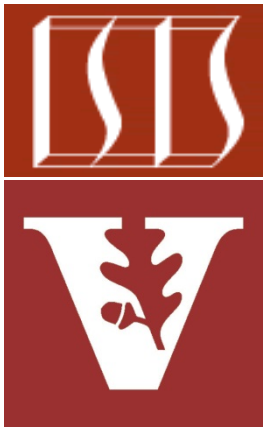
Android Concurrency: Java Semaphore



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

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

Learning Objectives in this Part of the Module

- Understand how Semaphores enable multiple threads to control access to a limited number of shared resources

Semaphore

[Expand All]

Added in API level 1

extends Object
implements Serializable

java.lang.Object
↳java.util.concurrent.Semaphore

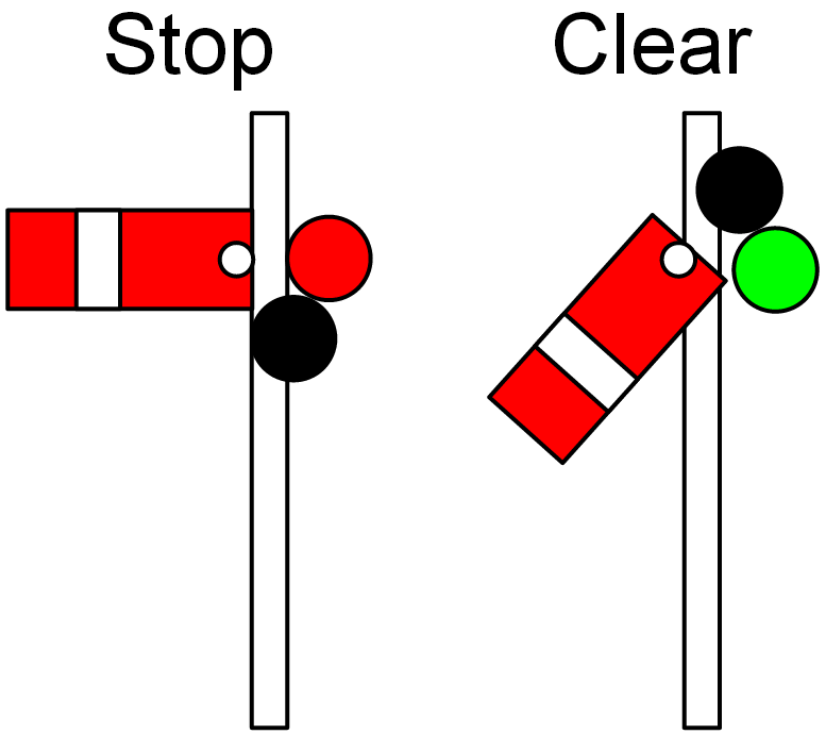
Class Overview

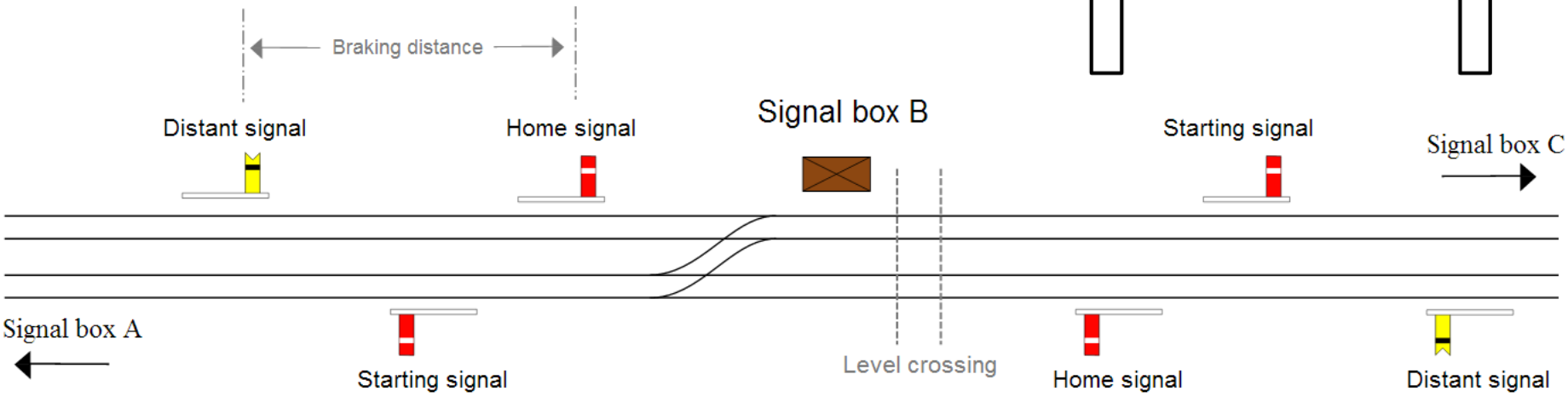
A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the `Semaphore` just keeps a count of the number available and acts accordingly.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource

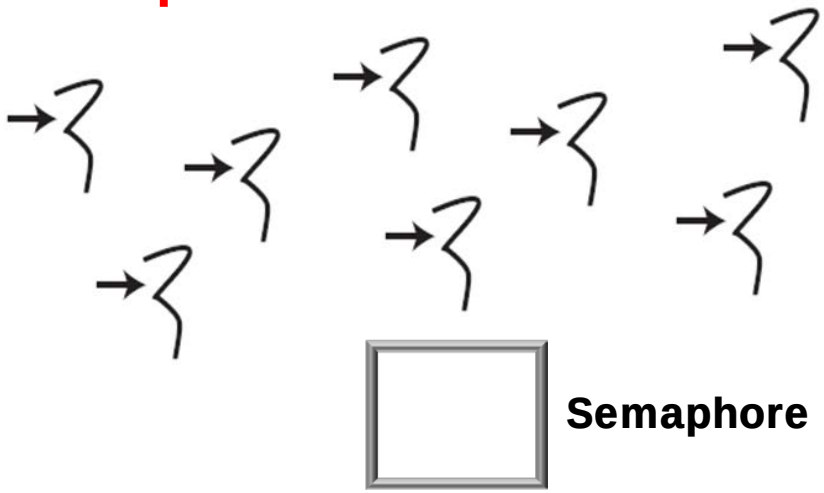




en.wikipedia.org/wiki/Railway_semaphore_signal has more info

Overview of Semaphores

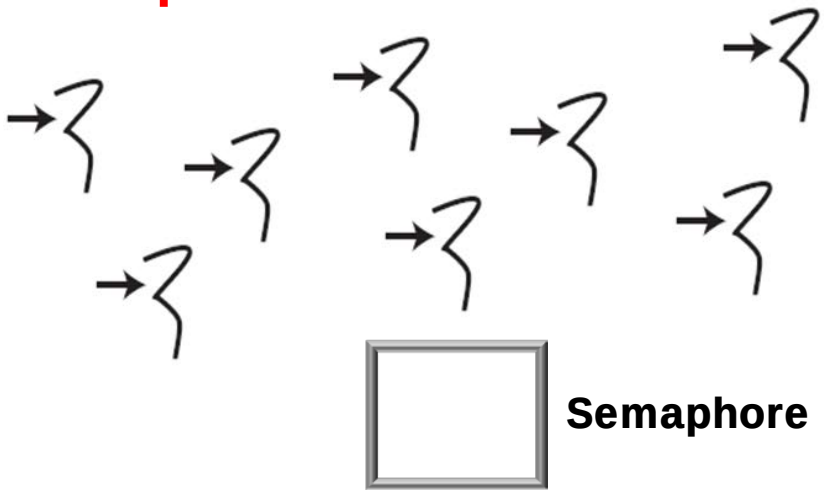
- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads

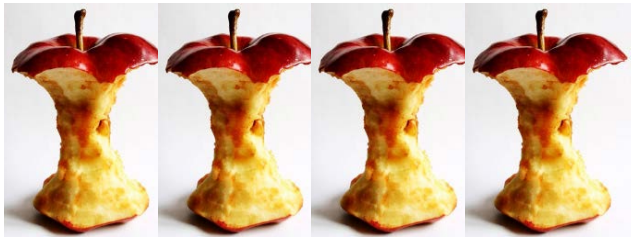


en.wikipedia.org/wiki/Semaphore_(programming)

