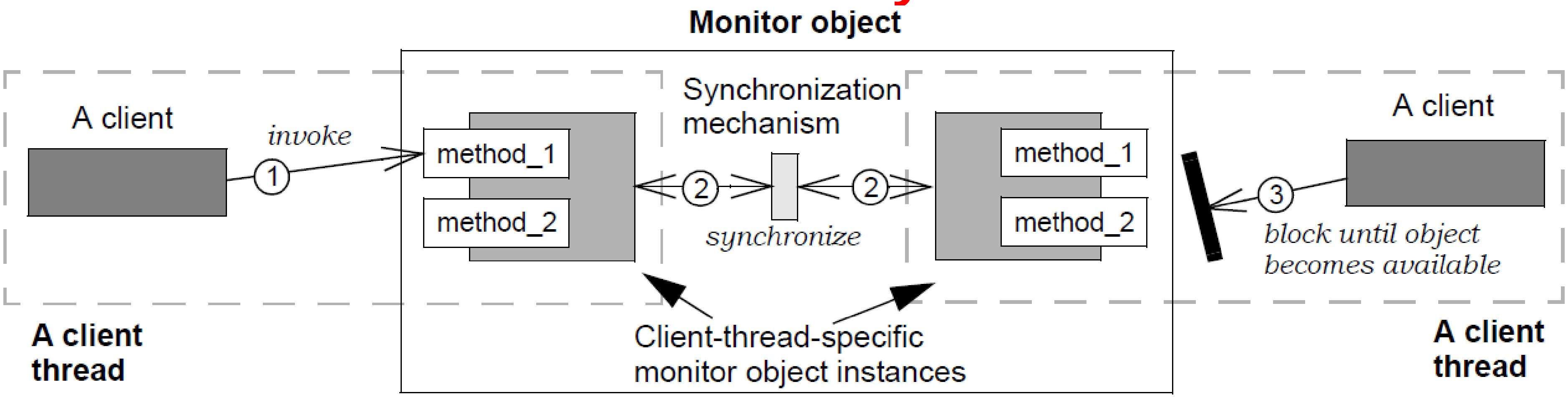
www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf has more info

Summary



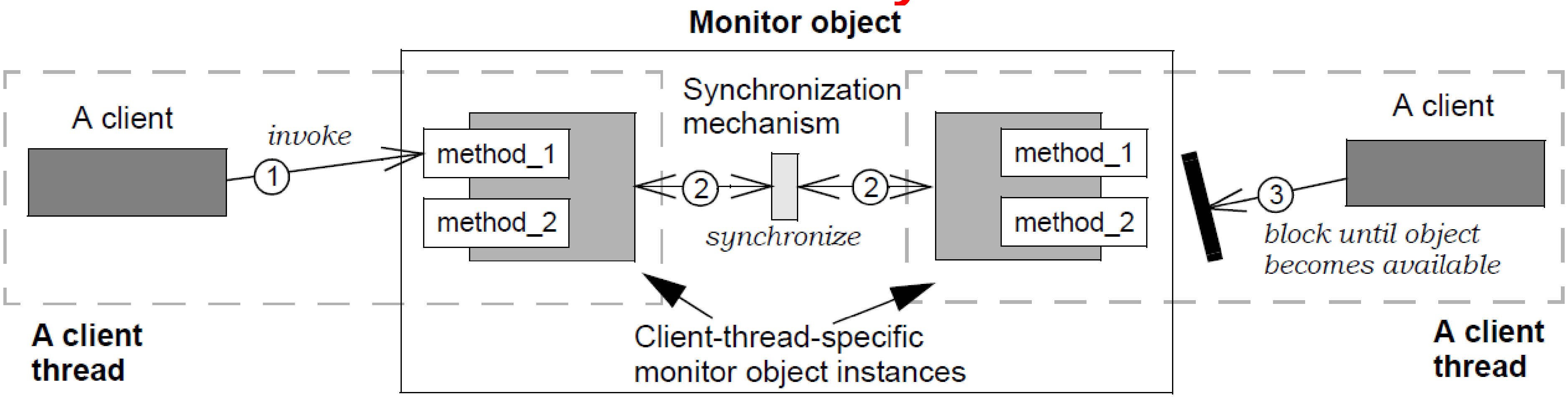
- Concurrent software often contains objects whose methods are invoked by multiple client threads

Summary



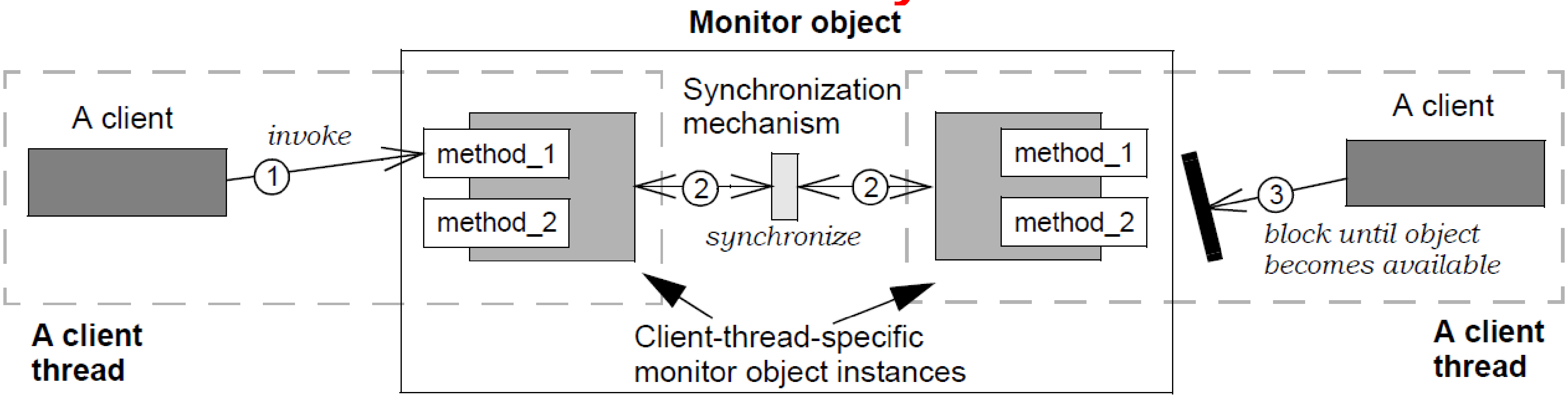
- Concurrent software often contains objects whose methods are invoked by multiple client threads
  - To protect the internal state of shared objects, it is necessary to synchronize & schedule client access to them

Summary



- Concurrent software often contains objects whose methods are invoked by multiple client threads
  - To protect the internal state of shared objects, it is necessary to synchronize & schedule client access to them
  - To simplify programming, however, clients should not need to distinguish programmatically between accessing shared & non-shared components

Summary



- Concurrent software often contains objects whose methods are invoked by multiple client threads
- The *Monitor Object* pattern enables the sharing of object by client threads that self-coordinate a serialized—yet interleaved—execution sequence

See www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf for *Monitor Object*