Android Concurrency & Synchronization: Introduction

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

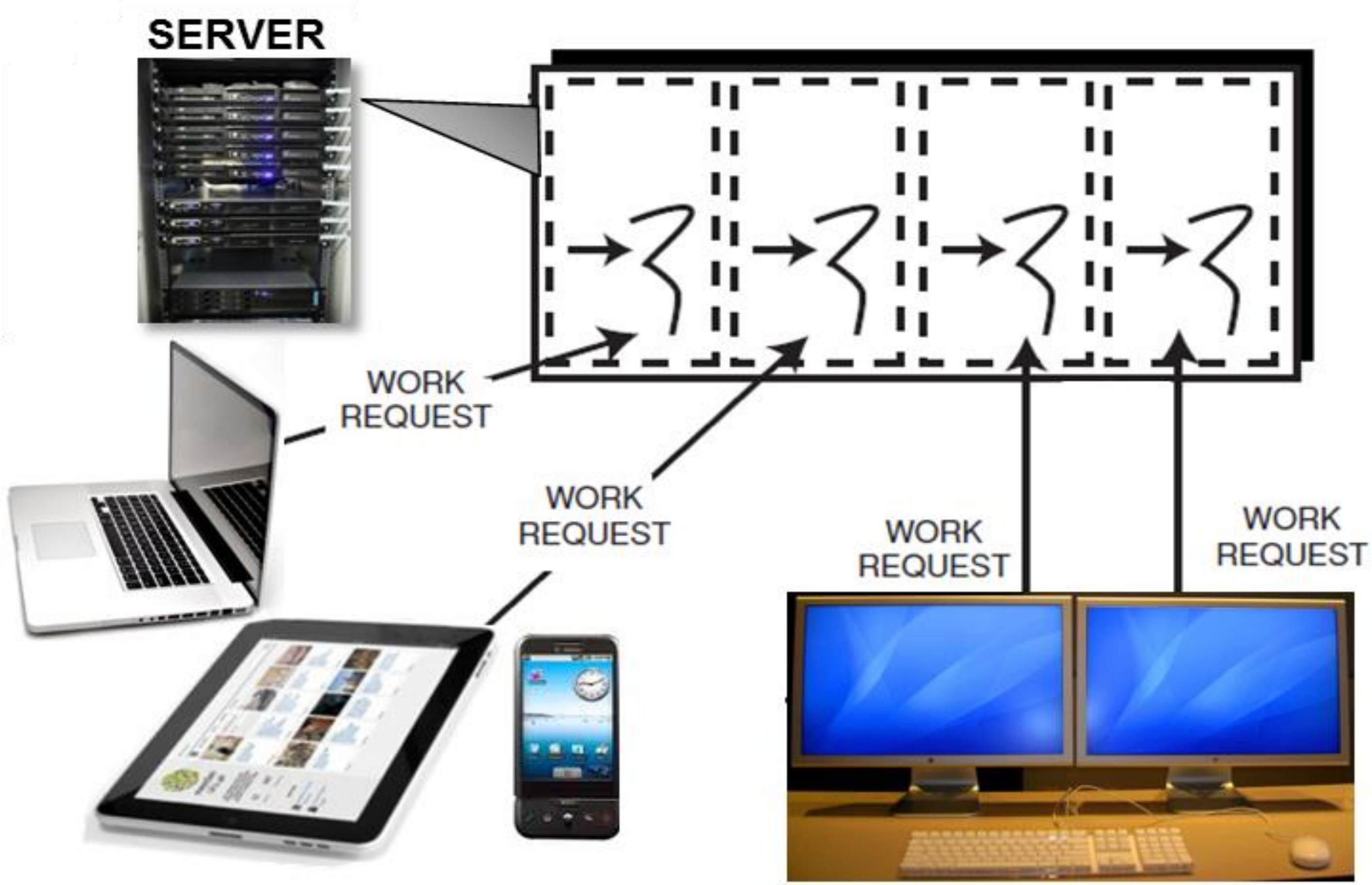
Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

CS 282 Principles of Operating Systems II

Systems Programming for Android

Introduction

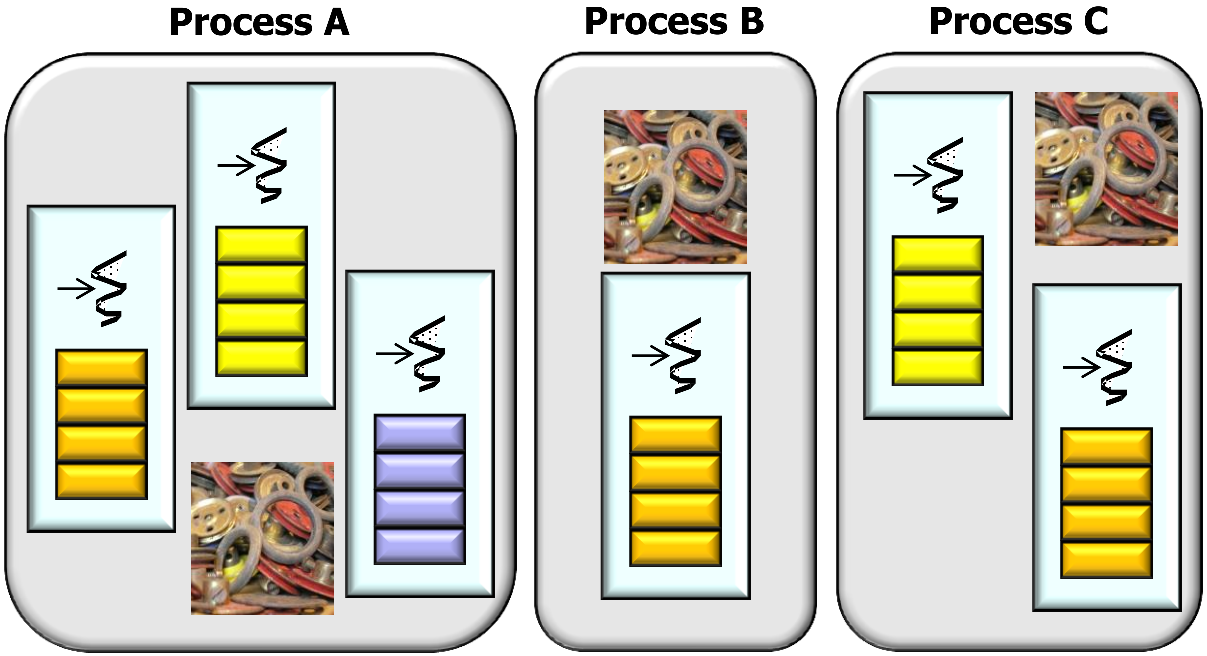
- Explore the motivations for & challenges of concurrent software

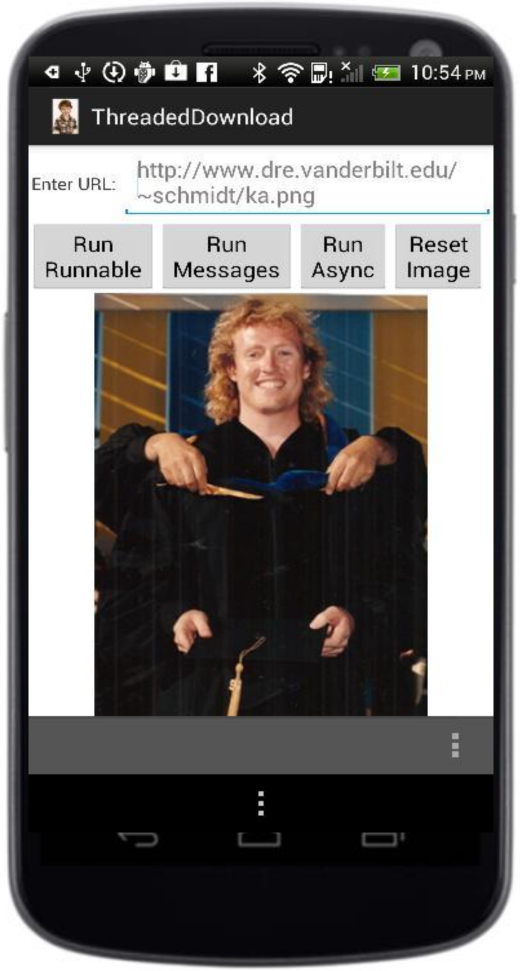


Concurrent software can simultaneously run multiple computations that potentially interact with each other

Introduction

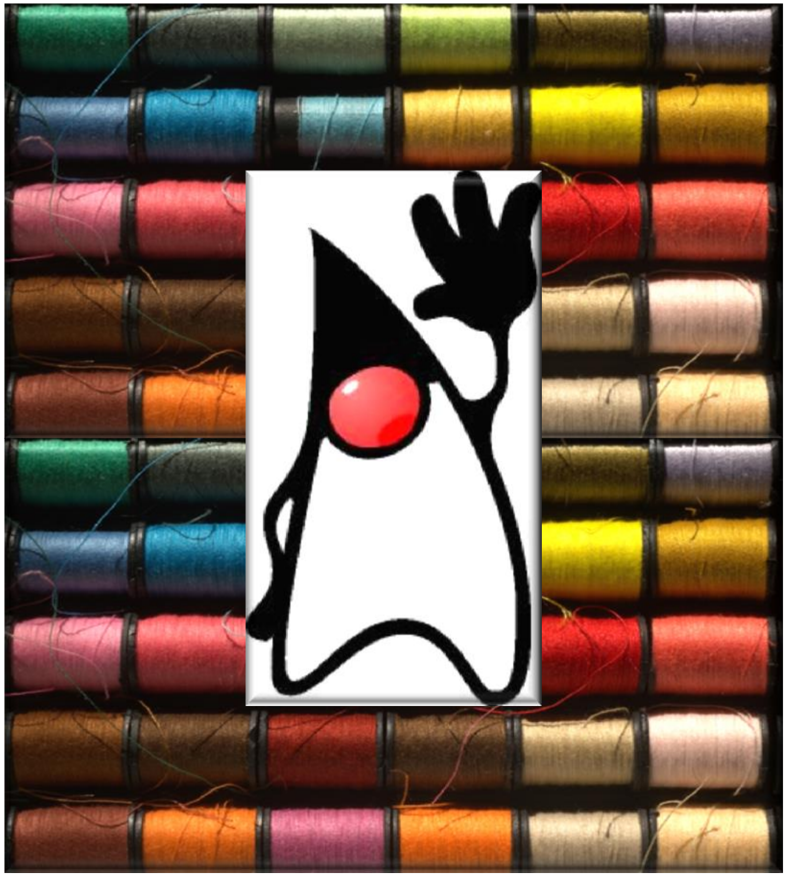
- Explore the motivations for & challenges of concurrent software
- Understand the mechanisms that Android provides to manage multiple threads that run concurrently within a process





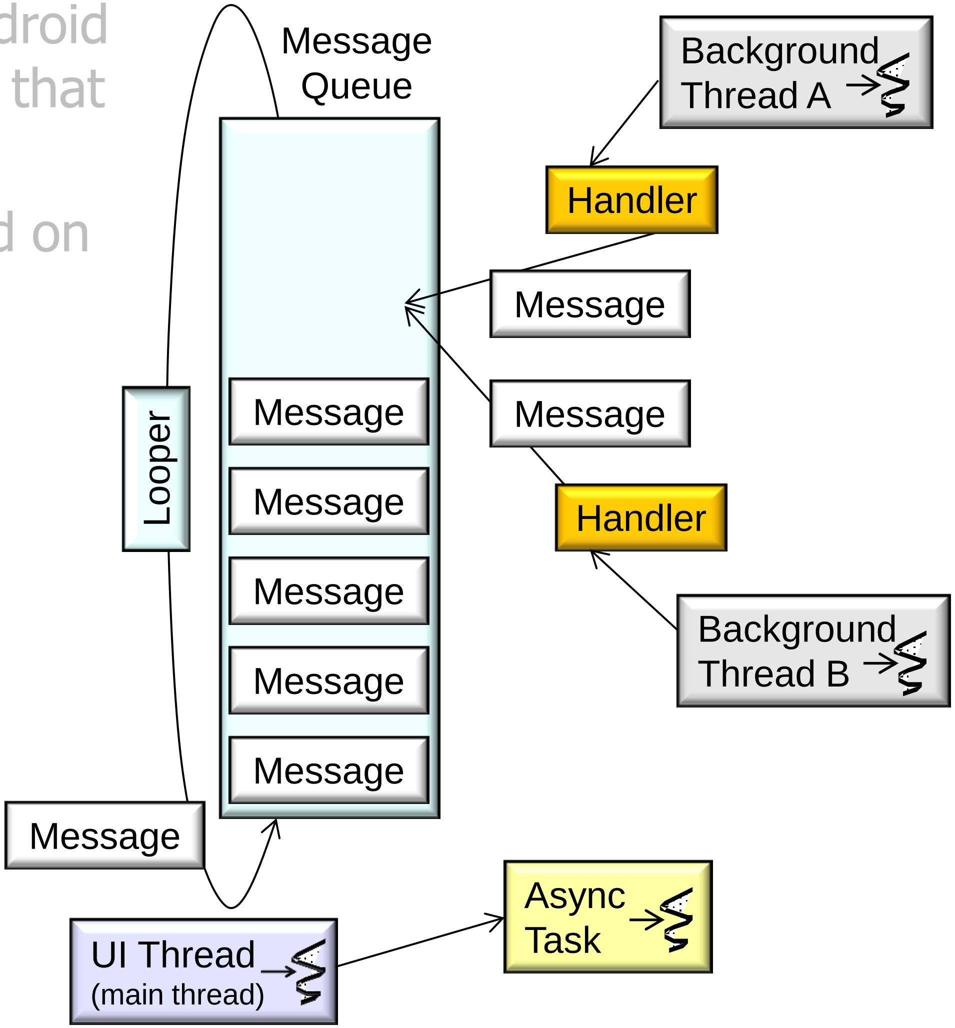
Introduction

- Explore the motivations for & challenges of concurrent software
- Understand the mechanisms that Android provides to manage multiple threads that run concurrently within a process
- Some Android mechanisms are based on standard Java threading & locking mechanisms



Introduction

- Explore the motivations for & challenges of concurrent software
- Understand the mechanisms that Android provides to manage multiple threads that run concurrently within a process
- Some Android mechanisms are based on standard Java threading & locking mechanisms
- Other mechanisms are based on Android concurrency idioms