Overview of Semaphores

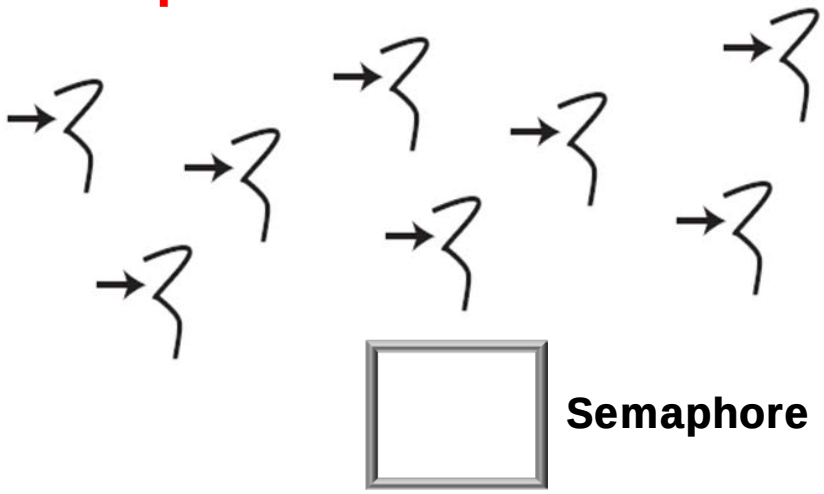
- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads
    - e.g., limit the number of resources devoted to a particular task

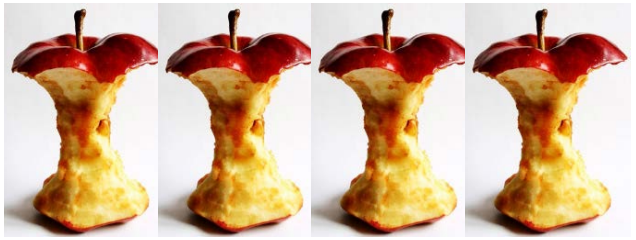




Overview of Semaphores

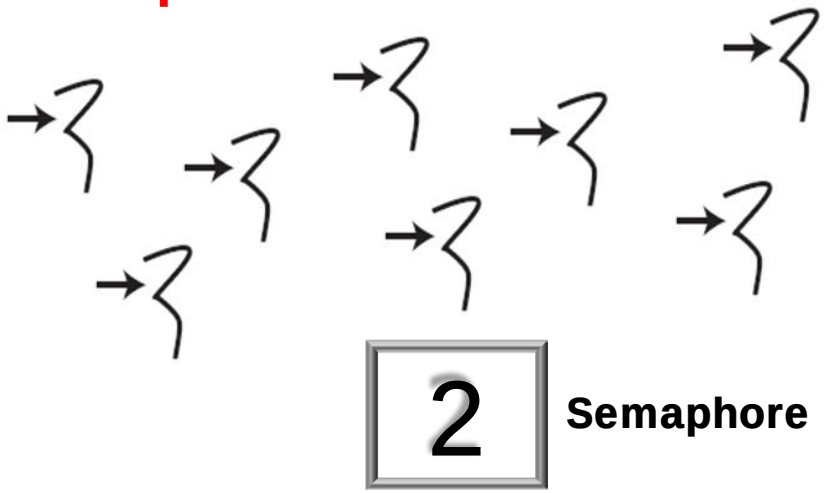
- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads
    - e.g., limit the number of resources devoted to a particular task
      - Consider image rendering threads accessing a fixed number of cores

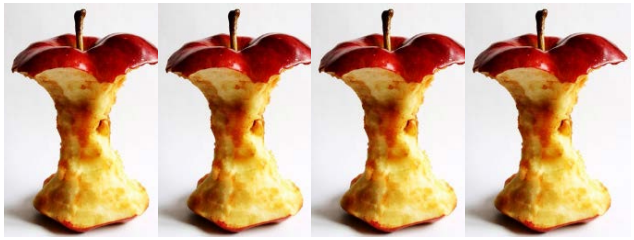




Overview of Semaphores

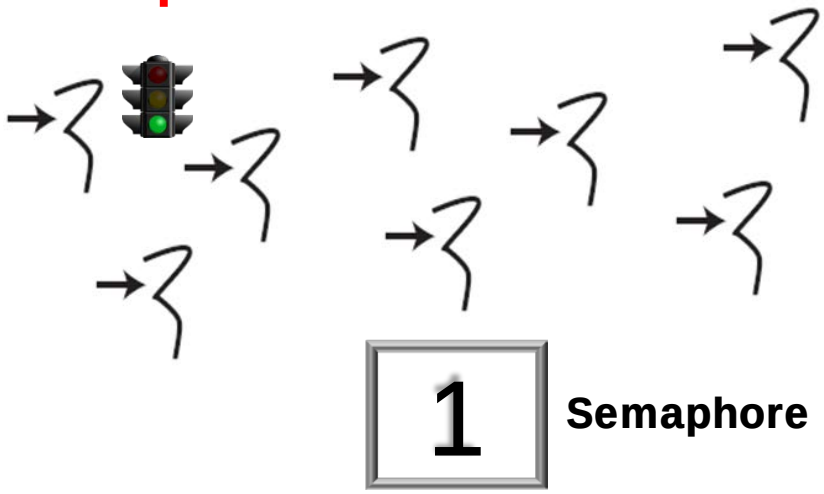
- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads
    - e.g., limit the number of resources devoted to a particular task
      - Consider image rendering threads accessing a fixed number of cores
      - Only allow use of two cores to ensure system responsiveness

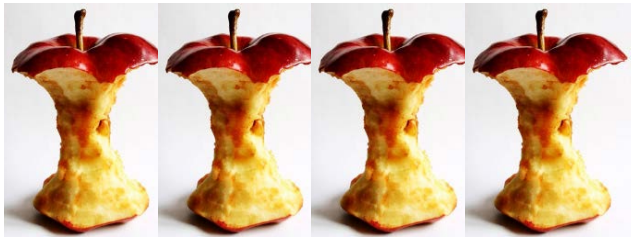




Overview of Semaphores

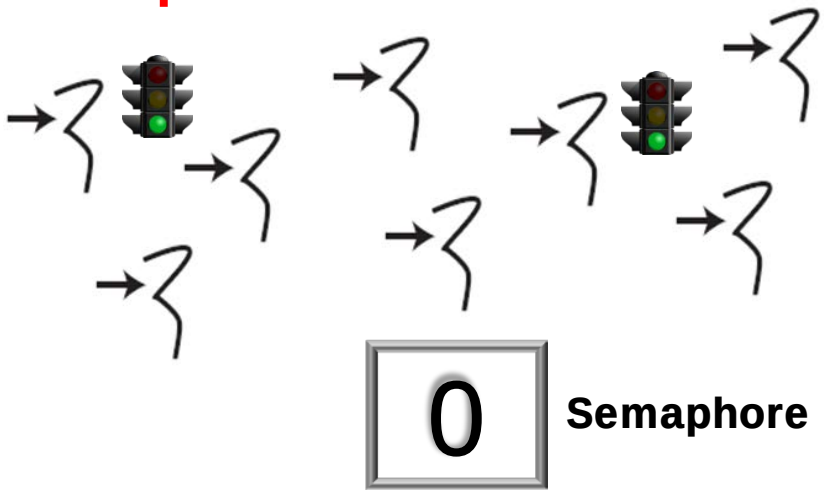
- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads
    - e.g., limit the number of resources devoted to a particular task
      - Consider image rendering threads accessing a fixed number of cores
      - Only allow use of two cores to ensure system responsiveness
      - A permit must be acquired from a semaphore before thread can run

