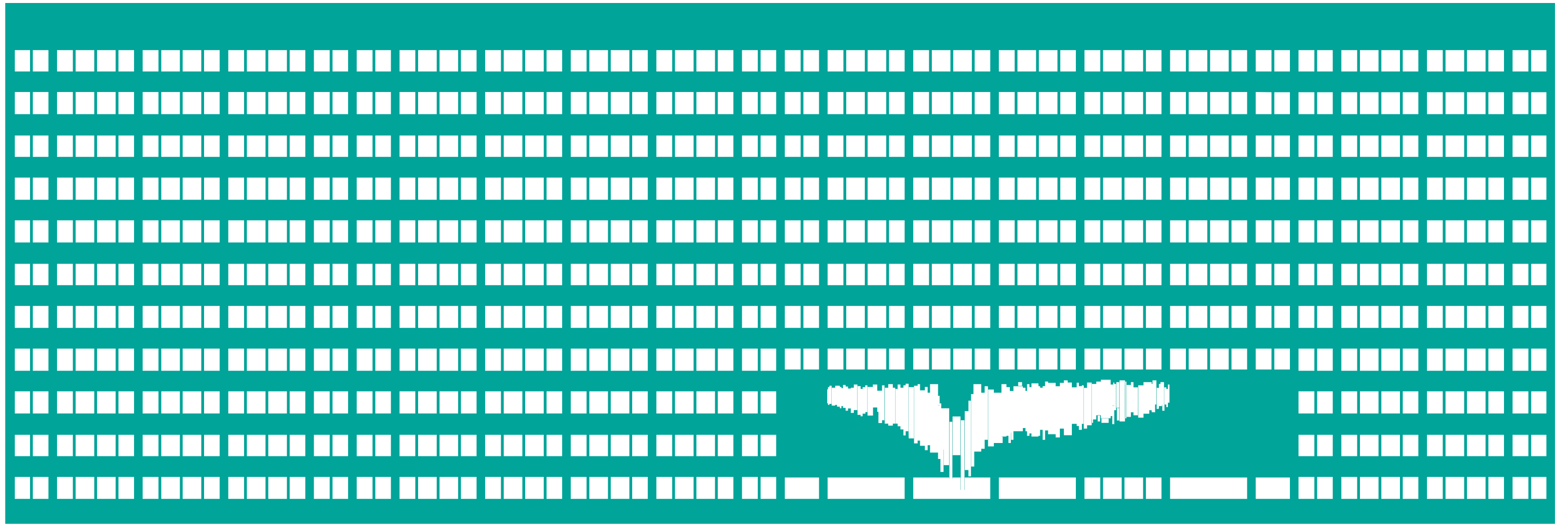
VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

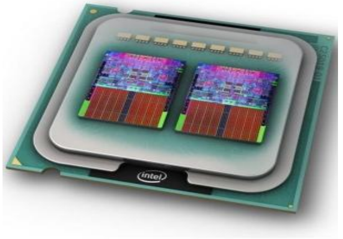
Concurrency Java, Kotlin
&
Android

Processes and Threads

- **Multitasking** is the capability of the operating system to load many programs into memory simultaneously and share CPU time.
  - **Processes**
    - Self-contained execution environment
    - Private set of resources
    - Private virtual memory address space
    - Communication through IPC
- **Multithreading** is the capability of the operating system to support many independent units of execution in a single process.
  - **Threads**
    - Lightweight processes
    - Exists within a process
    - Share resources, memory, open files

Threading in Android

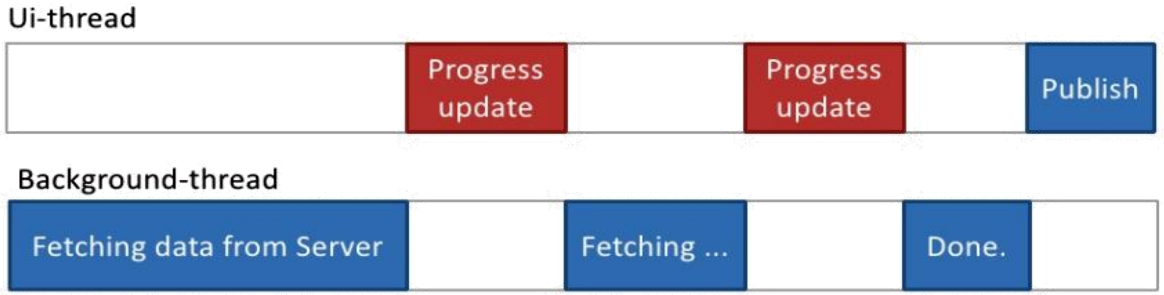
- Threads share the processes resources but are able to execute independently.
- Applications responsibilities can be separated
  - **Main thread runs UI**
  - **Slow tasks are sent to background threads**



- Particularly useful in the case of a single process that spawns multiple threads on top of a multiprocessor system. In this case real parallelism is achieved.
- Consequently, a multithreaded program operates faster on computer systems that have multiple CPUs.