Android Concurrency & Synchronization: Part 1

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

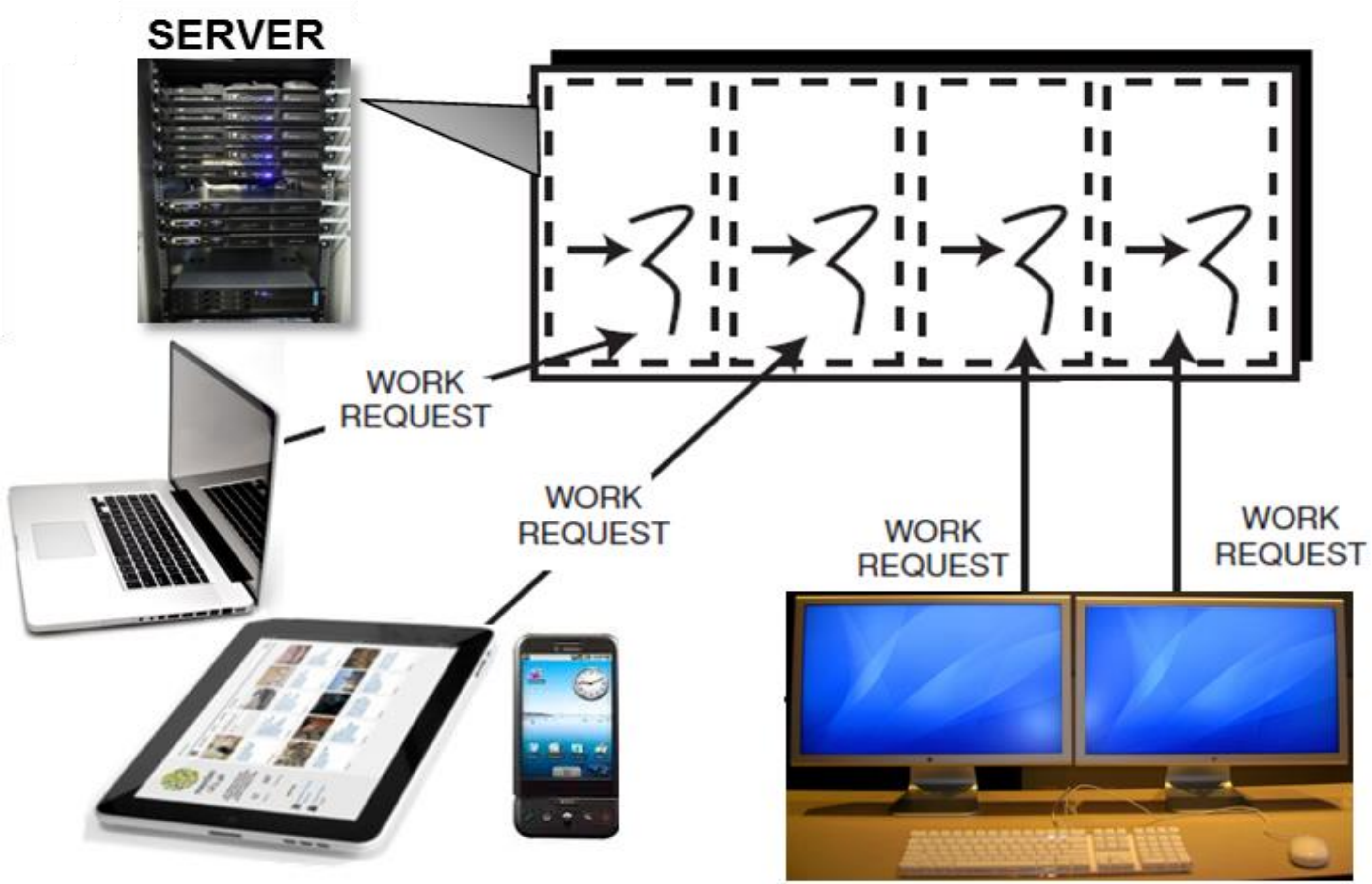
Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

CS 282 Principles of Operating Systems II

Systems Programming for Android

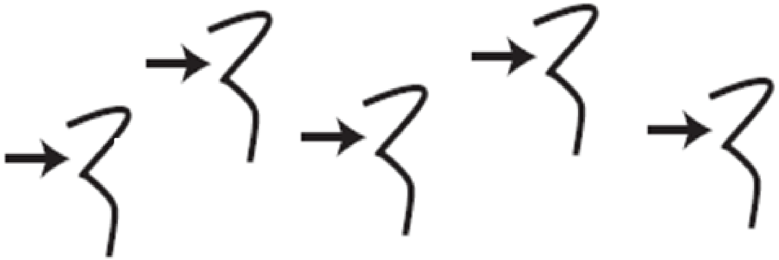
Learning Objectives in this Part of the Module

- Understand the motivations for & challenges of concurrent software



Motivations for Concurrent Software

- Leverage hardware/software advances
  - e.g., multi-core processors & multi-threaded operating systems, virtual machines, & middleware

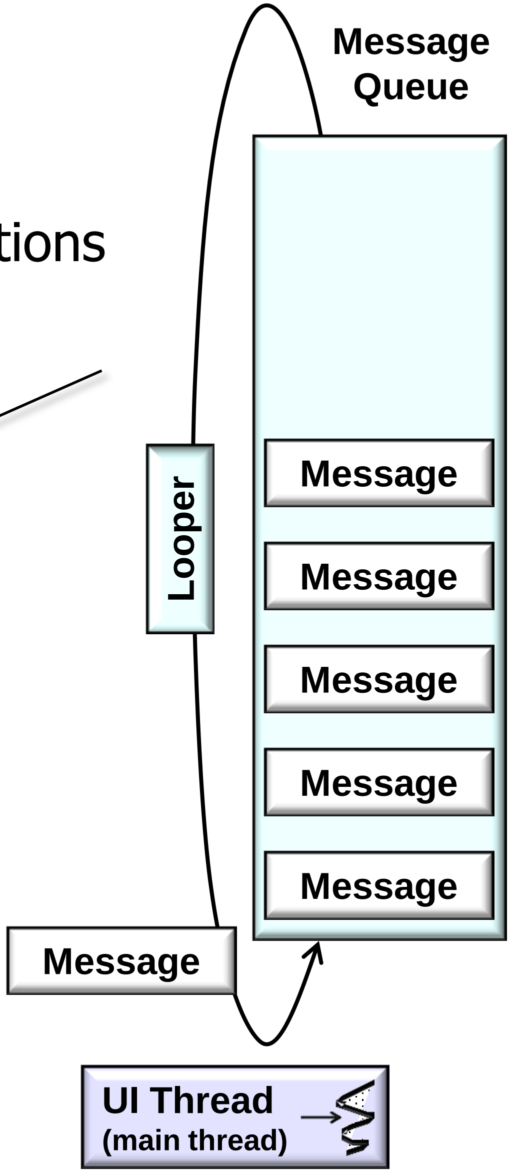


www.androidauthority.com/tag/quad-core-phones has more info

Motivations for Concurrent Software

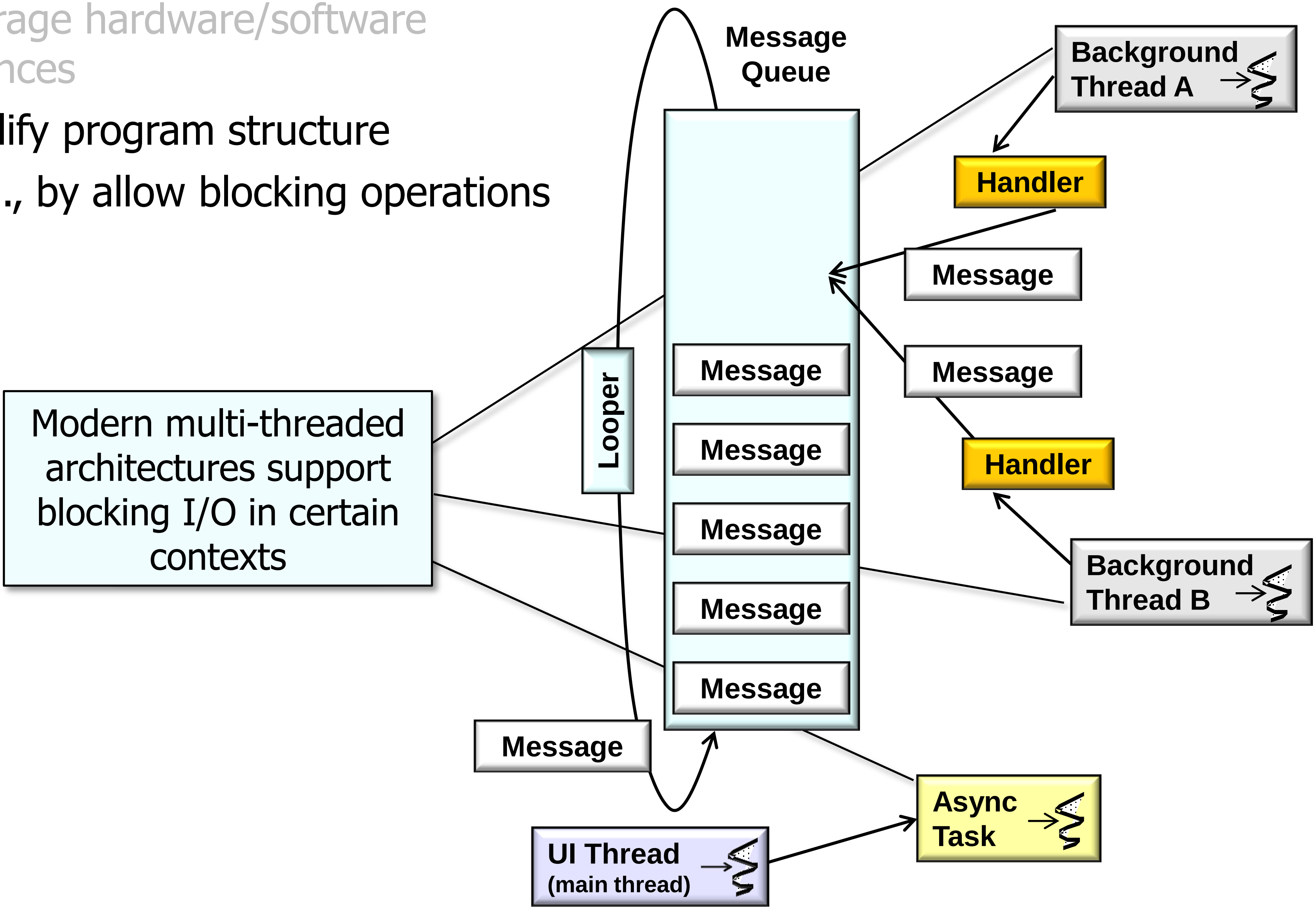
- Leverage hardware/software advances
- Simplify program structure
  - e.g., by allow blocking operations

- Classic single architectures can't perform blocking operations
- This complicates app implementations by decoupling the flow of control in time & space



Motivations for Concurrent Software

- Leverage hardware/software advances
- Simplify program structure
  - e.g., by allow blocking operations



Modern multi-threaded architectures support blocking I/O in certain contexts

Motivations for Concurrent Software

- Leverage hardware/software advances
- Simplify program structure
  - e.g., by allow blocking operations

Multi-threaded Android example

```
private Bitmap bitmap;
final ImageView iview = ...
final Button button = ...
button.setOnClickListener(new OnClickListener() {
     public void onClick(View v) {
       new Thread(new Runnable() {
           public void run() {
             bitmap = downloadImage(URI);
             iview.post(new Runnable() {
               public void run() { iview.setImageBitmap(bitmap);}
             });
           }
       }).start();
```

developer.android.com/guide/components/processes-and-threads.html#WorkerThreads

Motivations for Concurrent Software

- Leverage hardware/software advances
- Simplify program structure
  - e.g., by allow blocking operations

```
private Bitmap bitmap;
final ImageView iview = ...
final Button button = ...
button.setOnClickListener(new OnClickListener() {
     public void onClick(View v) {
       new Thread(new Runnable() {
           public void run() {
             bitmap = downloadImage(URI);
             iview.post(new Runnable() {
               public void run() { iview.setImageBitmap(bitmap);}
             });
           }
       }).start();
```

Handles button clicks

Download an image

Display bitmap in the UI thread

Start a new thread

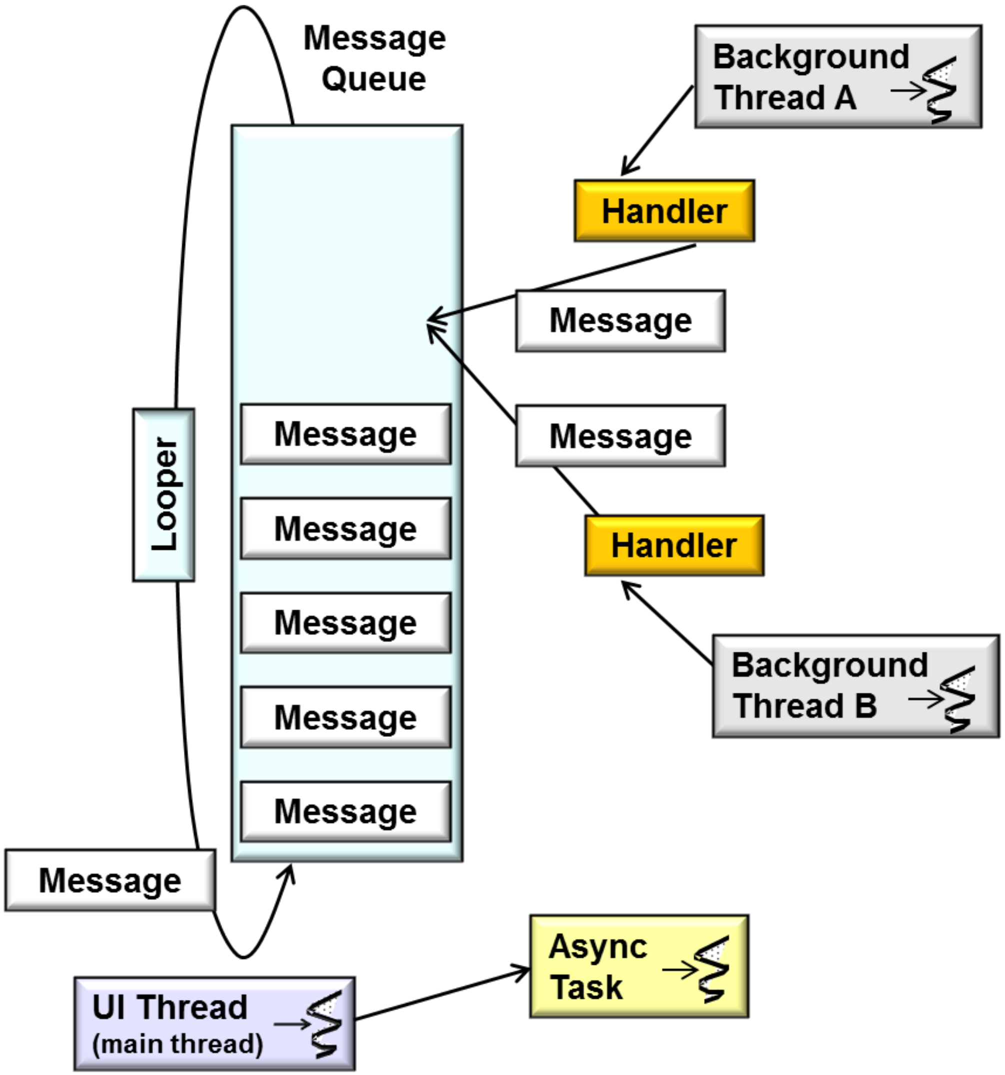
developer.android.com/guide/components/processes-and-threads.html#WorkerThreads

Motivations for Concurrent Software

- Leverage hardware/software advances
- Simplify program structure
- Increase performance
  - Parallelize computations & communications

Motivations for Concurrent Software

- Leverage hardware/software advances
- Simplify program structure
- Increase performance
- Improve response-time
  - e.g., don't starve the UI thread



Challenges for Concurrent Software

- ***Accidental Complexities***

Stem from limitations with development tools & techniques

Challenges for Concurrent Software

- ***Accidental Complexities***
  - Low-level APIs
    - Tedious, error-prone, & non-portable

See www.dre.vanderbilt.edu/~schmidt/PDF/BC-schmidt.pdf for more info

Challenges for Concurrent Software

- ***Accidental Complexities***
  - Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");
```

Cast from void *

Not portable to non-POSIX platforms