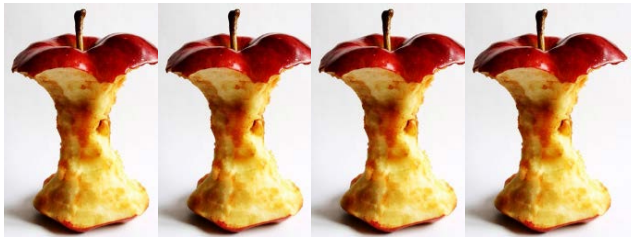




Overview of Semaphores

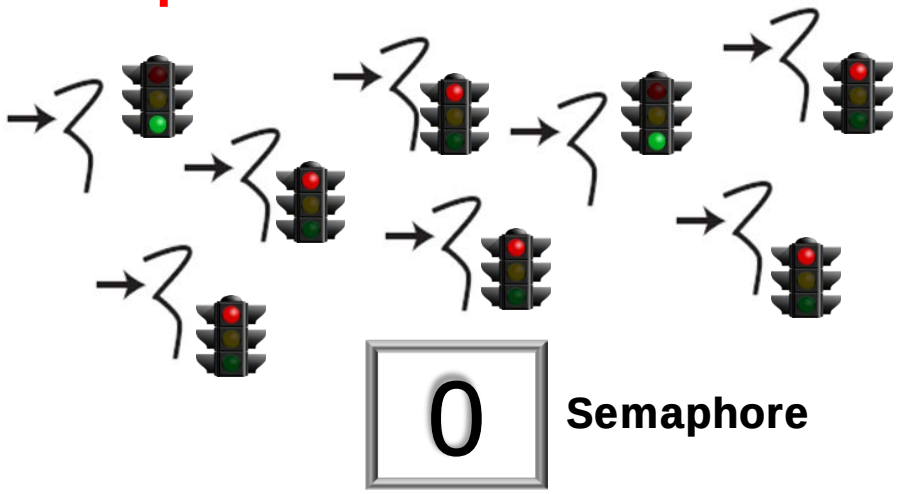
- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads
    - e.g., limit the number of resources devoted to a particular task
      - Consider image rendering threads accessing a fixed number of cores
      - Only allow use of two cores to ensure system responsiveness
      - A permit must be acquired from a semaphore before thread can run

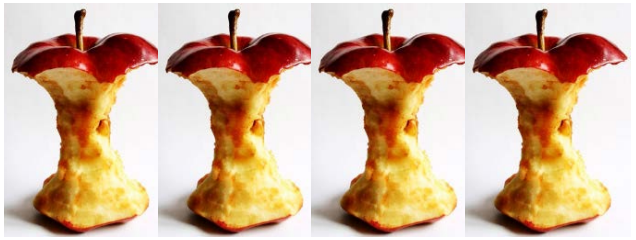




Overview of Semaphores

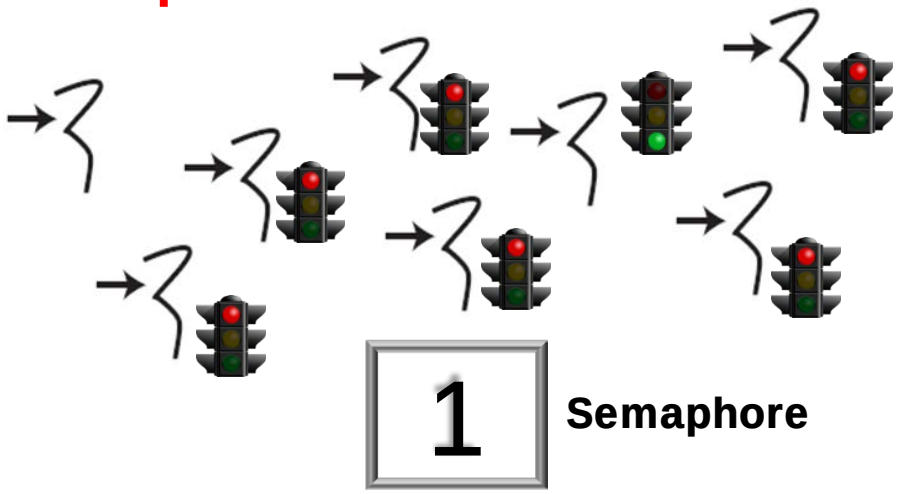
- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads
    - e.g., limit the number of resources devoted to a particular task
      - Consider image rendering threads accessing a fixed number of cores
      - Only allow use of two cores to ensure system responsiveness
      - A permit must be acquired from a semaphore before thread can run
      - Other threads will block

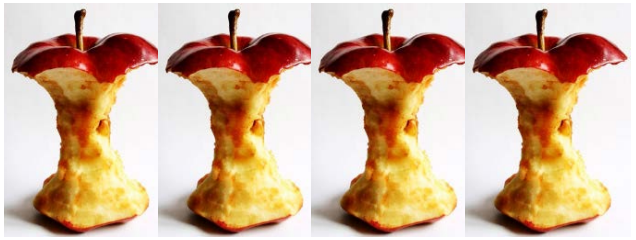




Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - They are used to synchronize & schedule the interactions between multiple concurrent threads
    - e.g., limit the number of resources devoted to a particular task
      - Consider image rendering threads accessing a fixed number of cores
      - Only allow use of two cores to ensure system responsiveness
      - A permit must be acquired from a semaphore before thread can run
      - Other threads will block, until a permit is returned to semaphore

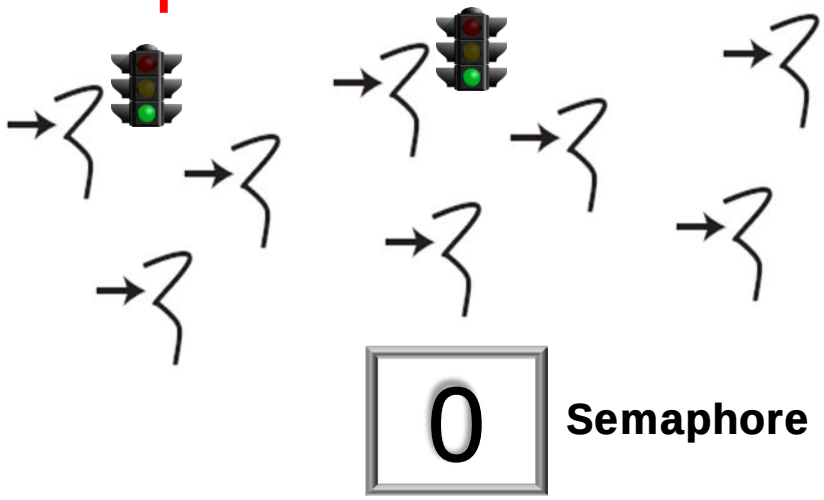


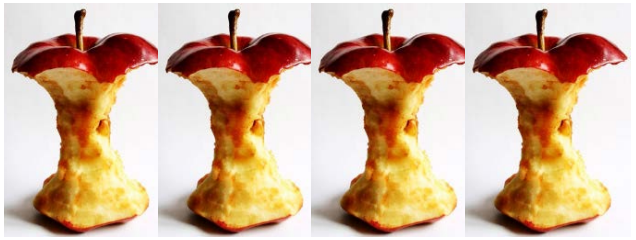


This example fully brackets acquire & release of permits to a semaphore

Overview of Semaphores

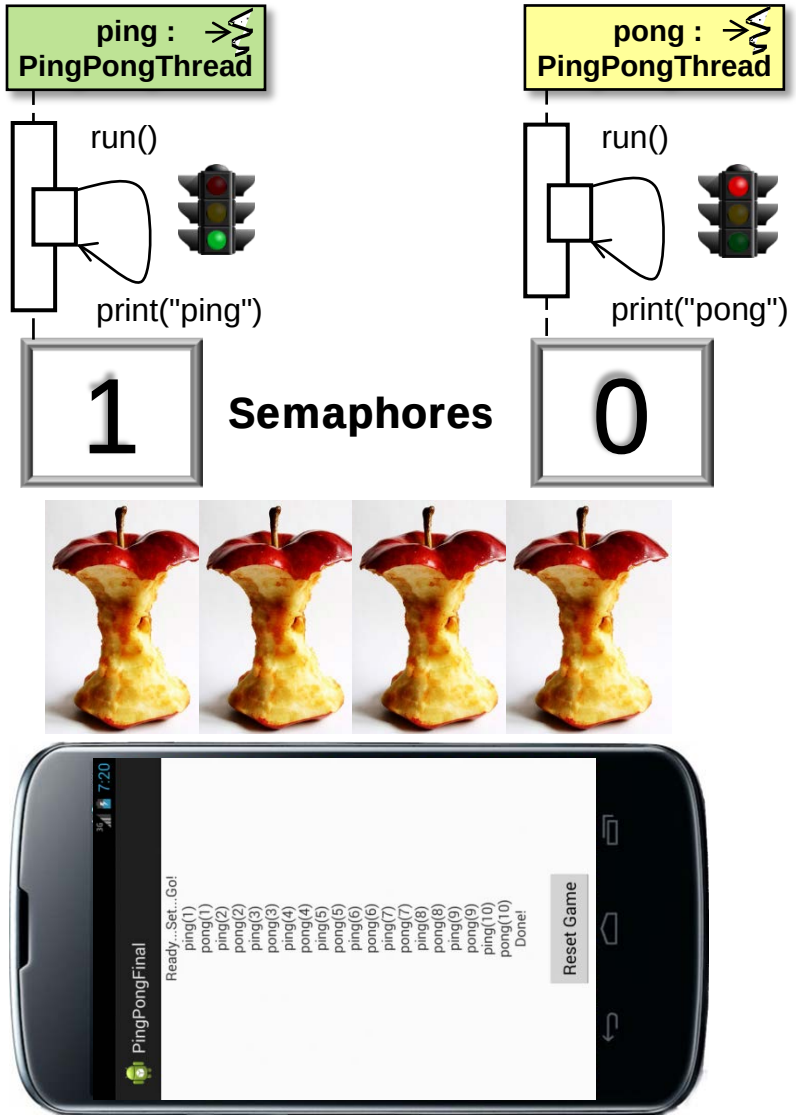
- A semaphore can be atomically incremented & decremented to control access to a shared resource
- There are two types of semaphores
  - **Counting semaphores**
    - Which allow an arbitrary resource count