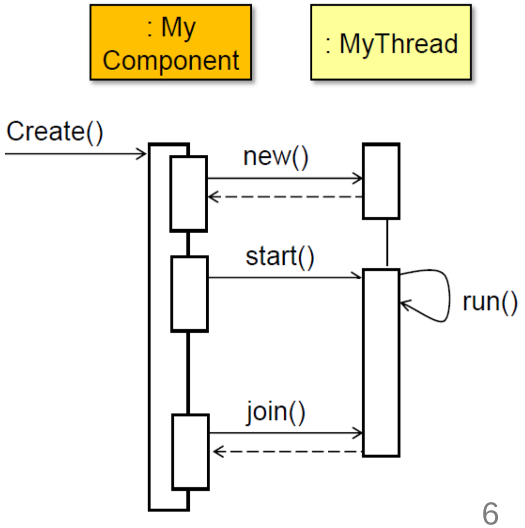
Threading in Android

- A Thread is a concurrent unit of execution
- Has its **own program counter, call stack** for methods being invoked, **arguments** and **local variables**
- **Heap and static areas are shared across threads.**
- Each virtual machine instance has at least one main Thread running when it is started – typically there are several others for housekeeping
- The application can decide to launch additional Threads for specific tasks



Basic Threading in Java

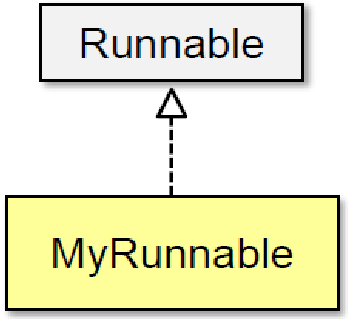
- Threads in the same VM interact and **synchronize** by the use of shared objects and monitors associated with these objects.
- There are basically two main ways of having a Thread execute application code.
  - Provide a new class that **extends Thread** and override its run() method
  - Provide a **new Thread instance** with a Runnable object during creation.
- Thread can be **active as long as the run() method hasn't returned**.
- Android Scheduler can **suspend/resume threads**.
- When **run() returns the thread is no longer active.**



Defining and Starting a Thread in Java

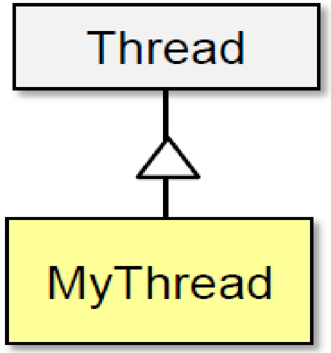
- Runnable object

```
public class MyRunnable implements Runnable {
      public void run()
         { System.out.println("Hello from a thread!"); }
        public static void main(String args[])
         { new Thread(new HelloRunnable())).start(); }
}
```



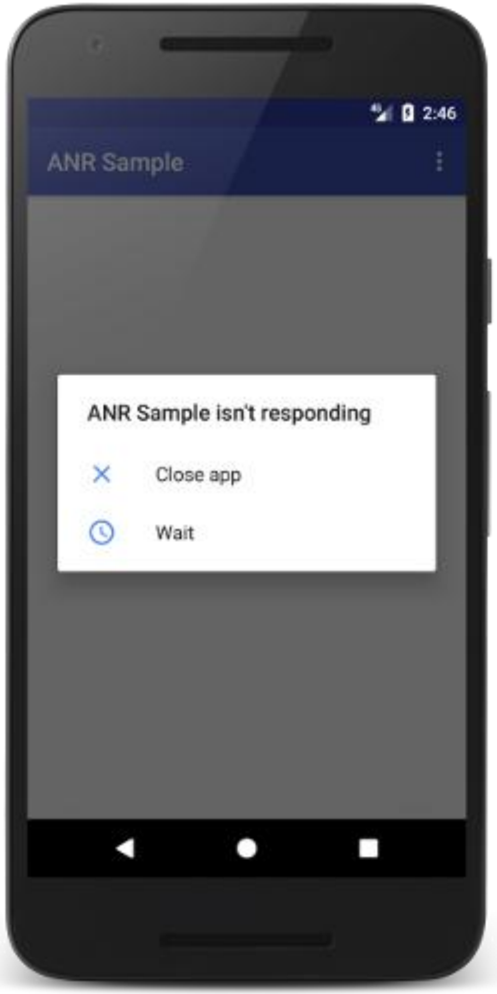
- Subclass Thread

```
public class MyThread extends Thread {
        public void run() {
         System.out.println("Hello thread!");
        }
        public static void main(String args[]) {
         (new HelloThread()).start();
        }
}
```