```
  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

Pointer-to-function

Cast to void *

“Quasi-typed” thread handle

Challenges for Concurrent Software

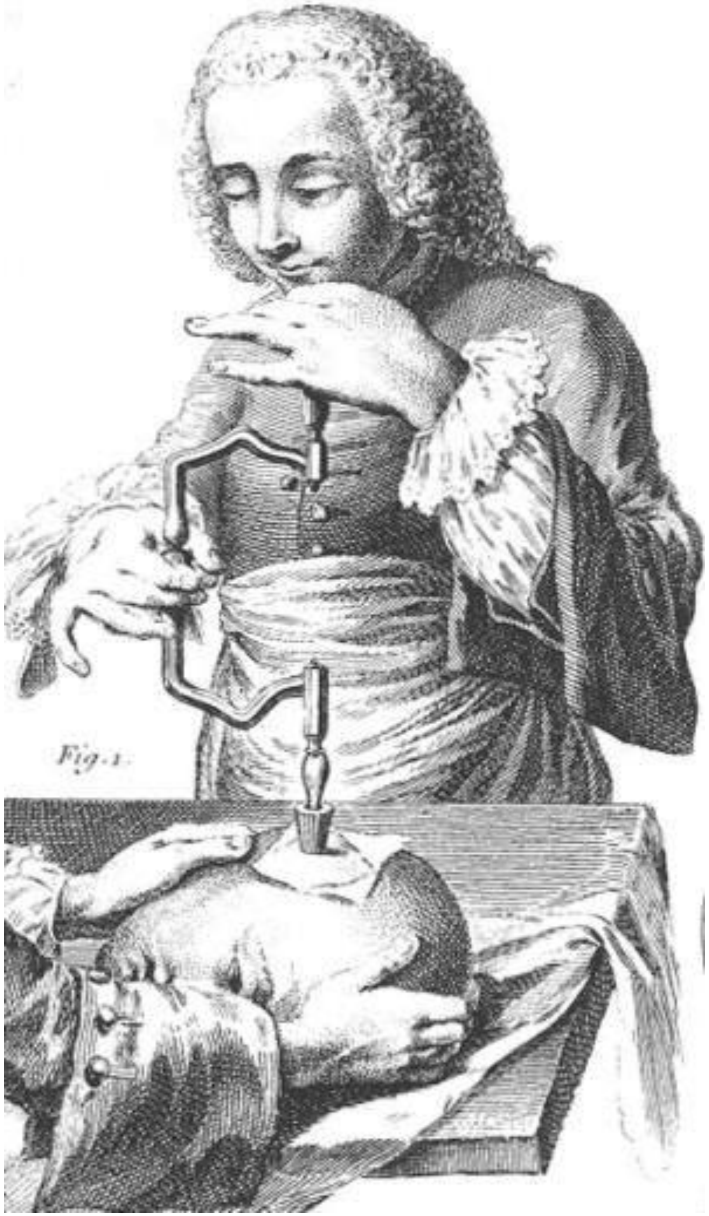
- ***Accidental Complexities***
  - Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");


  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

Other C threading APIs have similar accidental complexities

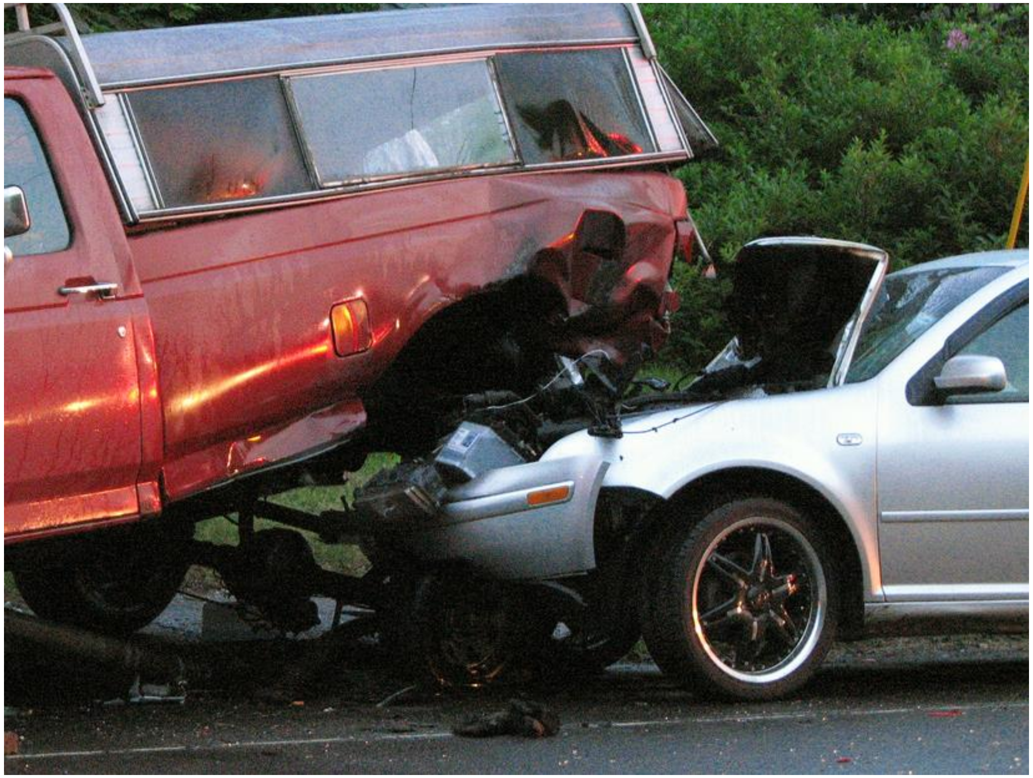
Challenges for Concurrent Software

- ***Accidental Complexities***
  - Low-level APIs
  - Limited debugging tools



Challenges for Concurrent Software

- ***Accidental Complexities***
  - Low-level APIs
  - Limited debugging tools



See www.dre.vanderbilt.edu/~schmidt/PDF/DSIS.pdf & www.fluid.cs.cmu.edu

Challenges for Concurrent Software

- *Accidental Complexities*
- ***Inherent Complexities***

Stem from fundamental domain challenges

Challenges for Concurrent Software

- *Accidental Complexities*
- ***Inherent Complexities***
  - Synchronization

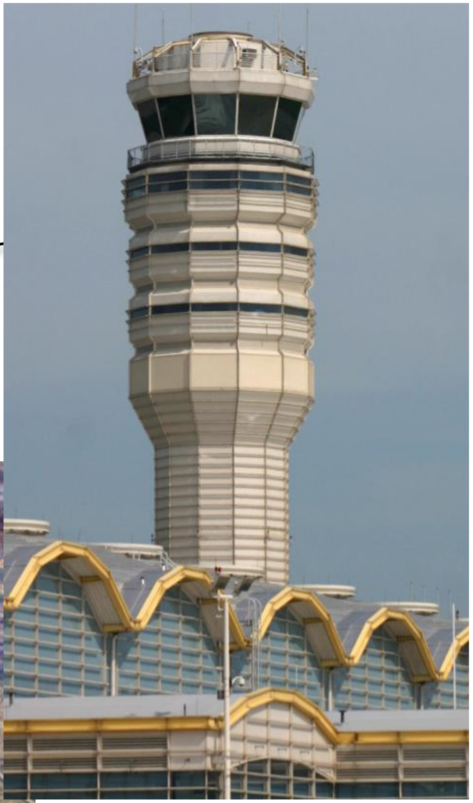
Synchronization is the application of mechanisms to ensure that two concurrently-executing threads do not execute specific portions of a program at the same time

en.wikipedia.org/wiki/Synchronization_(computer_science) has more info

Challenges for Concurrent Software

- *Accidental Complexities*
- ***Inherent Complexities***
  - Synchronization
  - Scheduling

Scheduling is the method by which threads, processes, or data flows are given access to system resources

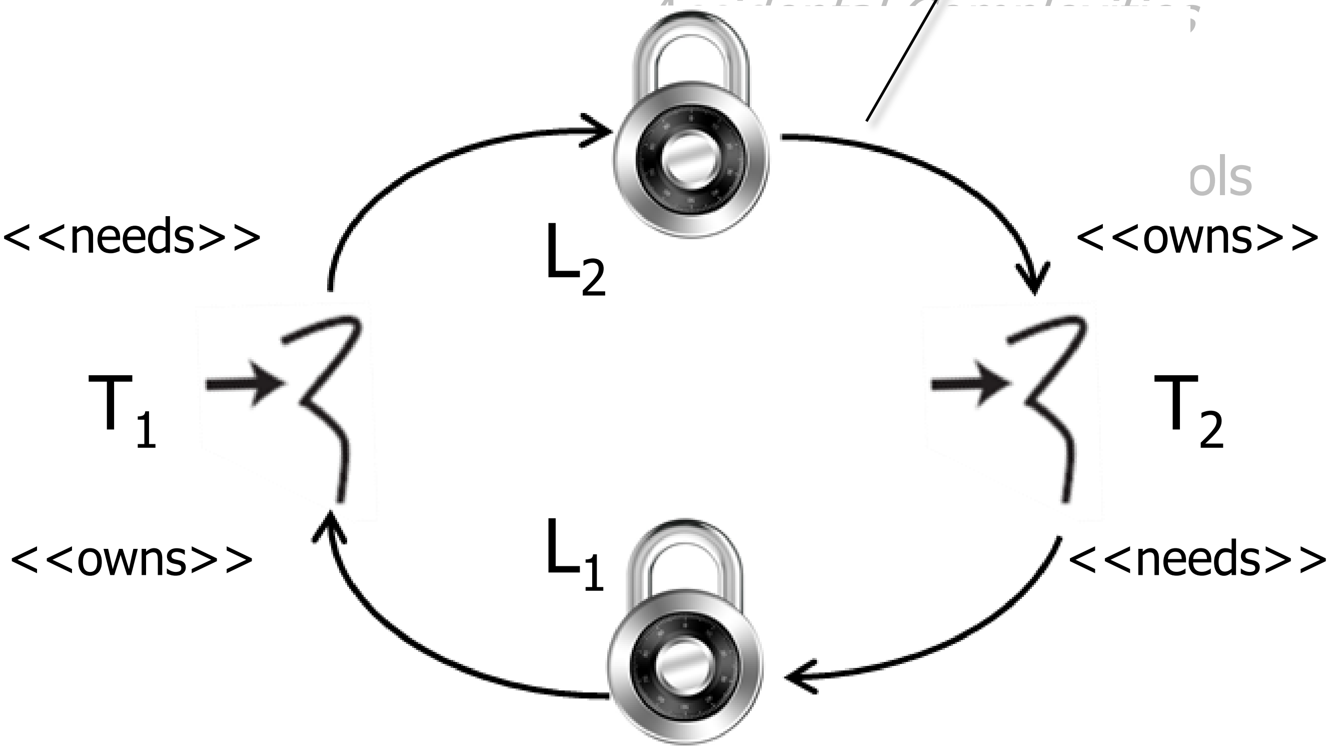


en.wikipedia.org/wiki/Scheduling_(computing) has more info

Challenges for Concurrent Software

- *Accidental Complexities*
- ***Inherent Complexities***
    - Synchronization
    - Scheduling
    - Deadlock

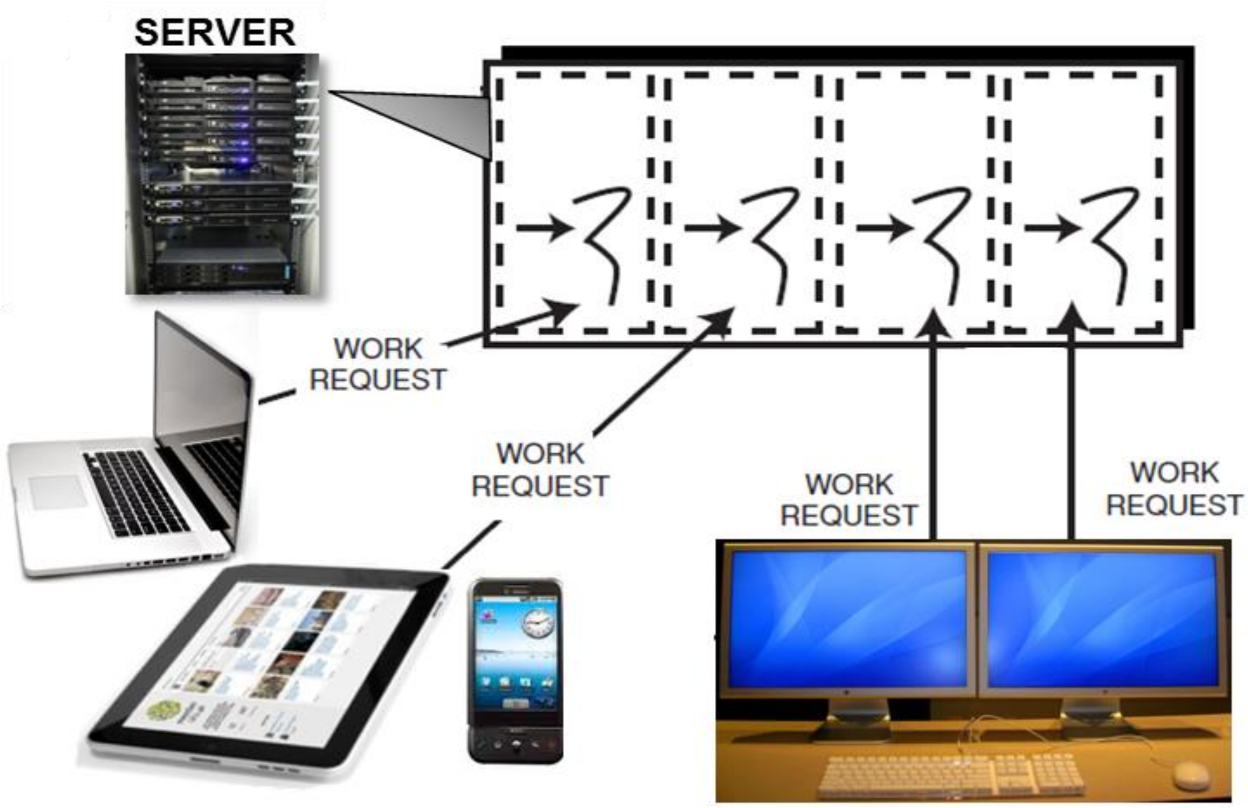
A ***deadlock*** *is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does*



See en.wikipedia.org/wiki/Deadlock for more info

Summary

- Concurrent software helps
  - Leverage advances in hardware technology
  - Meet the quality & performance needs of apps & services



Summary

- Concurrent software helps
  - Leverage advances in hardware technology
  - Meet the quality & performance needs of apps & services
- Successful concurrent software solutions must address key *accidental* & *inherent* complexities arising from
  - Limitations with development tools/techniques

Summary

- Concurrent software helps
  - Leverage advances in hardware technology
  - Meet the quality & performance needs of apps & services
- Successful concurrent software solutions must address key *accidental* & *inherent* complexities arising from
  - Limitations with development tools/techniques
  - Fundamental domain challenges