Overview of Semaphores

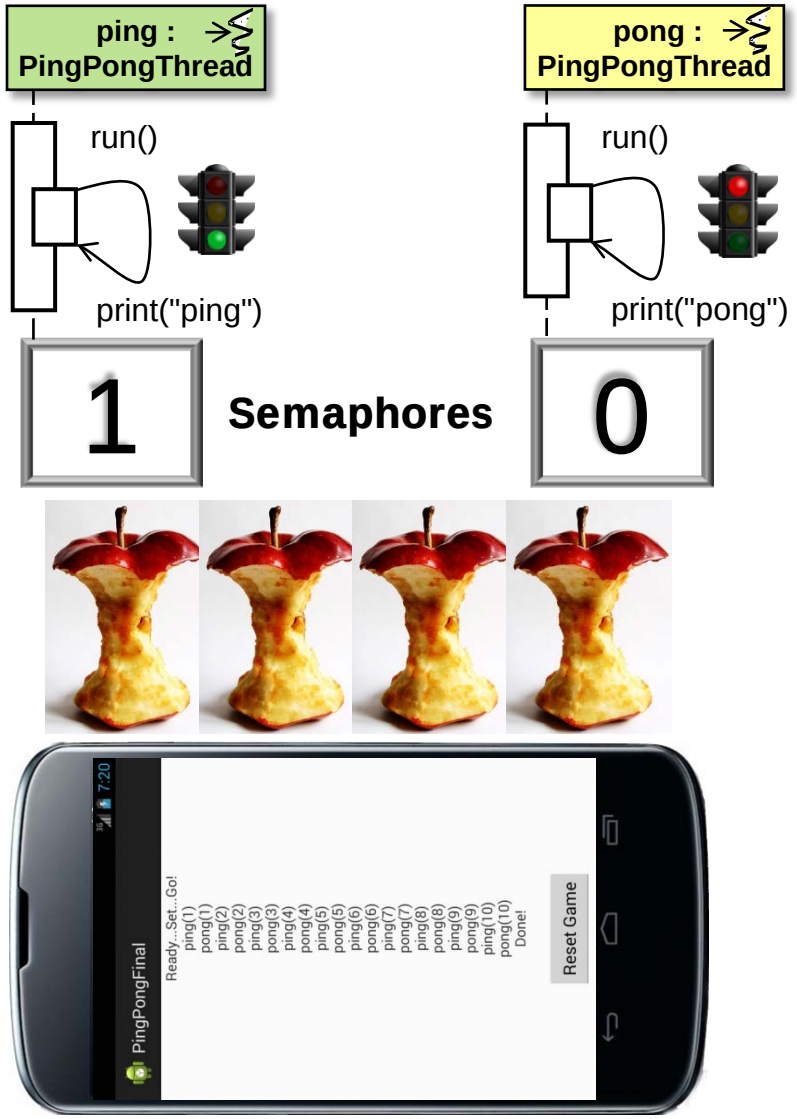
- A semaphore can be atomically incremented & decremented to control access to a shared resource
- There are two types of semaphores
  - **Counting semaphores**
  - **Binary semaphores**
    - Which are restricted to the values 0 & 1



See upcoming part on “Java Scheduling Example”

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- There are two types of semaphores
  - **Counting semaphores**
  - **Binary semaphores**
    - Which are restricted to the values 0 & 1

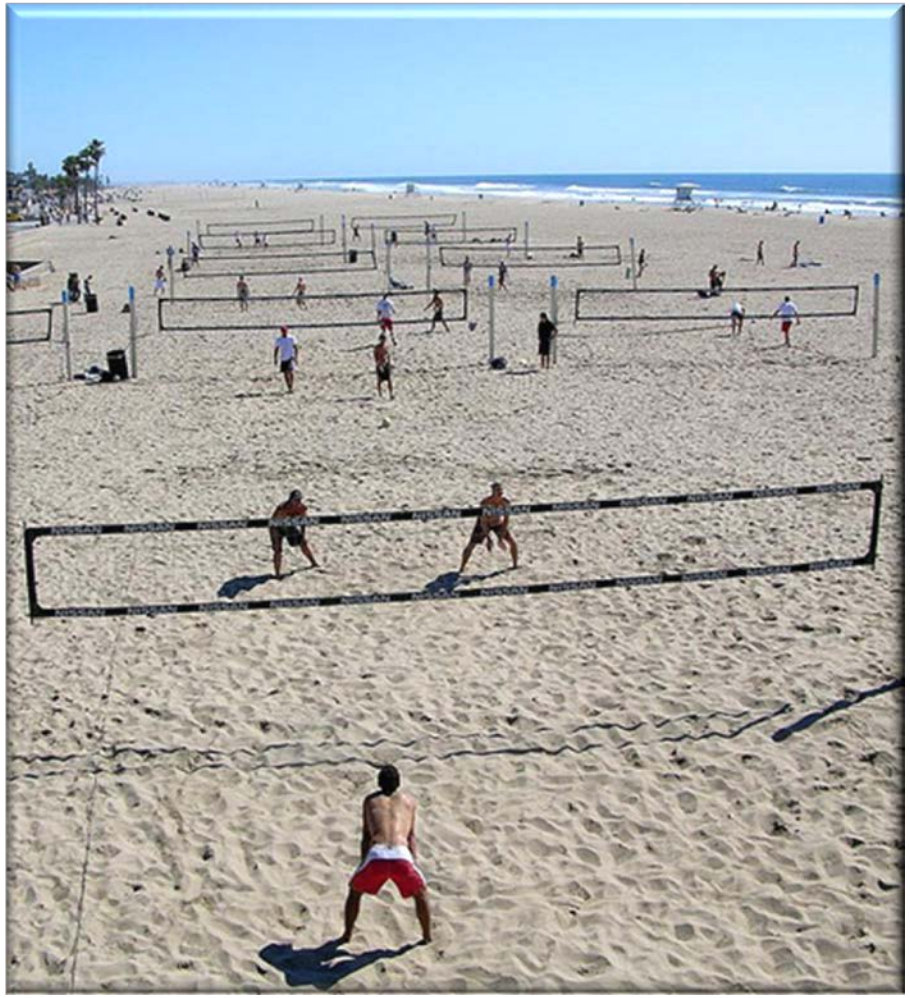


See earlier part on “Motivating Java Synchronization & Scheduling Mechanisms”

Human Known Use of Semaphores

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- There are two types of semaphores
- A human known use of counting semaphores applies them to schedule access to beach volleyball courts



en.wikipedia.org/wiki/Corona_del_Mar_State_Beach has more info

Overview of Java Semaphores

Overview of Java Semaphores

- The Semaphore class implements counting semaphores

Semaphore

[Expand All]

Added in API level 1

extends Object
implements Serializable

java.lang.Object
↳java.util.concurrent.Semaphore

Class Overview

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the `Semaphore` just keeps a count of the number available and acts accordingly.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:

See developer.android.com/reference/java/util/concurrent/Semaphore.html

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
  - Most of it is written in Java

```
public class Semaphore
             implements ... {
  ...
```

See libcore/luni/src/main/java/java/util/concurrent/Semaphore.java

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern

```
public class Semaphore
             implements ... {
  ...
  /** All mechanics via
      AbstractQueuedSynchronizer
      subclass */
  private final Sync sync;

  /**
    * Synchronization imp
    * for semaphore...
    */
  abstract static class Sync
      extends
      AbstractQueuedSynchronizer {
      ...
  }
```

en.wikipedia.org/wiki/Bridge_pattern contains more on the *Bridge* pattern

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class

```
public class Semaphore
             implements ... {
  ...
  /** All mechanics via
      AbstractQueuedSynchronizer
      subclass */
  private final Sync sync;

  /**
    * Synchronization imp
    * for semaphore...
    */
  abstract static class Sync
      extends
      AbstractQueuedSynchronizer {
      ...
  }
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...
  public Semaphore
              (int permits) {
    sync = new
      NonfairSync(permits);
  }

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new
        NonfairSync(permits);
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...
  public Semaphore
              (int permits) {
    sync = new
      NonfairSync(permits);
  }

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new
        NonfairSync(permits);
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...
  public Semaphore
              (int permits) {
    sync = new
      NonfairSync(permits);
  }

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new
        NonfairSync(permits);
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...
  public Semaphore
              (int permits) {
    sync = new
      NonfairSync(permits);
  }

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new
        NonfairSync(permits);
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model
  - The constructors create a Semaphore with a designated number of permits

```
public class Semaphore
             implements ... {
  ...
  public Semaphore
              (int permits) {
    sync = new
      NonfairSync(permits);
  }

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new
        NonfairSync(permits);
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
- Its key methods are acquire(), tryAcquire(), & release()

```
public class Semaphore
             implements ... {
  ...
  public void acquire() { ... }

  public void
    acquireUninterruptibly()
  { ... }

  public boolean tryAcquire
         (long timeout,
          TimeUnit unit)
  { ... }

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

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
- Its key methods are acquire(), acquireUninterruptibly(), tryAcquire(), & release()
  - acquire() obtains a permit from the semaphore

```
public class Semaphore
             implements ... {
  ...
  public void acquire() {
    sync.
    acquireSharedInterruptibly(1);
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
- Its key methods are acquire(), acquireUninterruptibly(), tryAcquire(), & release()
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts

```
public class Semaphore
             implements ... {
  ...
  public void
    acquireUninterruptibly() {
    sync.acquireShared(1)
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
- Its key methods are acquire(), acquireUninterruptibly(), tryAcquire(), & release()
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts
  - tryAcquire() obtains a permit if one becomes available in time

```
public class Semaphore
             implements ... {
  ...
  public boolean tryAcquire
         (long timeout,
          TimeUnit unit) ... {
    sync.
      nonfairTryAcquireShared(1)
      >= 0;
  }
  ...
```

Overview of Java Semaphores

- The Semaphore class implements counting semaphores
- Semaphore uses the *Bridge* pattern
- Its key methods are acquire(), acquireUninterruptibly(), tryAcquire(), & release()
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts
  - tryAcquire() obtains a permit if one becomes available in time
  - Release() returns a permit, increasing number by 1

```
public class Semaphore
             implements ... {
  ...
  public void release() {
    sync.releaseShared(1);
  }
  ...
```

Acquiring & releasing permits to a Semaphore need not be fully bracketed

Applying Java Semaphores on Android

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality

Interface VideoEditor

All Known Implementing Classes:

VideoEditorImpl

```
public interface VideoEditor
```

This is the interface implemented by classes which provide video editing functionality. The VideoEditor implementation class manages all input and output files. Unless specifically mentioned, methods are blocking. A typical editing session may consist of the following sequence of operations:

- Add a set of MediaItems
- Apply a set of Transitions between MediaItems
- Add Effects and Overlays to media items
- Preview the movie at any time
- Save the VideoEditor implementation class internal state
- Release the VideoEditor implementation class instance by invoking `release()`

The internal VideoEditor state consists of the following elements:

- Ordered & trimmed MediaItems
- Transition video clips
- Overlays
- Effects
- Audio waveform for the background audio and MediaItems
- Project thumbnail
- Last exported movie.
- Other project specific data such as the current aspect ratio.

developer.oesf.biz/em/developer/reference/eggplant/android/media/videoeditor/VideoEditor.html

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality

Interface VideoEditor

All Known Implementing Classes:

VideoEditorImpl

```
public interface VideoEditor
```

This is the interface implemented by classes which provide video editing functionality. The VideoEditor implementation class manages all input and output files. Unless specifically mentioned, methods are blocking. A typical editing session may consist of the following sequence of operations:

- Add a set of MediaItems
- Apply a set of Transitions between MediaItems
- Add Effects and Overlays to media items
- Preview the movie at any time
- Save the VideoEditor implementation class internal state
- Release the VideoEditor implementation class instance by invoking `release()`

The internal VideoEditor state consists of the following elements:

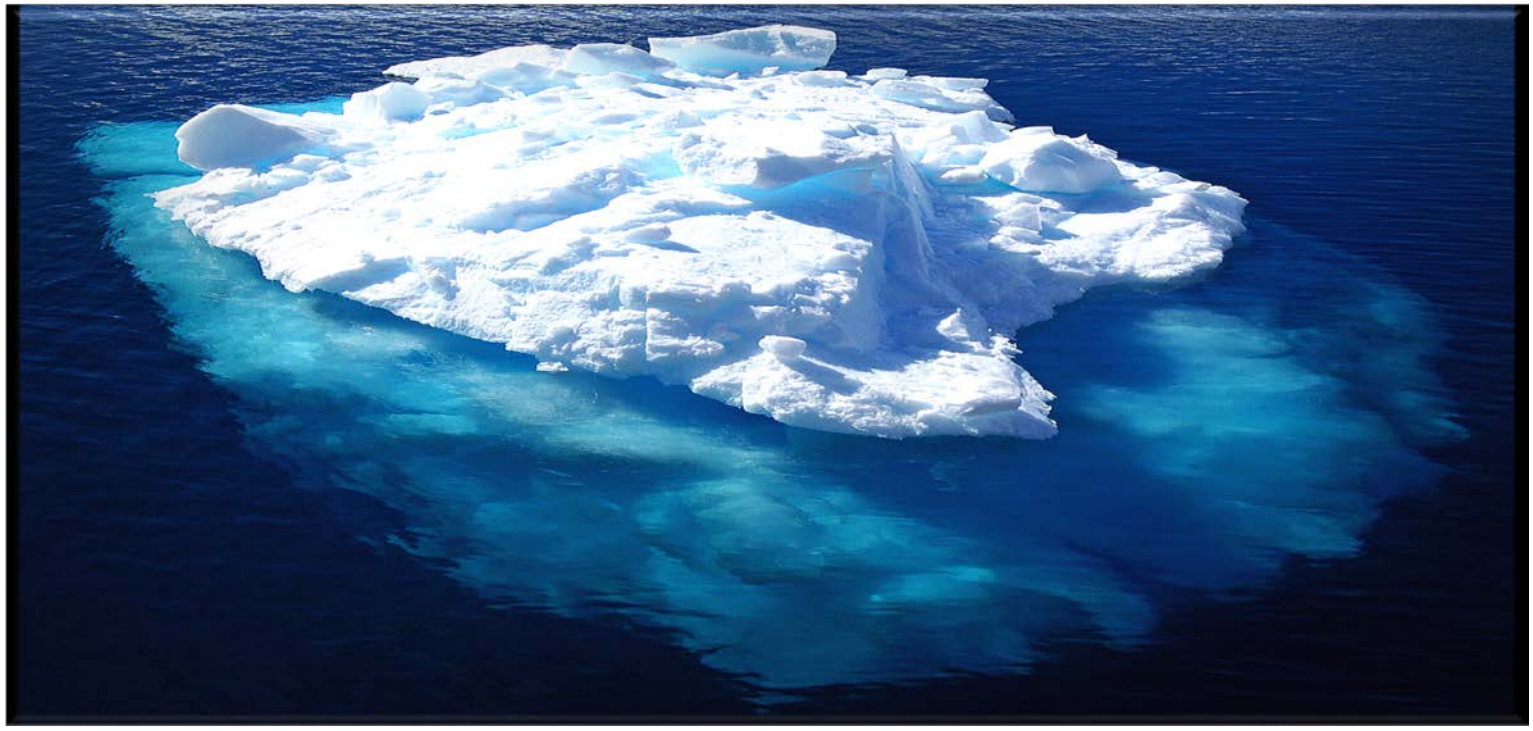
- Ordered & trimmed MediaItems
- Transition video clips
- Overlays
- Effects
- Audio waveform for the background audio and MediaItems
- Project thumbnail
- Last exported movie.
- Other project specific data such as the current aspect ratio.

packages/apps/VideoEditor has the video editor app source code

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
```



frameworks/base/media/java/android/media/videoeditor/VideoEditorImpl.java

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  private final Semaphore mLock;
  ...
  public VideoEditorImpl(...) ... {
    ...
    mLock = new Semaphore(1, true);
    ...
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  private final Semaphore mLock;
  ...
  public VideoEditorImpl(...) ... {
    ...
    mLock = new Semaphore(1, true);
    ...
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model
  - It defines lock() & unlock() methods that call tryAcquire() & release() on the semaphore