Android Concurrency & Synchronization: Part 2

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

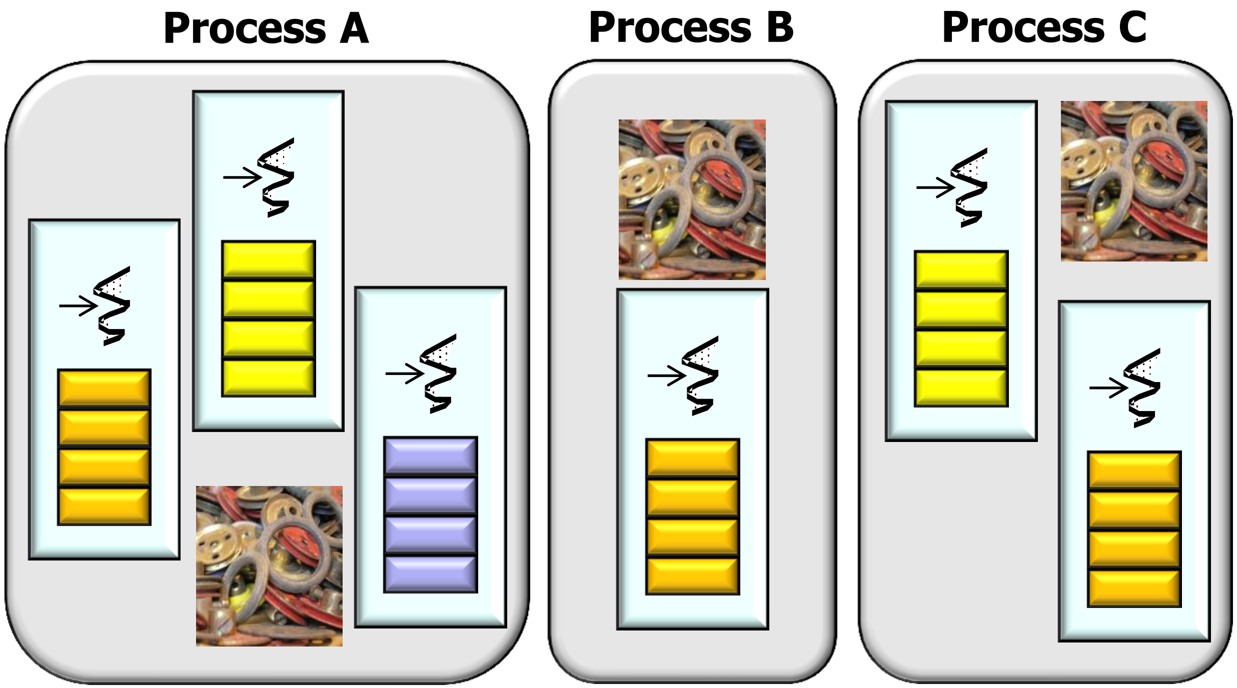
Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

CS 282 Principles of Operating Systems II

Systems Programming for Android

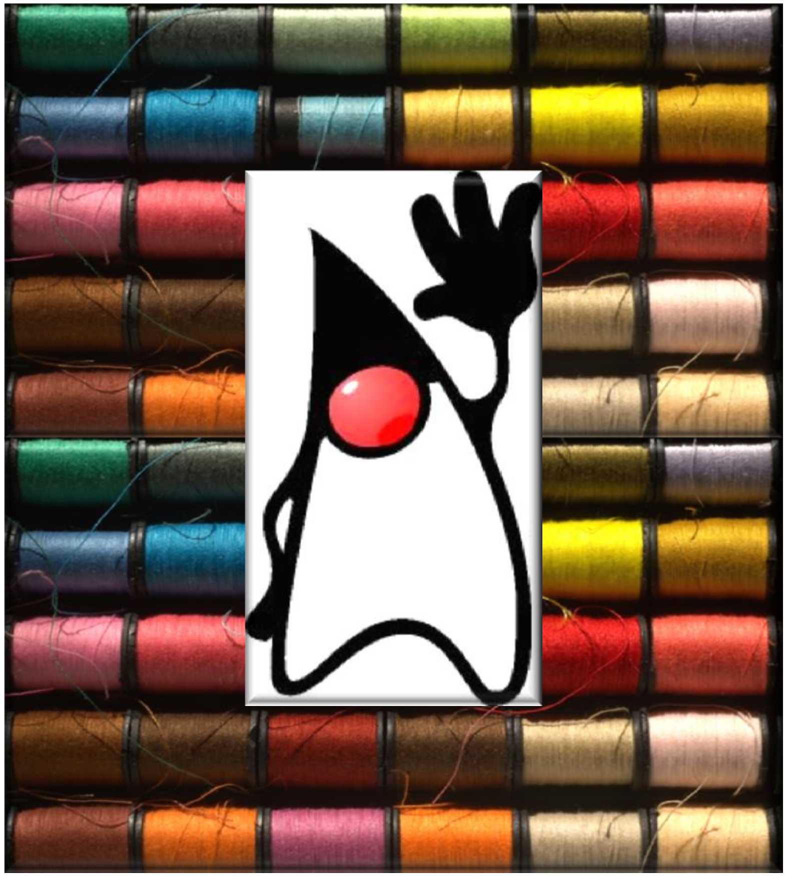
Learning Objectives in this Part of the Module

- Understand how to program Java mechanisms available in Android to implement *concurrent* apps that process requests simultaneously via multithreading



Overview of Java Threads in Android

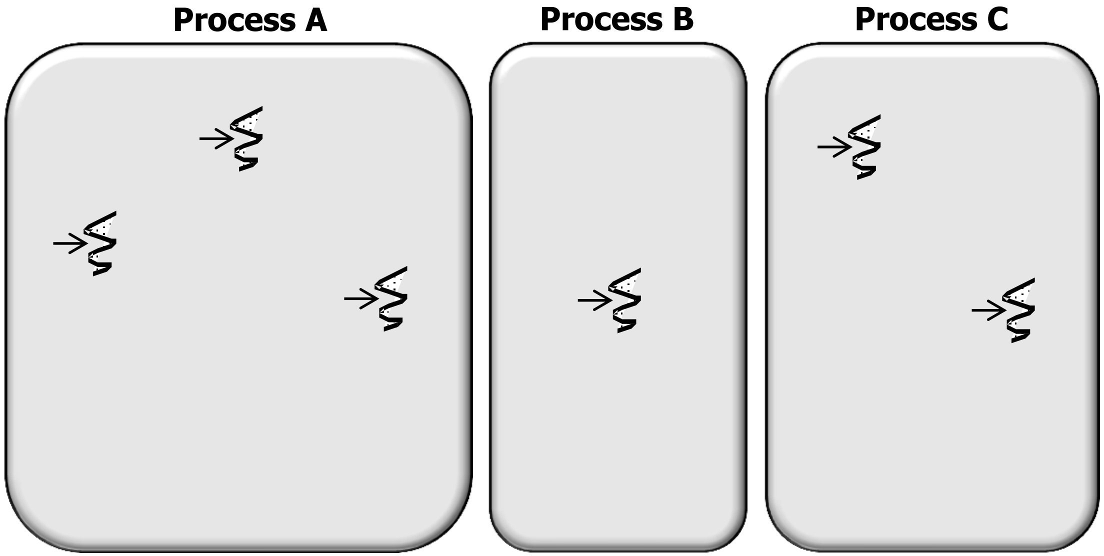
- Android implements many standard Java concurrency & synchronization classes

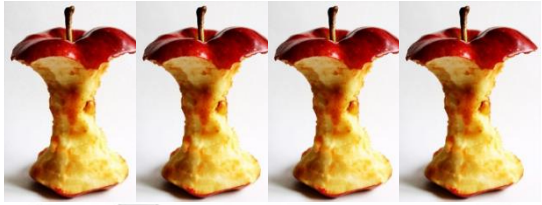


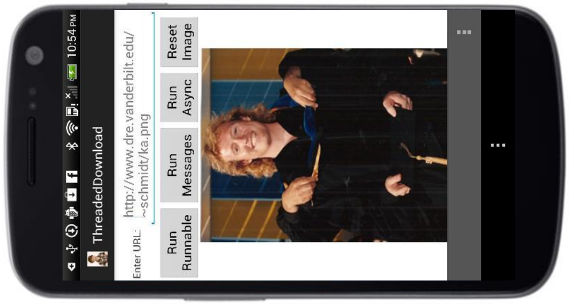
See docs.oracle.com/javase/tutorial/essential/concurrency

Overview of Java Threads in Android

- Android implements many standard Java concurrency & synchronization classes
- **Conceptual view**
  - Concurrent computations running in a (Linux) process that can communicate with each other via shared memory or message passing

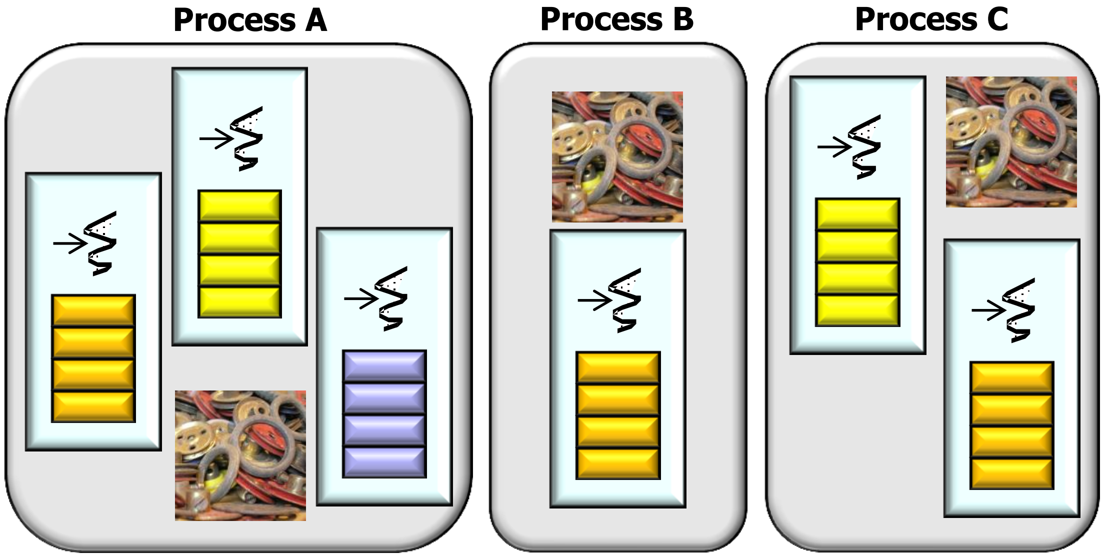


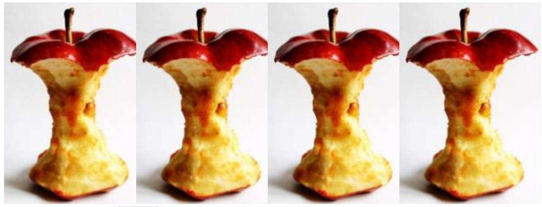


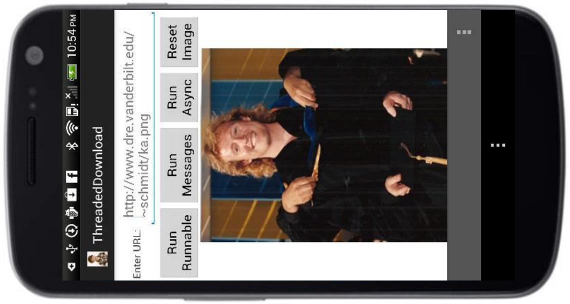


Overview of Java Threads in Android

- Android implements many standard Java concurrency & synchronization classes
- **Conceptual view**
- **Implementation view**
  - Each Java thread has a program counter & a stack (unique)
  - The heap & static areas are shared across threads (common)







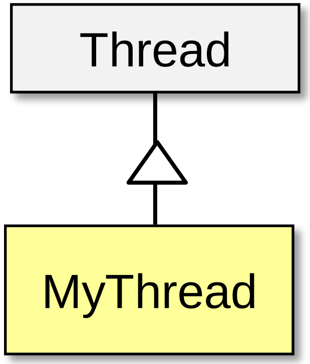
Enter URL: http://www.dre.vanderbilt.edu/~schmidt/ka.png

See developer.android.com/guide/components/processes-and-threads.html

Using Java Threads in Android

- All threads must be given some code to run by either
  - Extending the Thread class

```
public class MyThread
               extends Thread {
   public void run() {
      //code to run goes here
   }
}

MyThread myt = new MyThread();
myt.start();
```

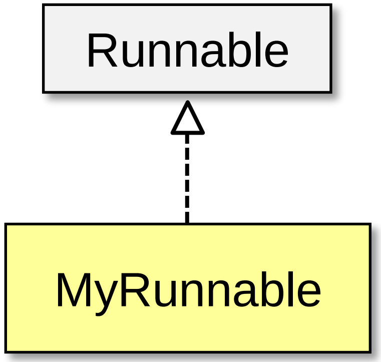


Starting a thread using a named class (or inner class)

Using Java Threads in Android

- All threads must be given some code to run by either
  - Extending the Thread class
  - Implementing the Runnable interface

```
public interface Runnable {
   public void run();
}

public class MyRunnable
             implements Runnable {
      public void run() {
        //code to run goes here
      }
   }

MyRunnable myr = new MyRunnable();
new Thread(myr).start();
```

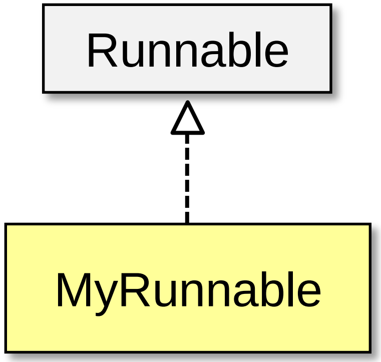


Starting a thread using a named implementation of Runnable

Using Java Threads in Android

- All threads must be given some code to run by either
  - Extending the Thread class
  - Implementing the Runnable interface

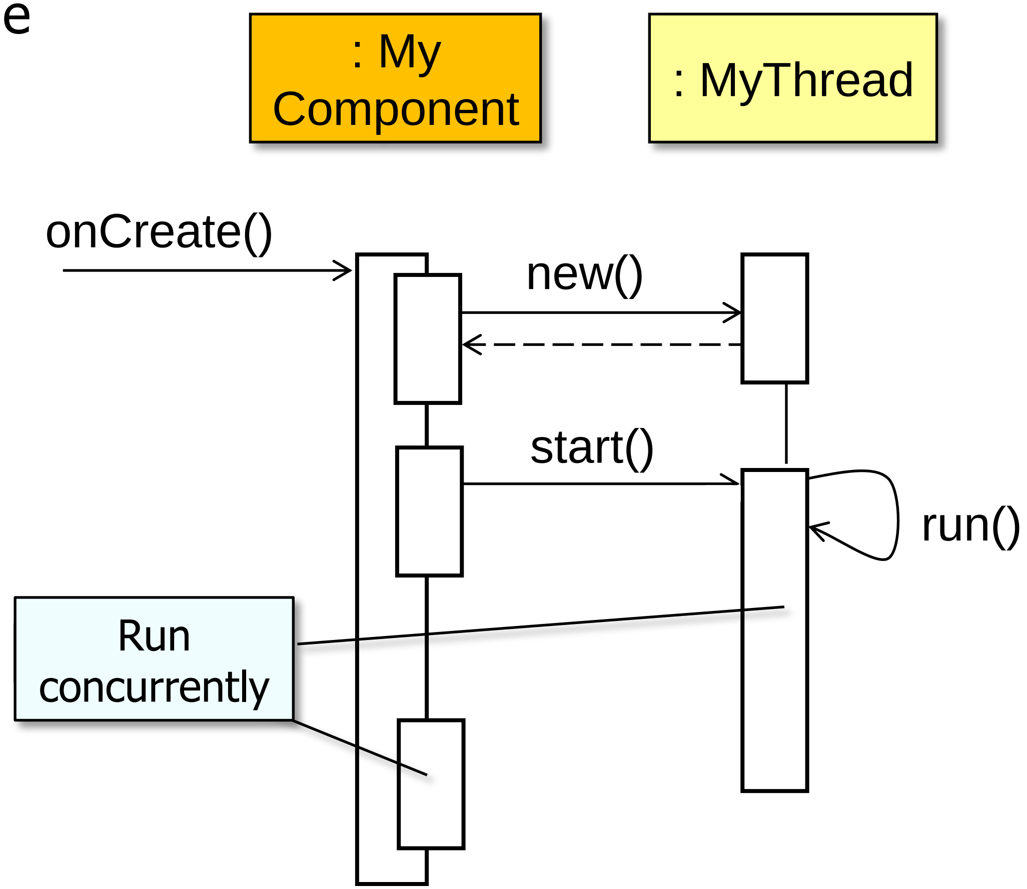
```
public interface Runnable {
   public void run();
}

new Thread(new Runnable() {
  public void run(){
    //code to run goes here
  }
}).start();
```