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  private final Semaphore mLock;
  ...

  private boolean lock
    (long timeoutMs) {
    ...
    mLock.tryAcquire(timeoutMs, ...);
    ...
  }

  private void unlock() {
    mLock.release();
  }
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model
  - It defines lock() & unlock() methods that call tryAcquire() & release() on the semaphore

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  private final Semaphore mLock;
  ...

  private boolean lock
    (long timeoutMs) {
    ...
    mLock.tryAcquire(timeoutMs, ...);
    ...
  }

  private void unlock() {
    mLock.release();
  }
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model
  - It defines lock() & unlock() methods that call tryAcquire() & release() on the semaphore
  - startPreview() tries to acquire the semaphore to do a preview

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  public void startPreview(…) {
    ...
    if (!mPreviewInProgress) {
      ...
      lock(...MAX_TIMEOUT_MS);
      ...
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model
  - It defines lock() & unlock() methods that call tryAcquire() & release() on the semaphore
  - startPreview() tries to acquire the semaphore to do a preview

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  public void startPreview(…) {
    ...
    if (!mPreviewInProgress) {
      ...
      lock(...MAX_TIMEOUT_MS);
      ...
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model
  - It defines lock() & unlock() methods that call tryAcquire() & release() on the semaphore
  - startPreview() tries to acquire the semaphore to do a preview
  - stopPreview() stops the preview & releases the semaphore

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  public long stopPreview() {
    ...
    if (mPreviewInProgress) {
      ...
      mPreviewInProgress = false;
      unlock();
      ...
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
  - It creates a binary semaphore that uses the “fair” acquisition model
  - It defines lock() & unlock() methods that call tryAcquire() & release() on the semaphore
  - startPreview() tries to acquire the semaphore to do a preview
  - stopPreview() stops the preview & releases the semaphore

```
public class VideoEditorImpl
             implements VideoEditor {
  ...
  public long stopPreview() {
    ...
    if (mPreviewInProgress) {
      ...
      mPreviewInProgress = false;
      unlock();
      ...
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
- VideoEditorActivity handles user requests on a project

```
public class VideoEditorActivity ... {
   ...
```

packages/apps/VideoEditor/src/com/android/videoeditor/VideoEditorActivity.java

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
- VideoEditorActivity handles user requests on a project

```
public class VideoEditorActivity ... {
  ...
  private class PreviewThread
                extends Thread {
    ...
```

Preview Thread

Semaphore

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
- VideoEditorActivity handles user requests on a project

```
public class VideoEditorActivity ... {
  ...
  private class PreviewThread
                extends Thread {
    ...
    private void
      startPreviewPlayback(...) {
      ...
        mQueue.add(new Runnable() {
          public void run() {
            project.startPreview(...)
            ...
```

Preview Thread

Semaphore

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
- VideoEditorActivity handles user requests on a project

Preview Thread

Semaphore

```
public class VideoEditorActivity ... {
  ...
  private class PreviewThread
                  extends Thread {
    ...
    private void
      startPreviewPlayback(...) {
      ...
        mQueue.add(new Runnable() {
          public void run() {
            project.startPreview(...)
            ...
```

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
- VideoEditorActivity handles user requests on a project

```
public class VideoEditorActivity ... {
  ...
  private class PreviewThread
                  extends Thread {
    ...
    private void previewStopped(...) {
      ...
      mProject.stopPreview();
      ...
```

Preview Thread

UI Thread

Semaphore

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
- VideoEditorActivity handles user requests on a project

```
public class VideoEditorActivity ... {
  ...
  private class PreviewThread
                  extends Thread {
    ...
    private void previewStopped(...) {
      ...
      mProject.stopPreview();
      ...
```

Preview Thread

UI Thread

Semaphore

Applying Java Semaphores on Android

- VideoEditor is an interface to video editing functionality
- VideoEditorImpl implements the VideoEditor interface
- VideoEditorActivity handles user requests on a project

```
public class VideoEditorActivity ... {
  ...
  private class PreviewThread
                  extends Thread {
    ...
    private void previewStopped(...) {
      ...
      mProject.stopPreview();
      ...
```

Preview Thread

UI Thread

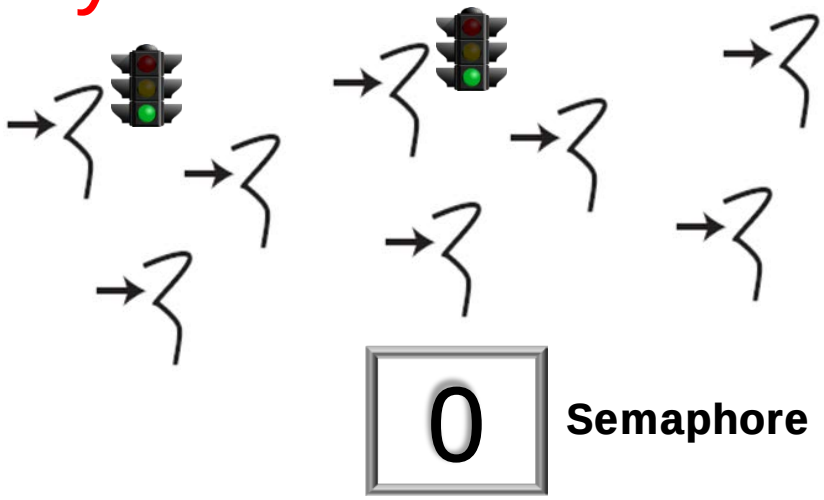
Semaphore

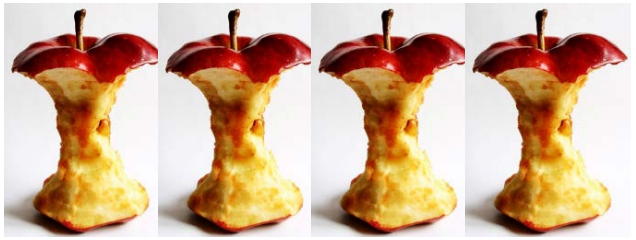
This example doesn't fully bracket acquire & release of permits to a semaphore

Summary

Summary

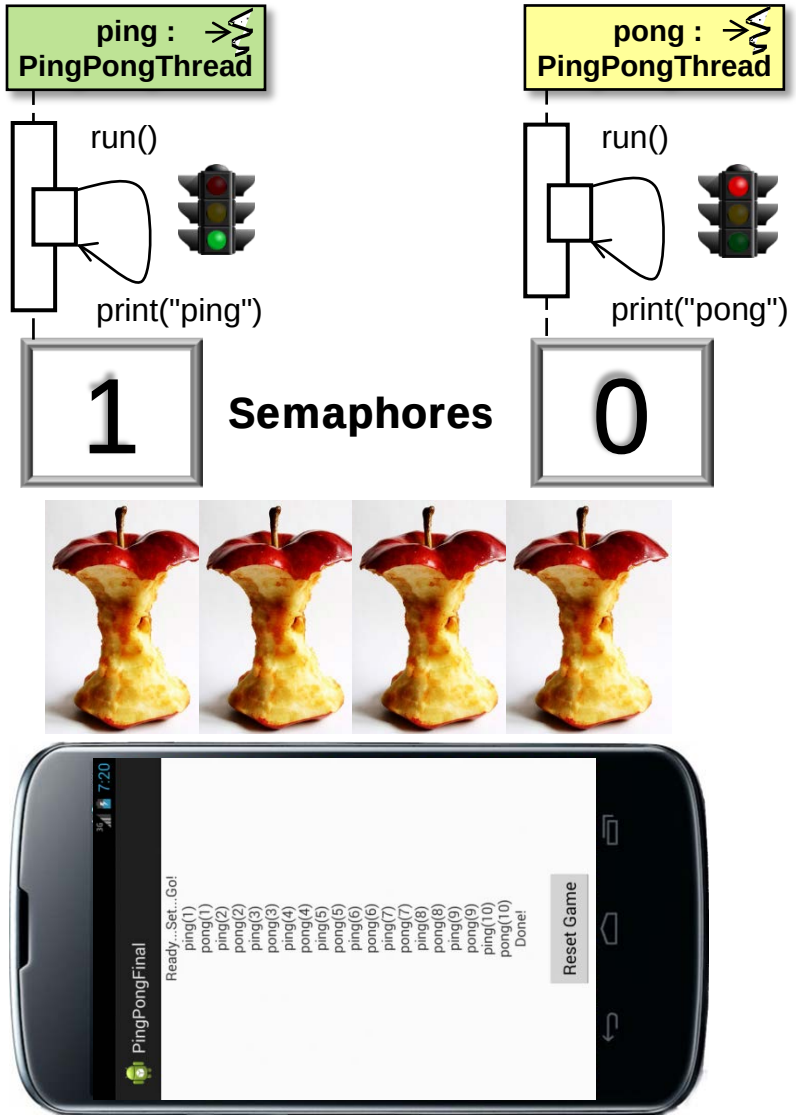
- A Semaphore provides a flexible synchronization & scheduling mechanism





Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
  - Its acquire & release methods need not be fully bracketed



Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
  - Its acquire & release methods need not be fully bracketed
  - It supports several types of acquire & release operations
    - e.g., blocking, non-blocking, timed, & multi-permit

void	acquire() Acquires a permit from this semaphore, blocking until one is available, or the thread is `interrupted`.
void	acquire(int permits) Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is `interrupted`.
void	acquireUninterruptibly() Acquires a permit from this semaphore, blocking until one is available.
void	acquireUninterruptibly(int permits) Acquires the given number of permits from this semaphore, blocking until all are available.
boolean	tryAcquire(long timeout, TimeUnit unit) Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been interrupted.
boolean	tryAcquire() Acquires a permit from this semaphore, only if one is available at the time of invocation.
boolean	tryAcquire(int permits, long timeout, TimeUnit unit) Acquires the given number of permits from this semaphore, if all become available within the given waiting time and the current thread has not beeninterrupted.
boolean	tryAcquire(int permits) Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.

Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
  - Its acquire & release methods need not be fully bracketed
  - It supports several types of acquire & release operations
  - Its blocking operations are implemented via “sleep” locks, which trigger a context switch

stackoverflow.com/questions/195853/spinlock-versus-semaphore has more info

Summary

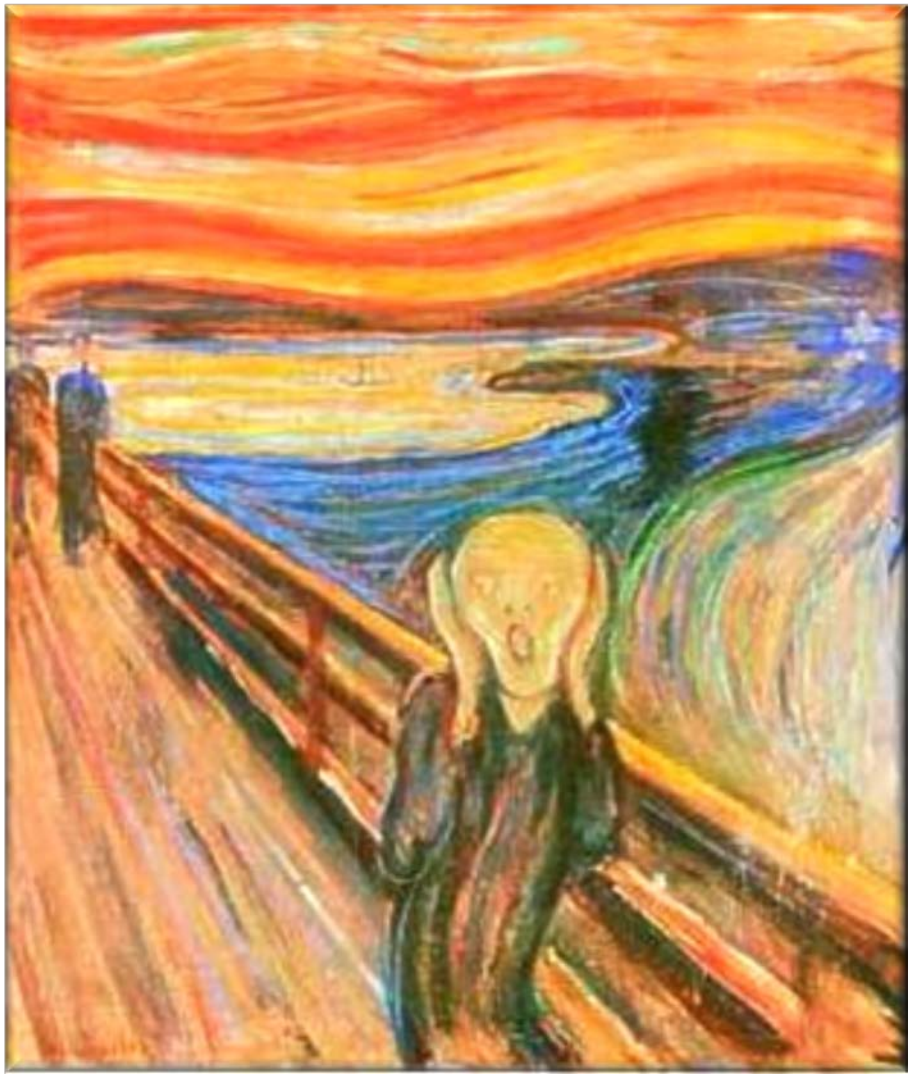