Starting a thread using an anonymous class (or inner class) as the Runnable

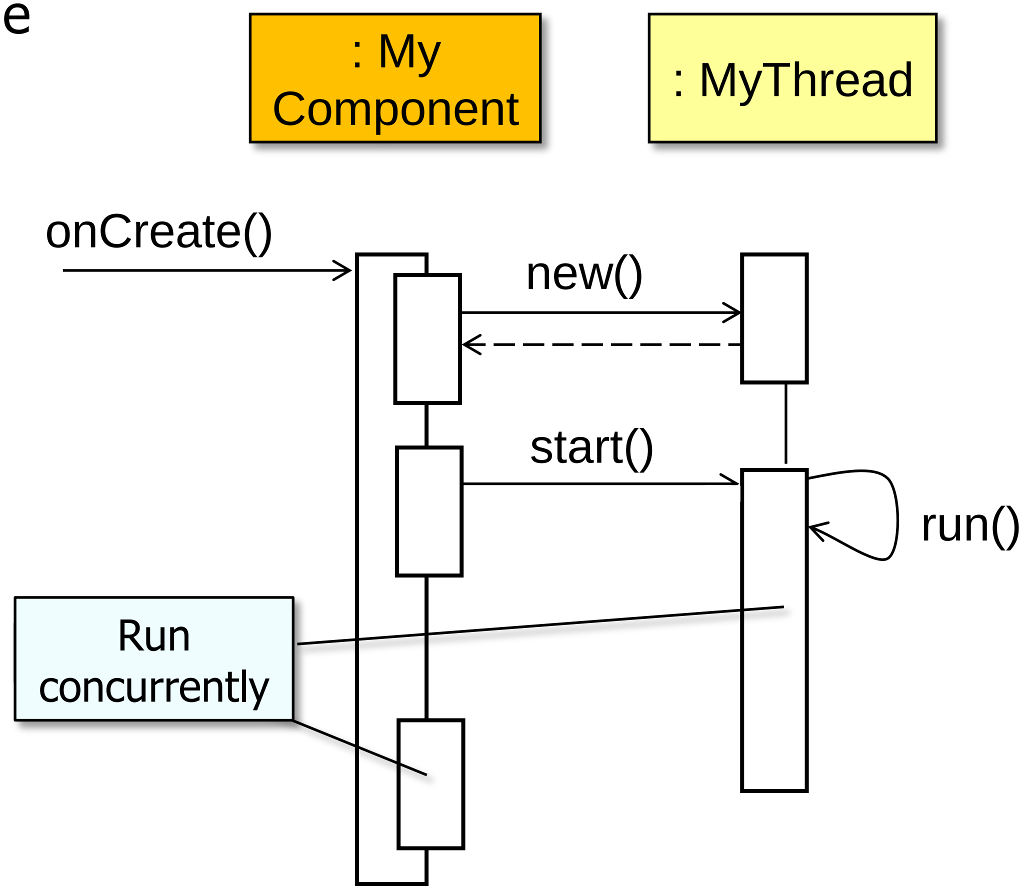
Using Java Threads in Android

- All threads must be given some code to run
- Android calls the Thread/Runnable run() method after a new thread starts up



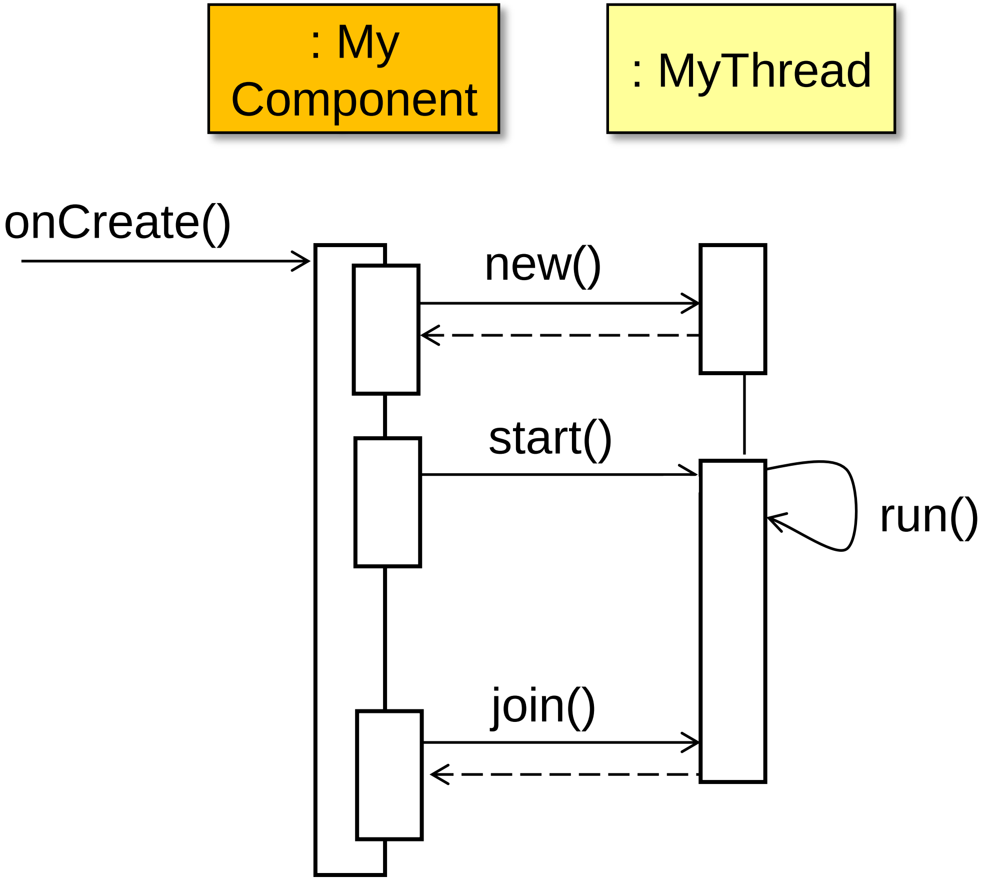
Using Java Threads in Android

- All threads must be given some code to run
- Android calls the Thread/Runnable run() method after a new thread starts up
  - You can run any code in a thread, but it must be inside of a run() method or called from a run() method



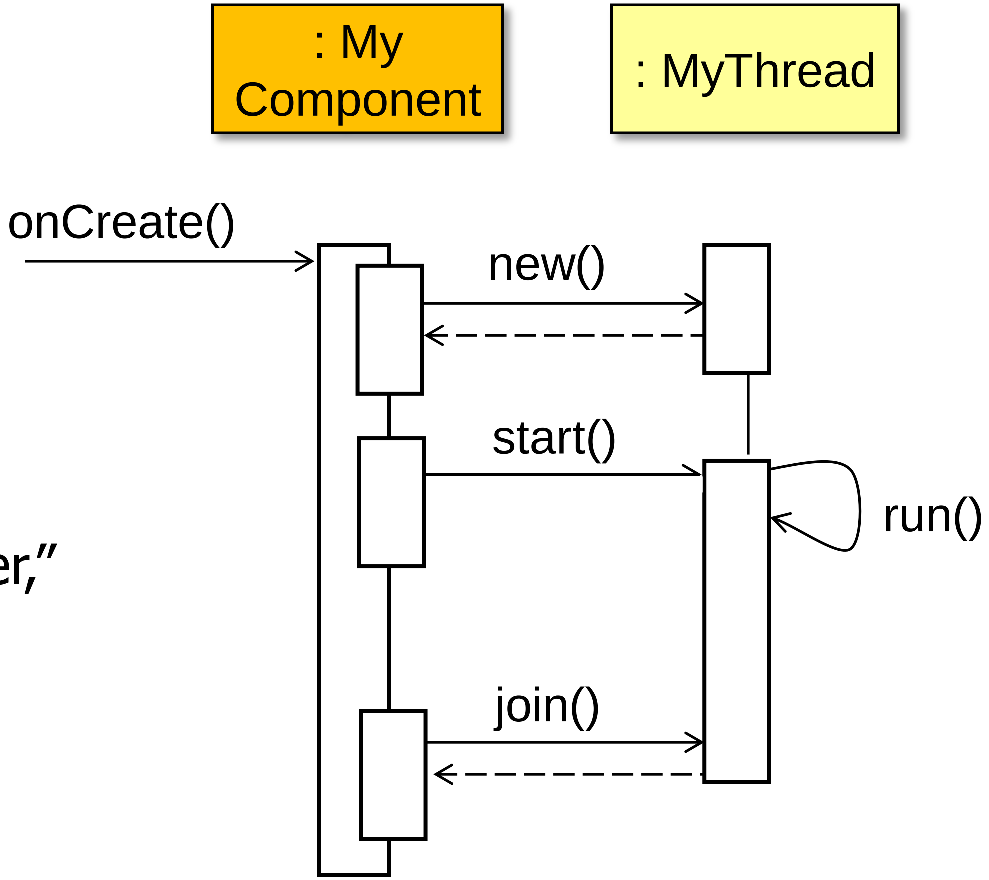
Using Java Threads in Android

- All threads must be given some code to run
- Android calls the Thread/Runnable run() method after a new thread starts up
- The thread can be active as long as the run() method hasn't returned
  - Naturally, the Android scheduler can suspend/resume threads



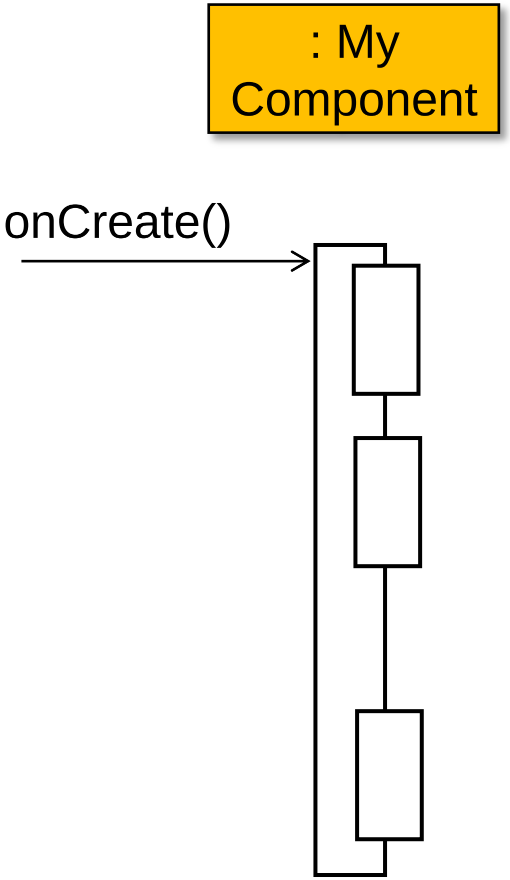
Using Java Threads in Android

- All threads must be given some code to run
- Android calls the Thread/Runnable run() method after a new thread starts up
- The thread can be active as long as the run() method hasn't returned
  - Naturally, the Android scheduler can suspend/resume threads
  - If you want thread to run "forever," you need to have a while(true) statement in that run() method



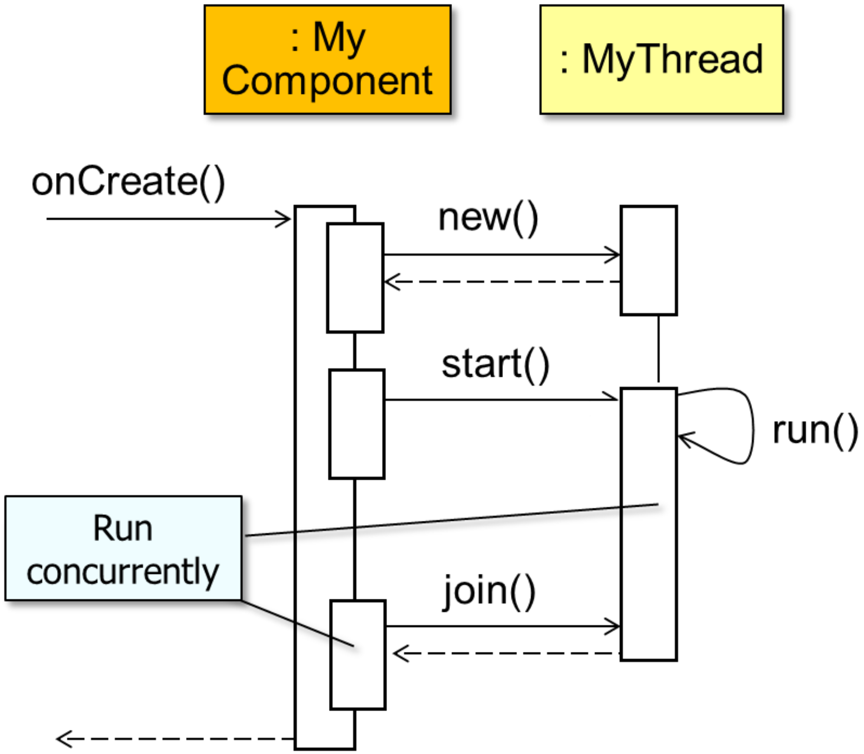
Using Java Threads in Android

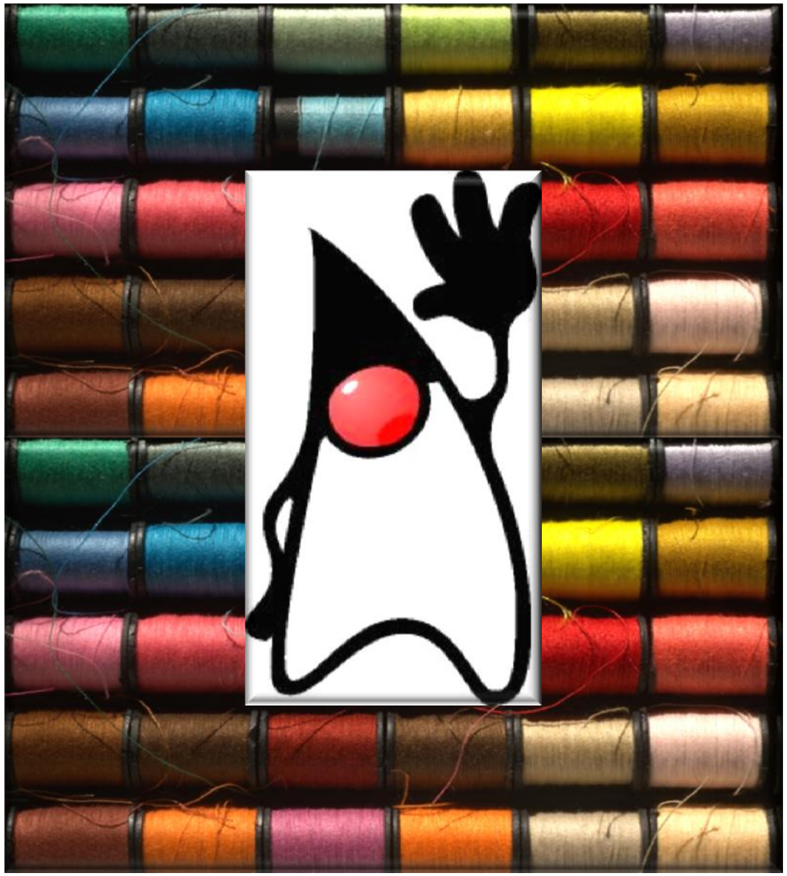
- All threads must be given some code to run
- Android calls the Thread/Runnable run() method after a new thread starts up
- The thread can be active as long as the run() method hasn't returned
- When run() returns the thread is no longer active