- A Semaphore provides a flexible synchronization & scheduling mechanism
  - Its acquire & release methods need not be fully bracketed
  - It supports several types of acquire & release operations
  - Its blocking operations are implemented via “sleep” locks, which trigger a context switch
  - When used for a resource pool, it tracks how many resources are free, *not* which resources are free

Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
  - Its acquire & release methods need not be fully bracketed
  - It supports several types of acquire & release operations
  - Its blocking operations are implemented via “sleep” locks, which trigger a context switch
- When used for a resource pool, it tracks how many resources are free, *not* which resources are free
  - Other mechanisms may be needed to select a particular free resource

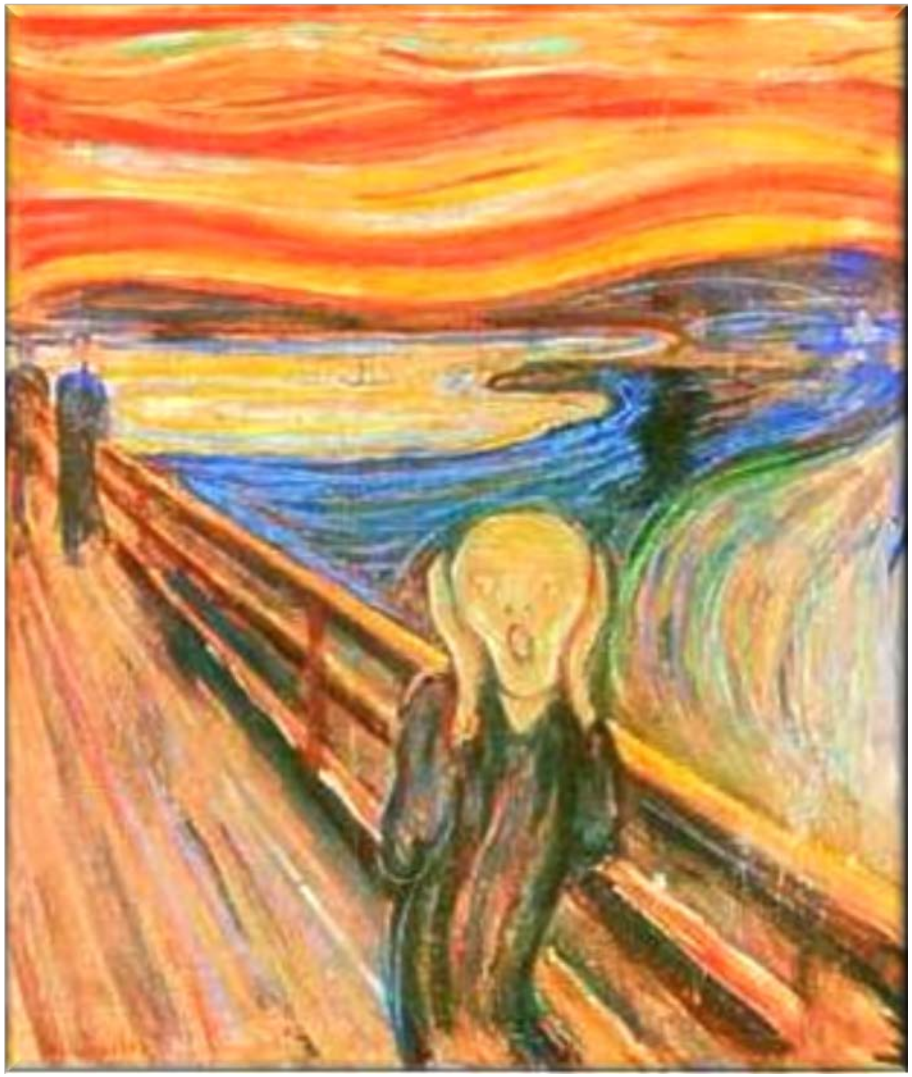
Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
- Semaphores can be tedious & error-prone to program due to the following common mistakes
  - Requesting a permit & forgetting to release it



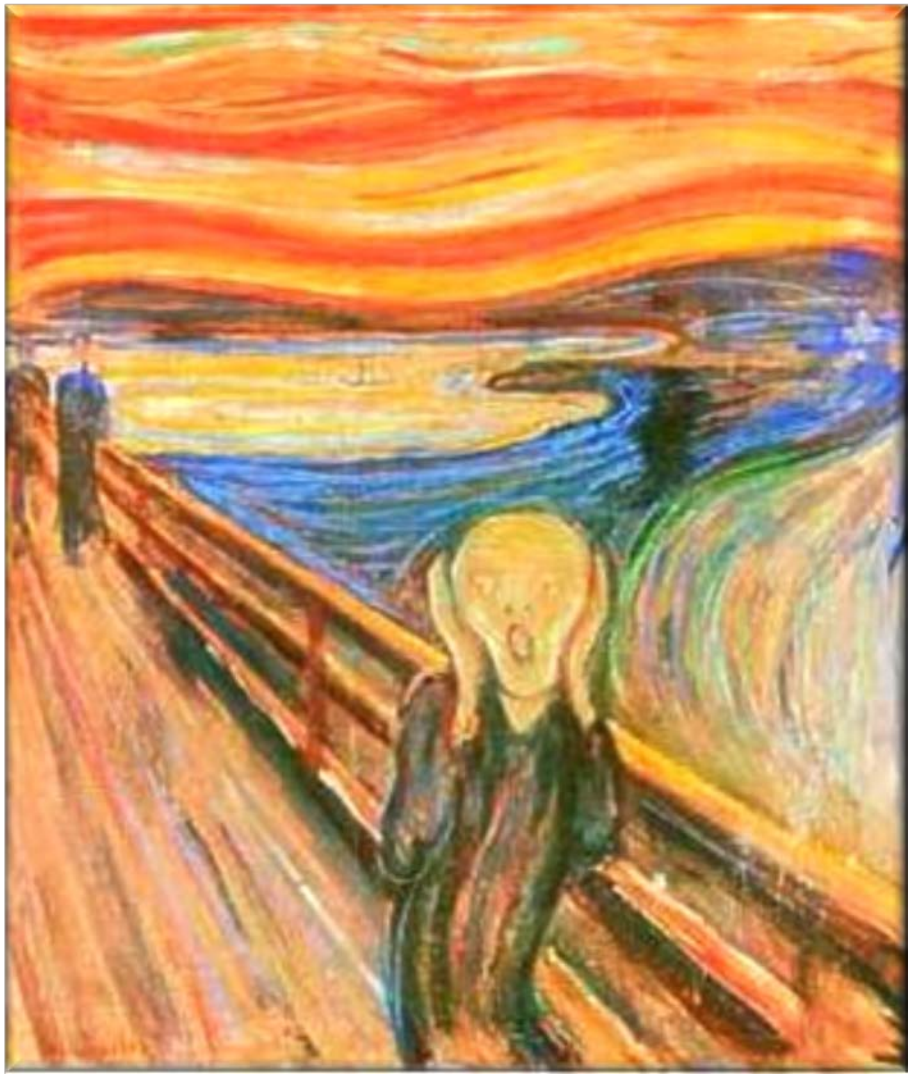
Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
- Semaphores can be tedious & error-prone to program due to the following common mistakes
  - Requesting a permit & forgetting to release it
  - Releasing a permit that was never requested



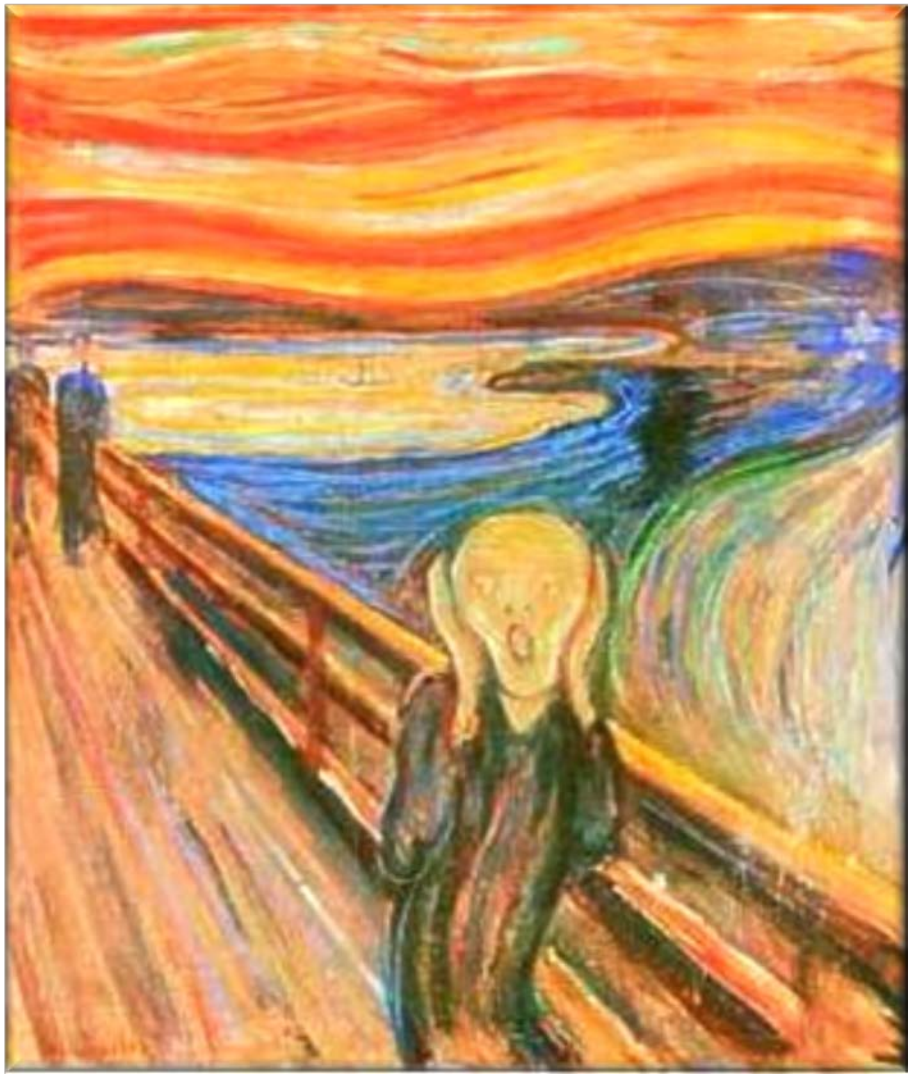
Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
- Semaphores can be tedious & error-prone to program due to the following common mistakes
  - Requesting a permit & forgetting to release it
  - Releasing a permit that was never requested
  - Holding a permit for a long time without needing it



Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
- Semaphores can be tedious & error-prone to program due to the following common mistakes
  - Requesting a permit & forgetting to release it
  - Releasing a permit that was never requested
  - Holding a permit for a long time without needing it
  - Accessing a resource without requesting a permit for it first (or after releasing it)



Summary

- A Semaphore provides a flexible synchronization & scheduling mechanism
- Semaphores can be tedious & error-prone to program due to the following common mistakes
- Java Semaphores are rarely seen in Android
  - Primarily applied in unit tests