Summary

- Some concurrency mechanisms provided by Android are based on standard Java threading classes





Android Concurrency & Synchronization: Part 3

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

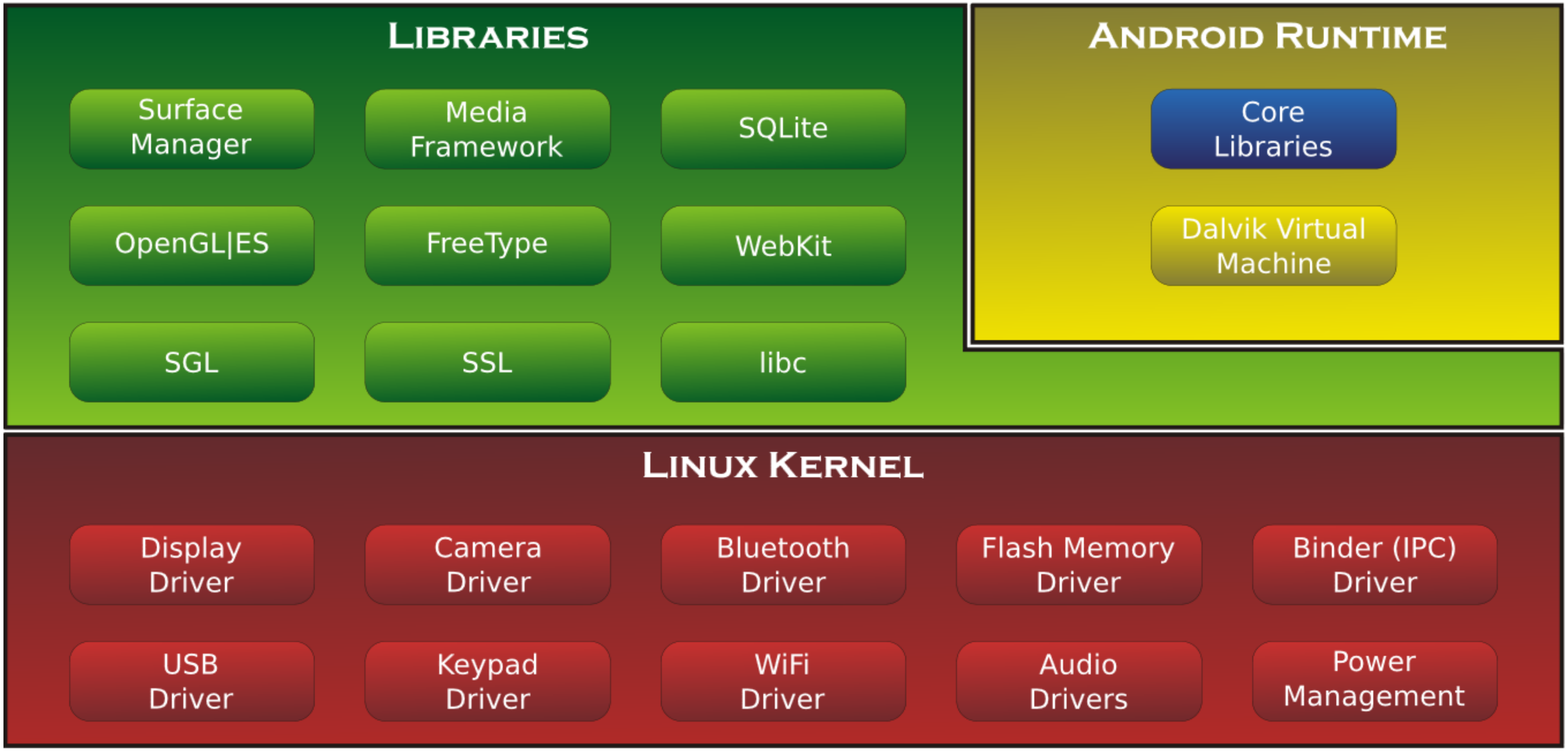
Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

CS 282 Principles of Operating Systems II

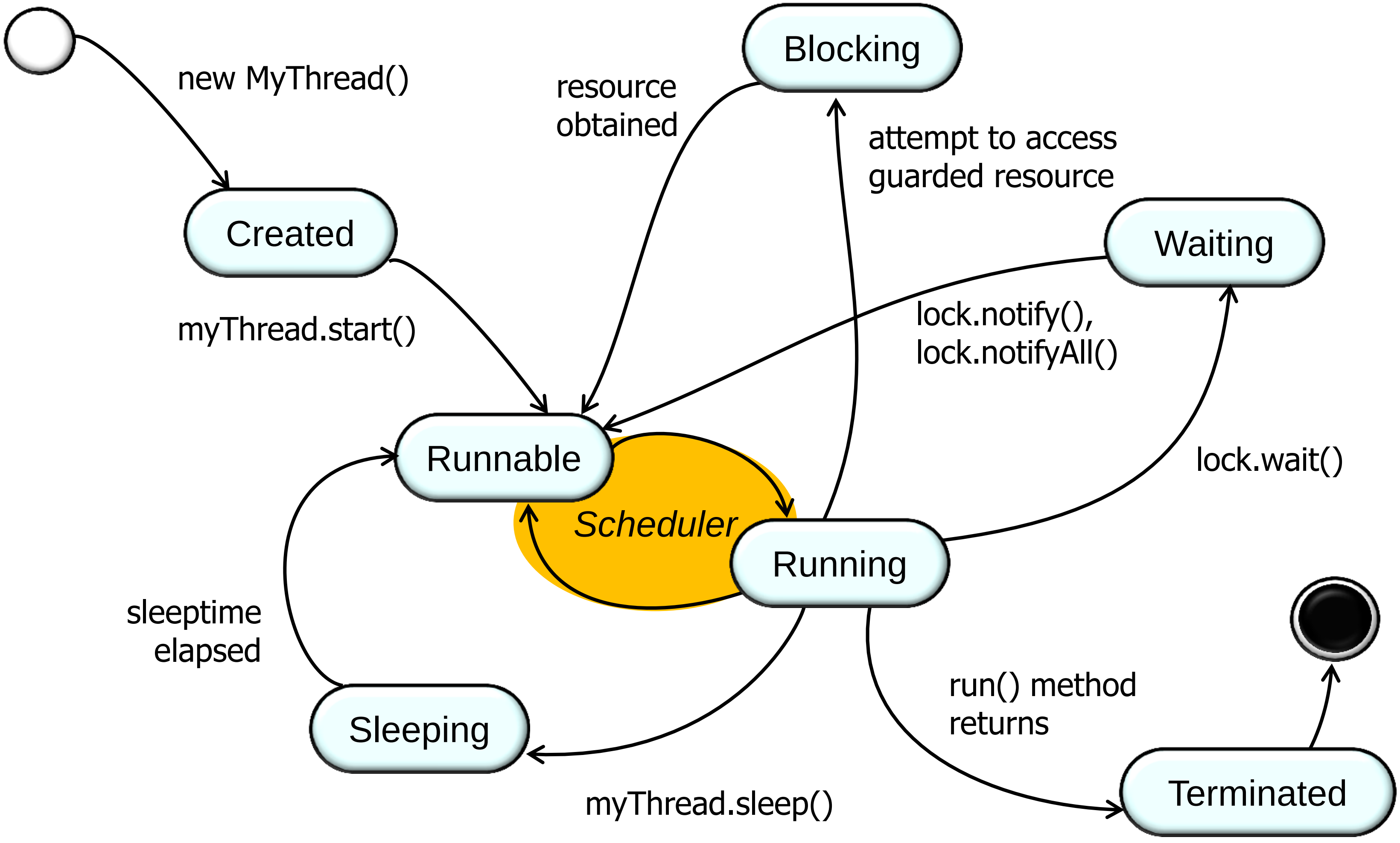
Systems Programming for Android

Learning Objectives in this Part of the Module

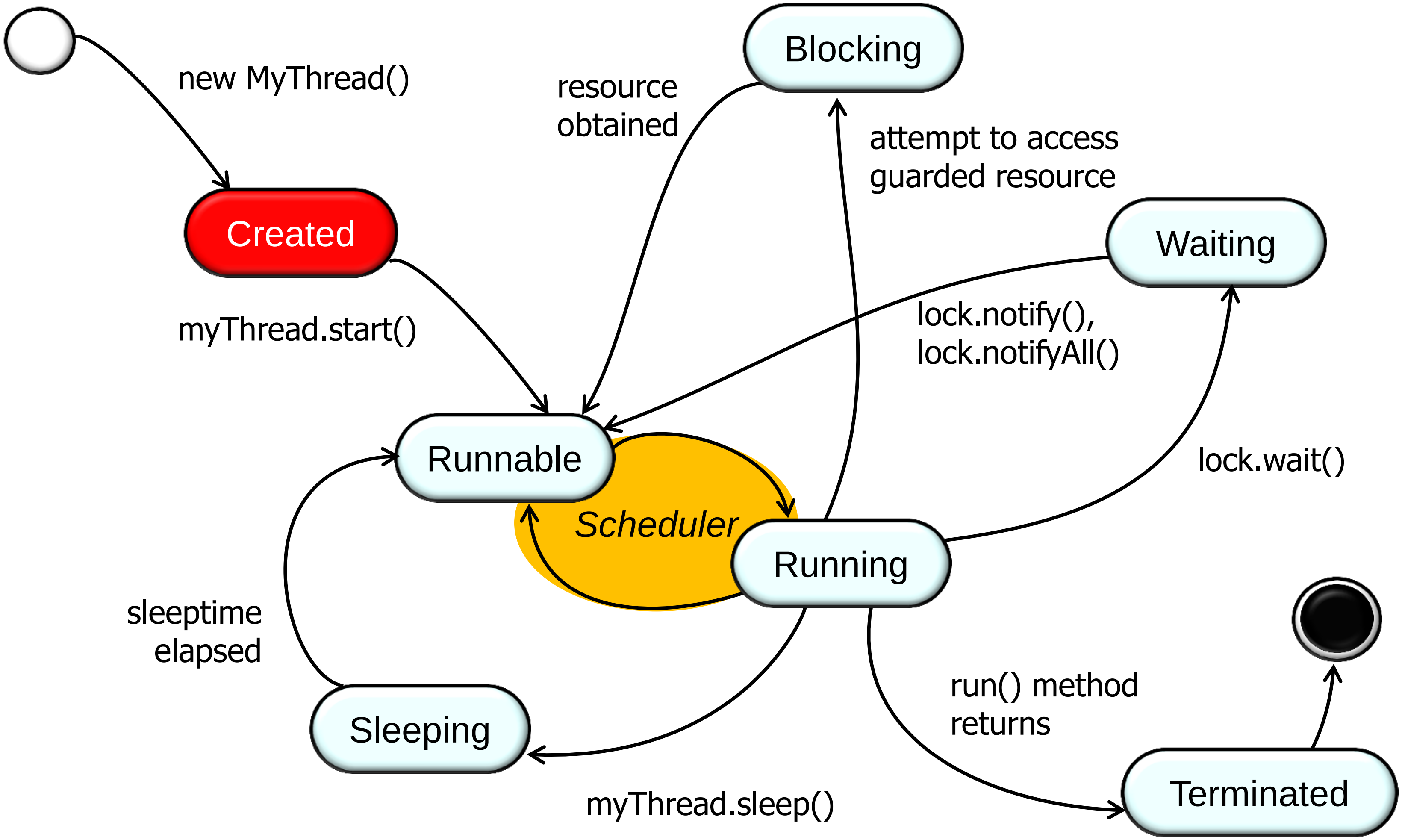
- Understand how the Java concurrency mechanisms available in Android are implemented



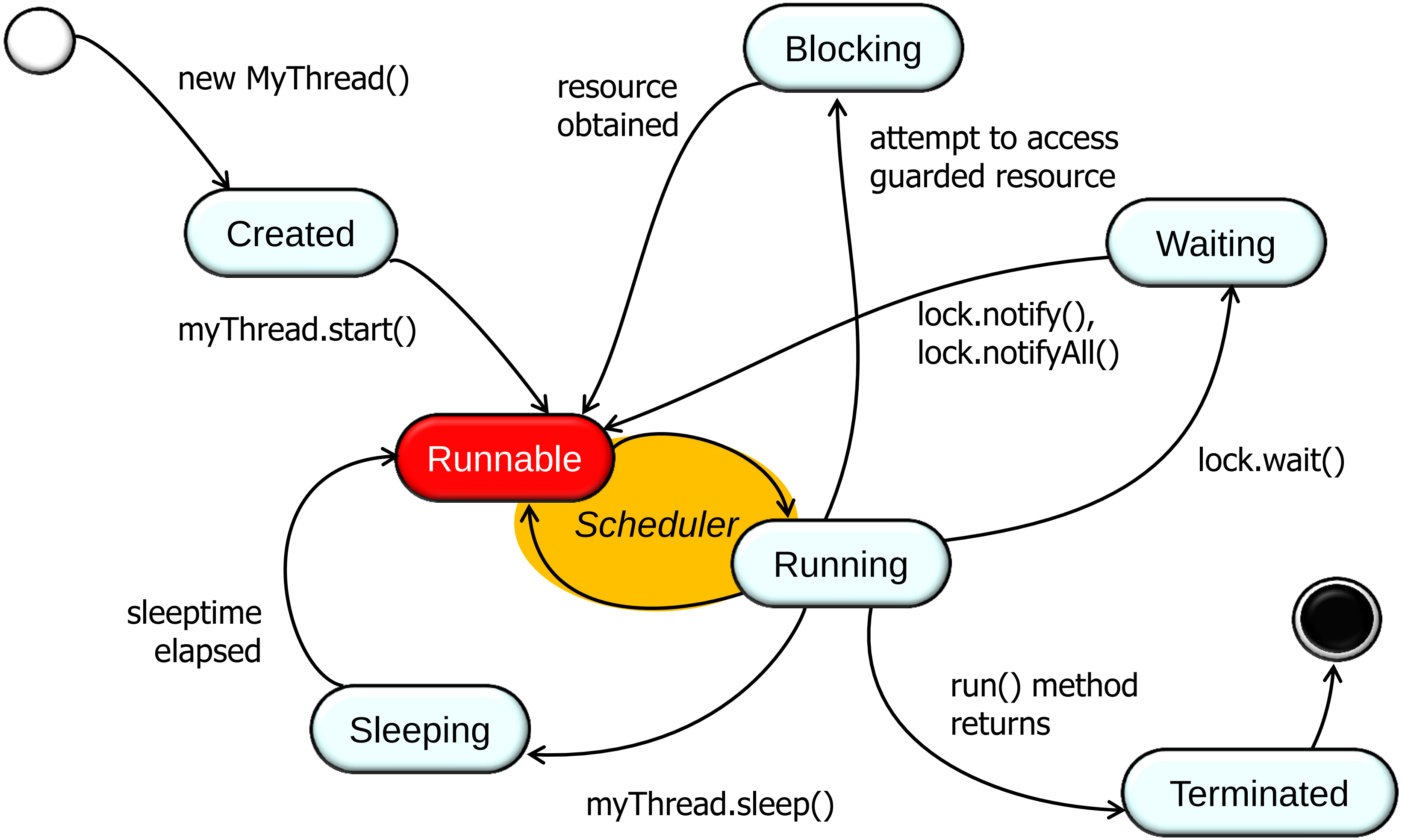
State Machine for Java Threads in Android



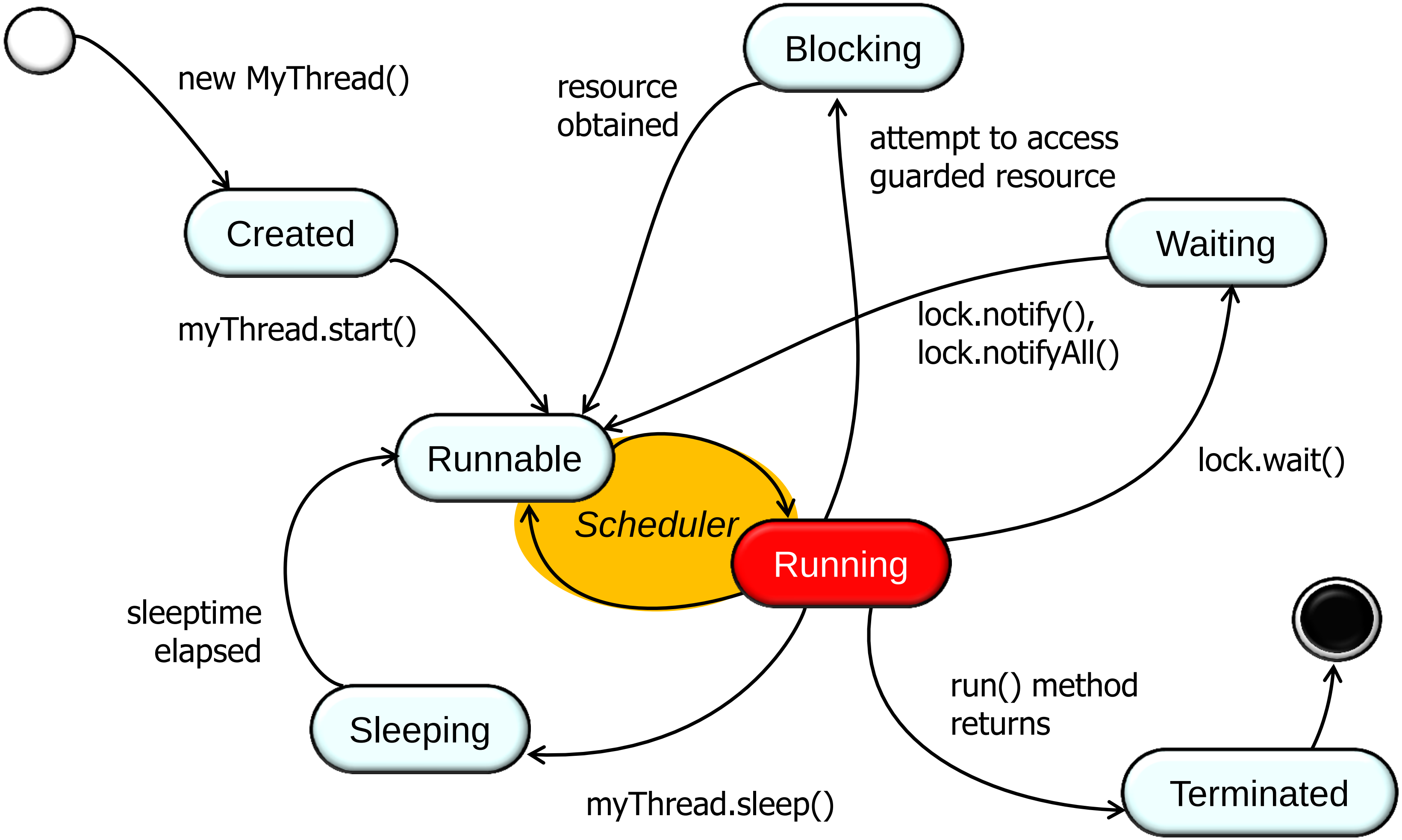
State Machine for Java Threads in Android



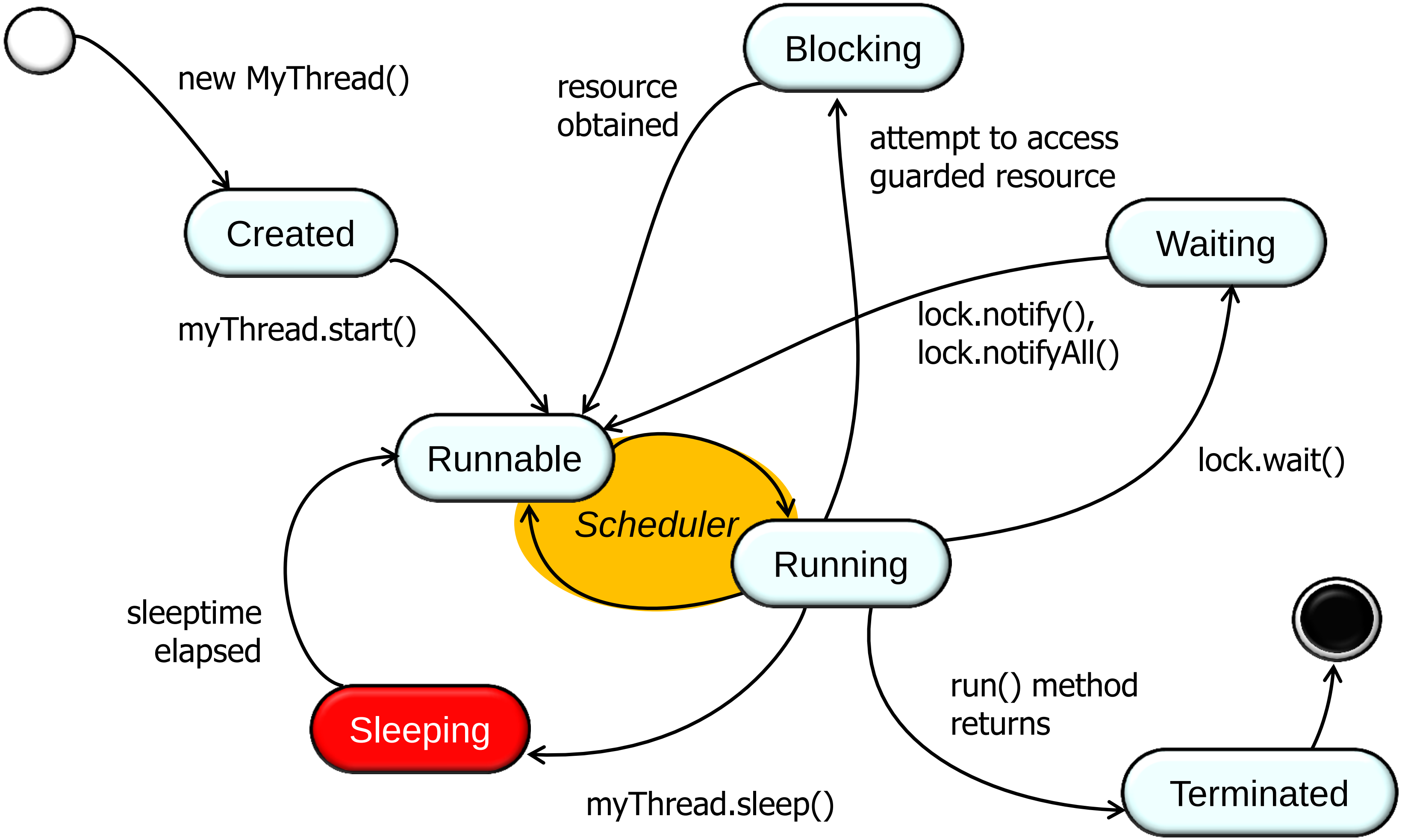
State Machine for Java Threads in Android



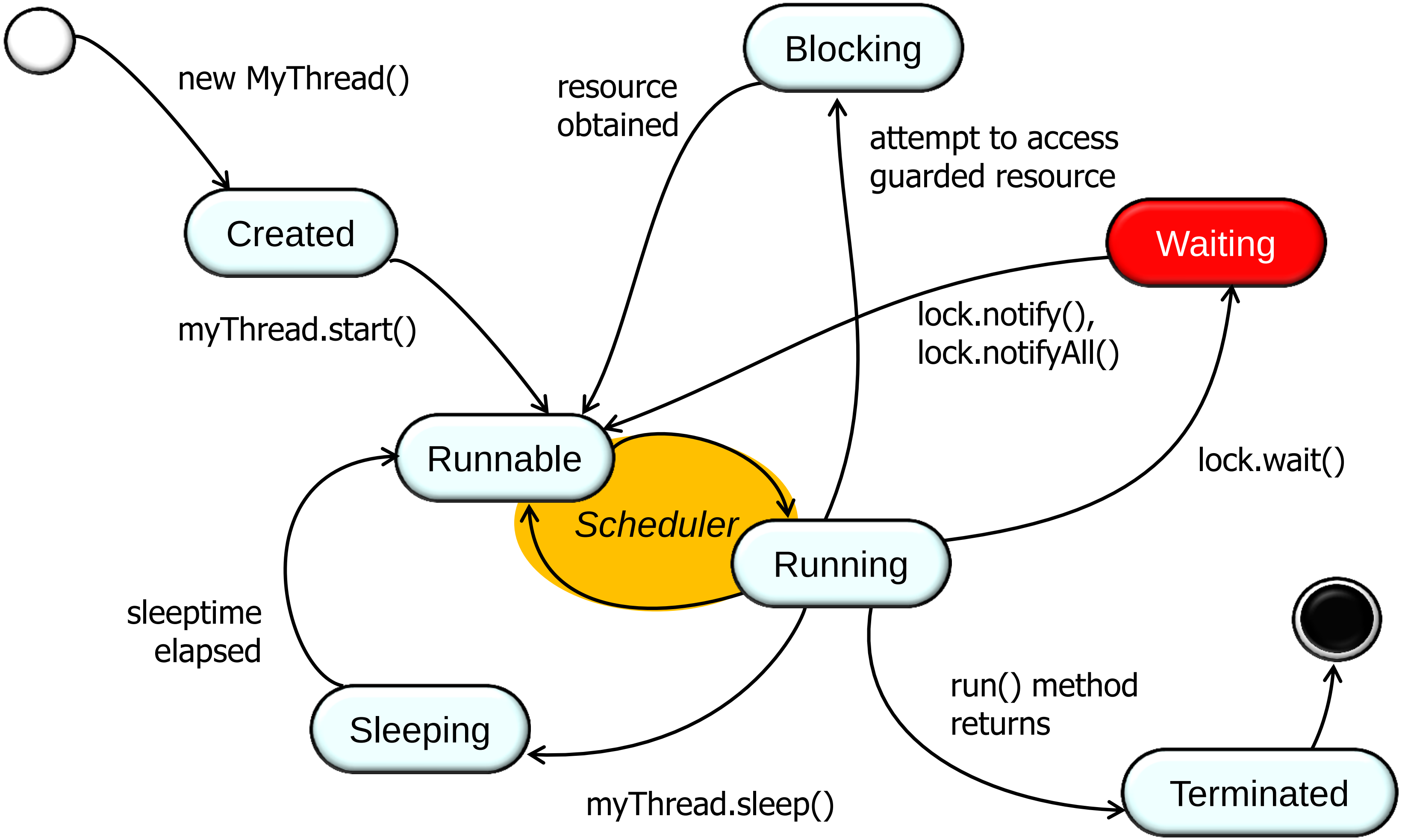
State Machine for Java Threads in Android



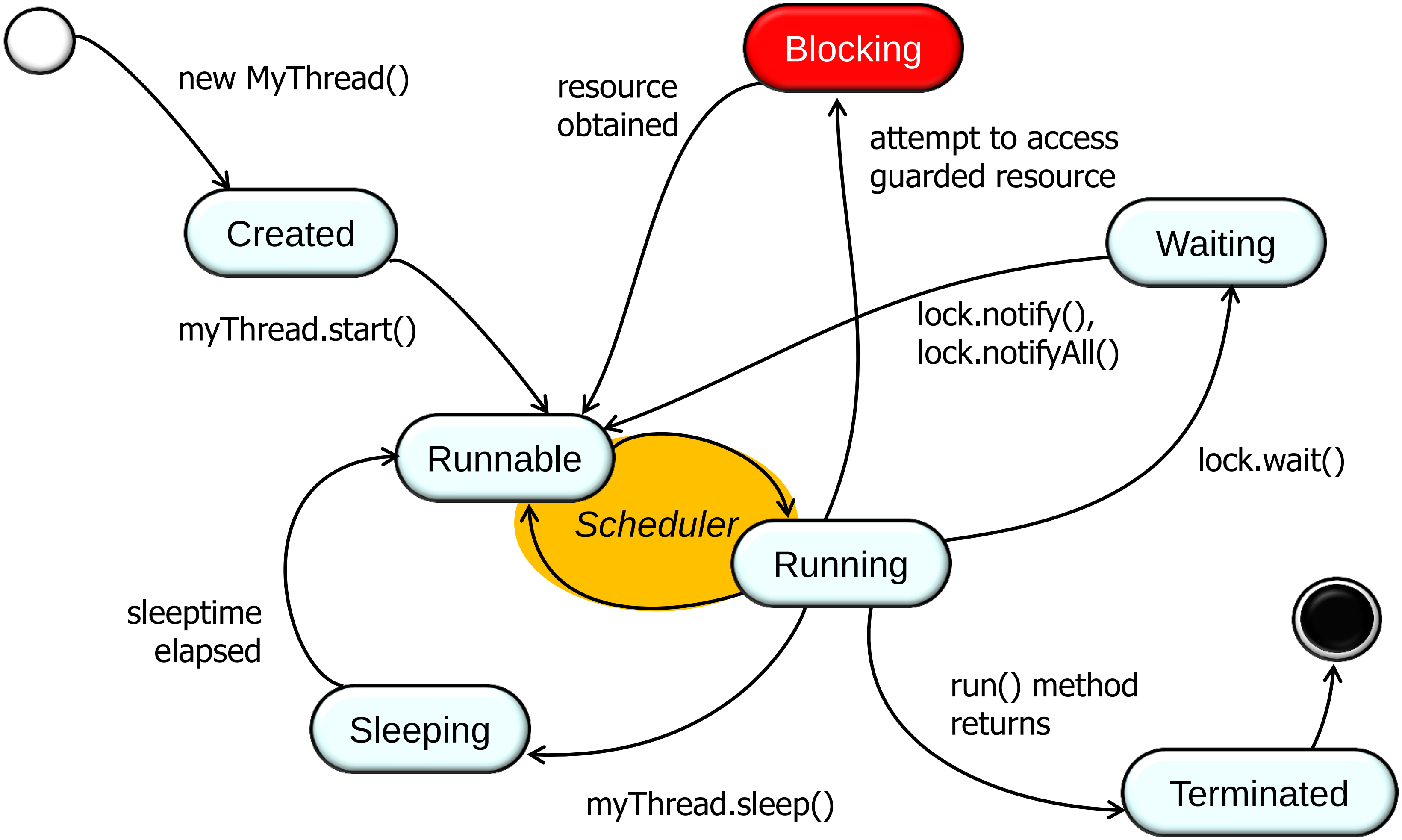
State Machine for Java Threads in Android



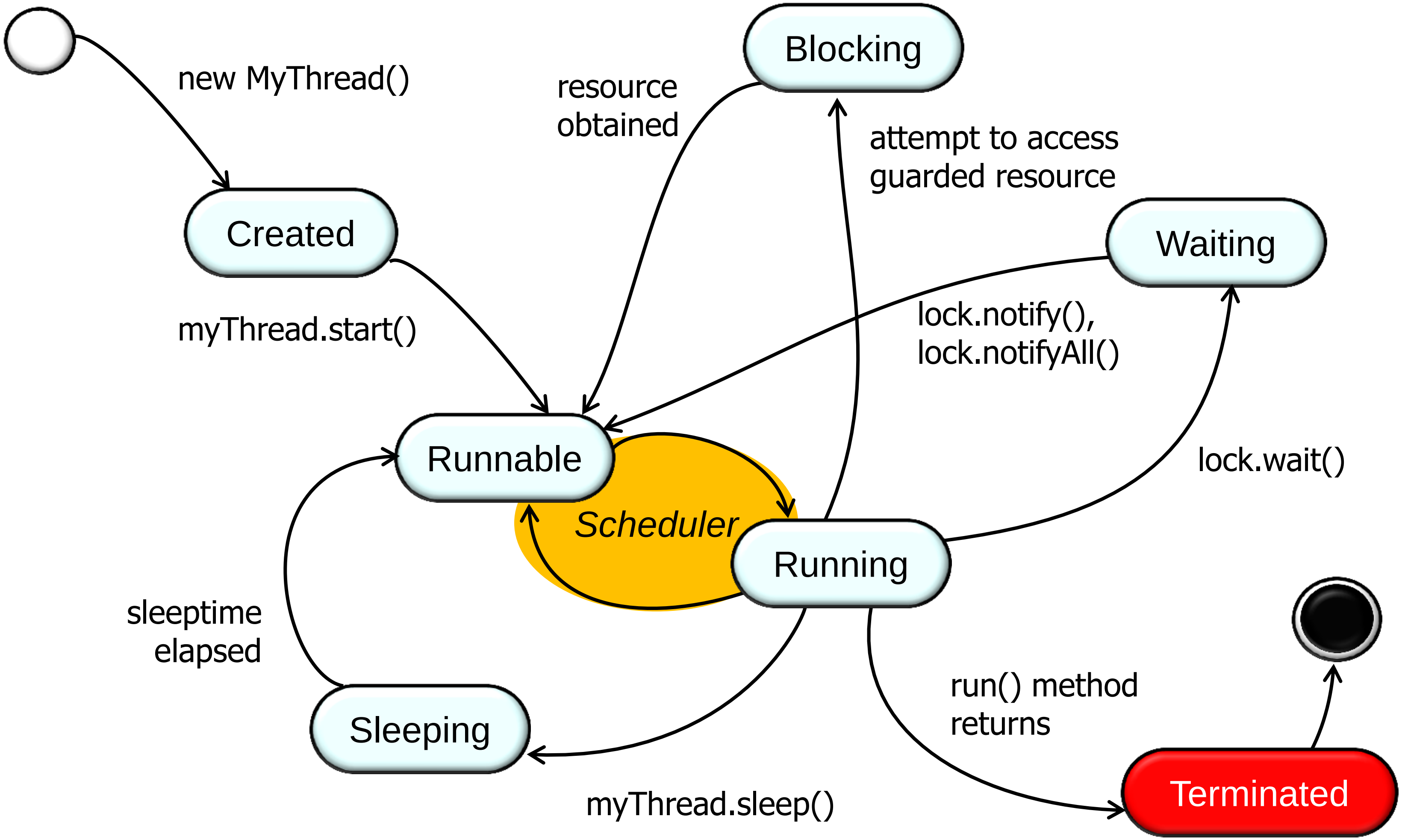
State Machine for Java Threads in Android



State Machine for Java Threads in Android



State Machine for Java Threads in Android



Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
```

Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
2. Thread.start() // Java method
```

See libcore/luni/src/main/java/java/lang/Thread.java

Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
```

See libcore/luni/src/main/java/java/lang/VMThread.java

Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
```

See dalvik/vm/native/java_lang_VMThread.cpp

Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

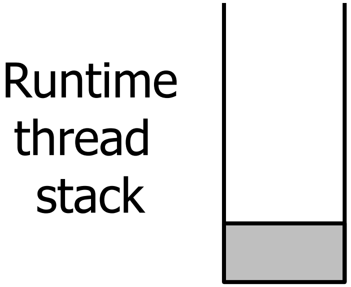
```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
```

See dalvik/vm/Thread.cpp

Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                   JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
```

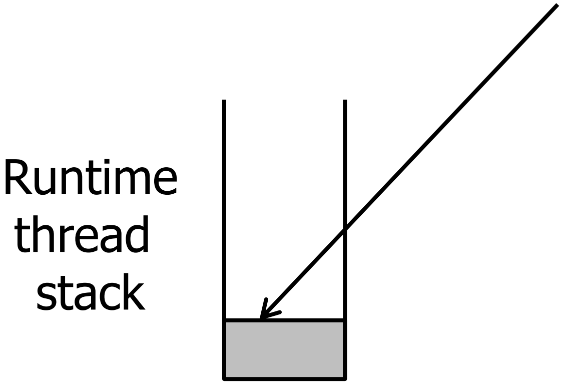


See bionic/libc/bionic/pthread.c

Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                   JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
                        7. interpThreadStart(void* arg) // Adapter
```

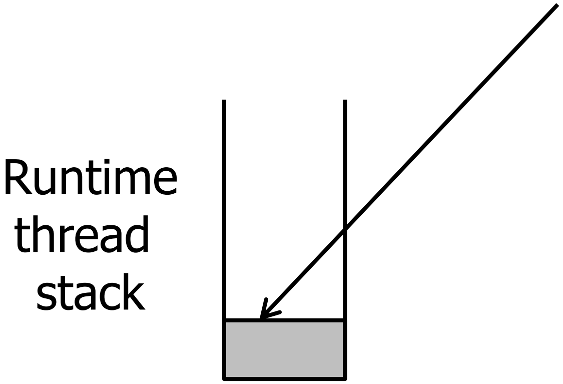


See dalvik/vm/Thread.cpp

Starting Java Threads

- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
                      7. interpThreadStart(void* arg) // Adapter
                      8. dvmCallMethod(self, run,
                                       self->threadObj,
                                       &unused) // Dalvik method
```

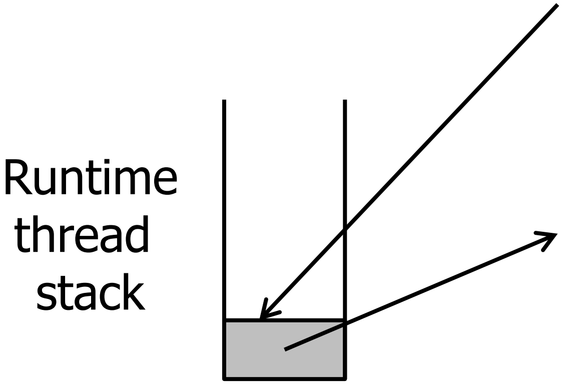


See dalvik/vm/interp/Stack.cpp

Starting Java Threads

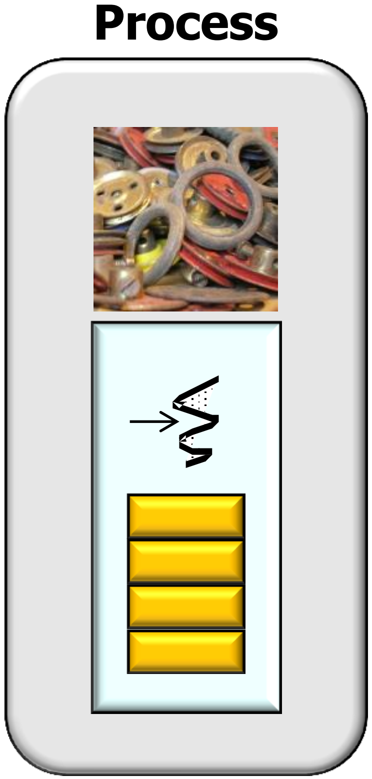
- When start() is called on a Java Thread object a whole series of steps occur

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
                       7. interpThreadStart(void* arg) // Adapter
                       8. dvmCallMethod(self, run,
                                        self->threadObj,
                                        &unused) // Dalvik method
                       9. MyThread.run() // User-defined hook
```



Stopping Java Threads

- Other than returning from run(), there's no "stop" method for a Java Thread
  - If you are going to create a long running operation inside of your run() method, you must ensure your code can stop voluntarily!



Stopping Java Threads

- Other than returning from run(), there's no "stop" method for a Java Thread
- One way to stop a thread is to use the interrupt() method
  - This method sends an interrupt request to the designated thread

```
Thread t1 =
  new Thread(new Runnable() {
   public void run(){
     for (int i = 0;
          i < input.length;
          i++) {
       process(input[i]);
       if (Thread.interrupted())
         throw InterruptedException();
   }
  }

t1.start();
...
t1.interrupt();
```

Stopping Java Threads

- Other than returning from run(), there's no "stop" method for a Java Thread
- One way to stop a thread is to use the interrupt() method
  - This method sends an interrupt request to the designated thread
  - Check Thread.interrupted() periodically to see if the thread's been stopped & throw InterruptedException

```
Thread t1 =
  new Thread(new Runnable() {
   public void run(){
     for (int i = 0;
          i < input.length;
          i++) {
       process(input[i]);
       if (Thread.interrupted())
         throw InterruptedException();
   }
  }

t1.start();
...
t1.interrupt();
```

Stopping Java Threads

- Other than returning from run(), there's no "stop" method for a Java Thread
- One way to stop a thread is to use the interrupt() method
  - This method sends an interrupt request to the designated thread
  - Check Thread.interrupted() periodically to see if the thread's been stopped & throw InterruptedException
  - Certain blocking operations will be automatically be interrupted
    - e.g., wait(), join(), sleep() & blocking I/O calls

```
Thread t1 =
   new Thread(new Runnable() {
    public void run(){
      for (int i = 0;
               i < input.length;
               i++) {
        process(input[i]);
        if (Thread.interrupted())
          throw InterruptedException();
    }
   }

t1.start();
...
t1.interrupt();
```

See developer.android.com/reference/java/lang/Thread.html#interrupt()

Stopping Java Threads

- Other than returning from run(), there's no "stop" method for a Java Thread
- One way to stop a thread is to use the interrupt() method
- Another way is to use a "stop" flag

```
public class MyRunnable
                implements Runnable {
  private volatile boolean
                      running_ = true;
  public void stop() {
    running_ = false;
  }
  public void run() {
    while(running_) {
      // do stuff
    }
  }
}
```

Stopping Java Threads

- Other than returning from run(), there's no "stop" method for a Java Thread
- One way to stop a thread is to use the interrupt() method
- Another way is to use a "stop" flag
  - Add a volatile boolean flag "running_" to your class that implements Runnable
    - Initially, set "running_" to true

```
public class MyRunnable
             implements Runnable {
  private volatile boolean
                  running_ = true;
  public void stop() {
    running_ = false;
  }
  public void run() {
    while(running_) {
      // do stuff
    }
  }
}
```

en.wikipedia.org/wiki/Volatile_variable#In_Java has more on volatile

Stopping Java Threads

- Other than returning from run(), there's no "stop" method for a Java Thread
- One way to stop a thread is to use the interrupt() method
- Another way is to use a "stop" flag
  - Add a volatile boolean flag "running_" to your class that implements Runnable
  - Have a stop() method that sets "running_" to false

```
public class MyRunnable
             implements Runnable {
  private volatile boolean
                  running_ = true;
  public void stop() {
    running_ = false;
  }
  public void run() {
    while(running_) {
      // do stuff
    }
  }
}
```

Stopping Java Threads

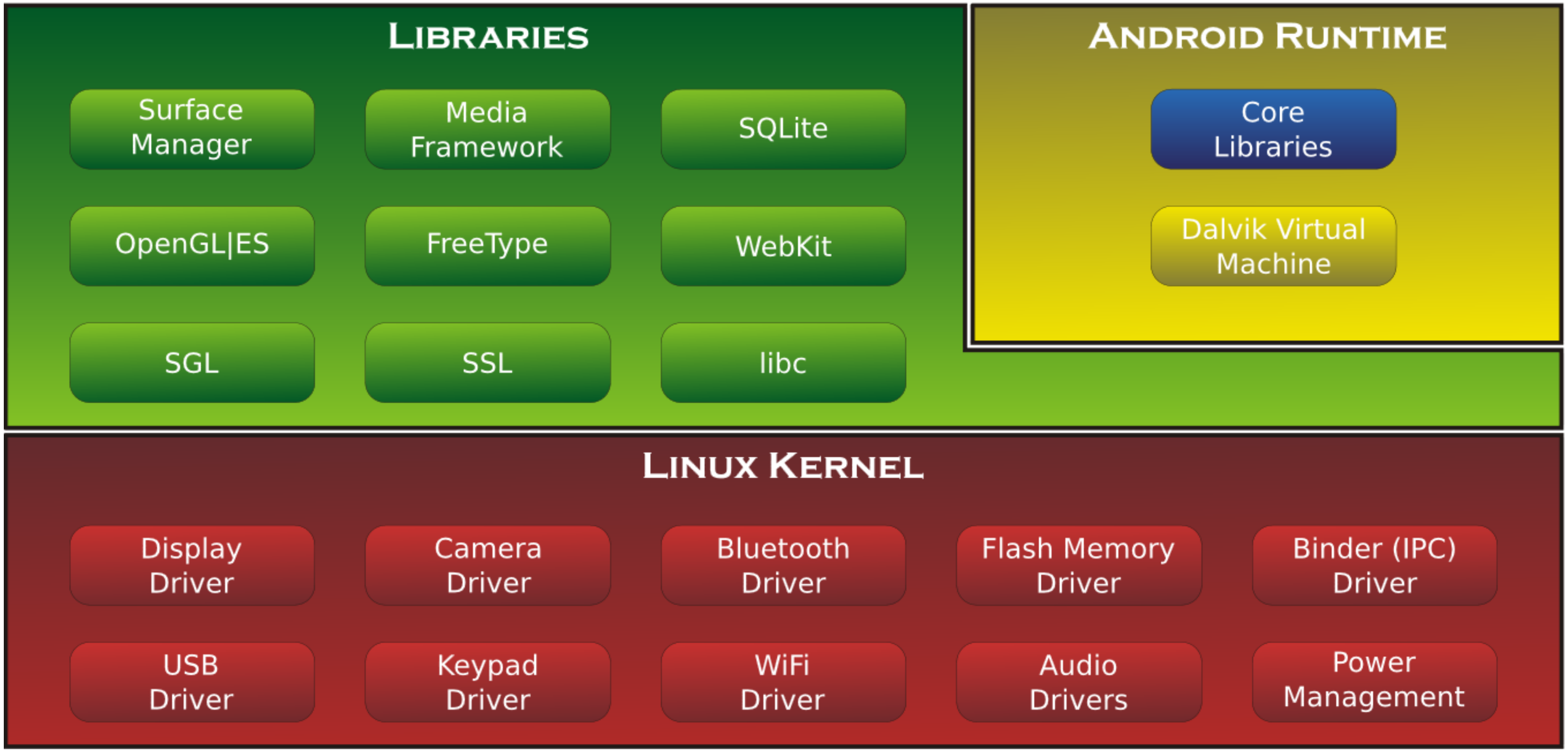
- Other than returning from run(), there's no "stop" method for a Java Thread
- One way to stop a thread is to use the interrupt() method
- Another way is to use a "stop" flag
  - Add a volatile boolean flag "running_" to your class that implements Runnable
  - Have a stop() method that sets "running_" to false
  - Check "running_" periodically to see if the thread's been stopped

```
public class MyRunnable
             implements Runnable {
  private volatile boolean
                   running_ = true;
  public void stop() {
    running_ = false;
  }
  public void run() {
    while(running_) {
      // do stuff
    }
  }
}
```

This solution requires developers to periodically check if thread was stopped

Summary

- Java Threads are implemented using various methods & functions defined by lower layers of the Android software stack


LIBRARIES
Surface Manager
Media Framework
SQLite
OpenGL|ES
FreeType
WebKit
SGL
SSL
libc
ANDROID RUNTIME
Core Libraries
Dalvik Virtual Machine
LINUX KERNEL
Display Driver
Camera Driver
Bluetooth Driver
Flash Memory Driver
Binder (IPC) Driver
USB Driver
Keypad Driver
WiFi Driver
Audio Drivers
Power Management