ANR Error

- When the UI thread of an Android app is blocked for too long, an "**Application Not Responding**" (ANR) error is triggered.
- If the app is in the foreground, the system displays a dialog to the user
- There are some common patterns to look for when diagnosing ANRs:
  - The app is doing **slow operations involving I/O** on the main thread.
  - The app is doing **a long calculation** on the main thread.
  - The main thread is doing **a synchronous binder call** to another process, and that other process is taking a long time to return.
  - The main thread is **blocked waiting for a synchronized block** for a long operation that is happening on another thread.
  - The main thread is in a **deadlock with another thread**, either in your process or via a binder call.

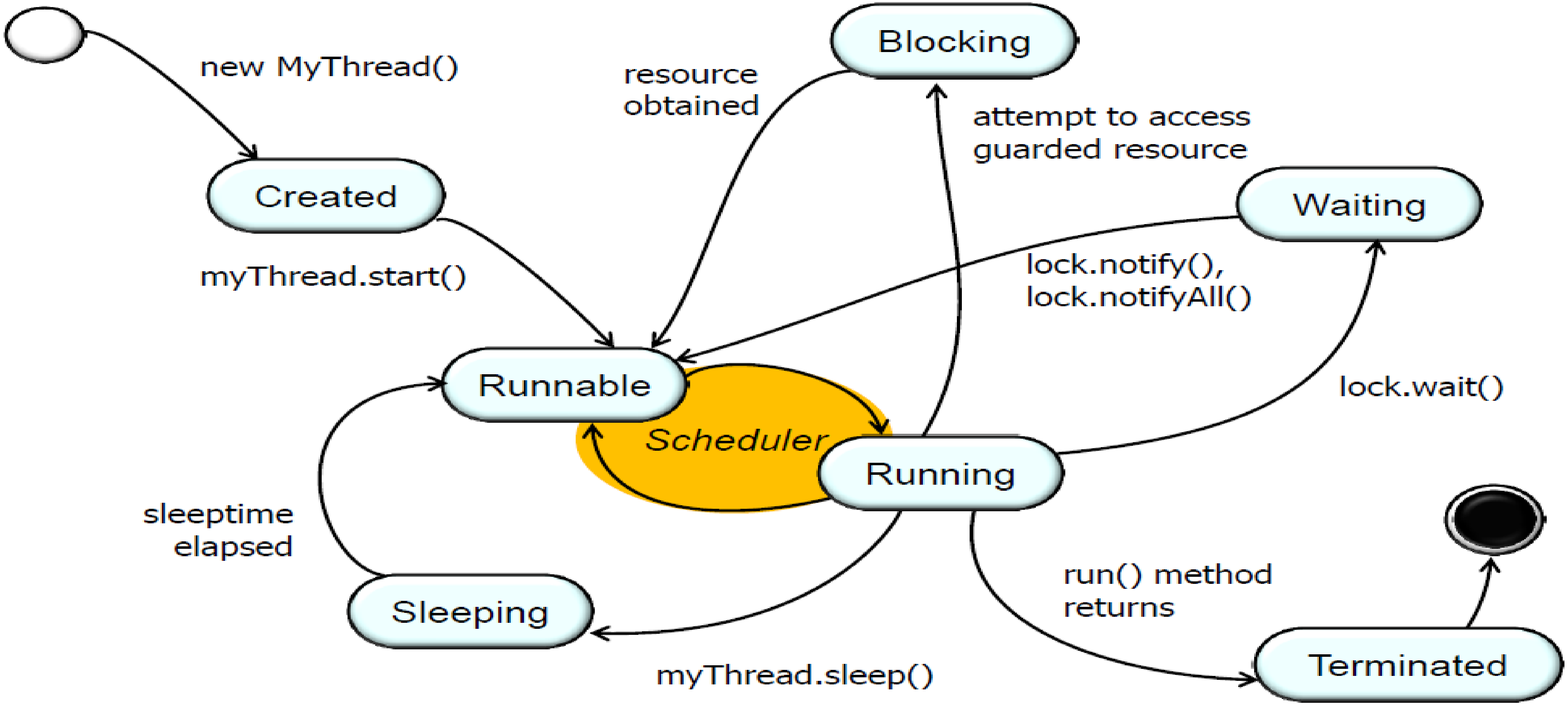


ANR Sample isn't responding

Thread Internals in Android

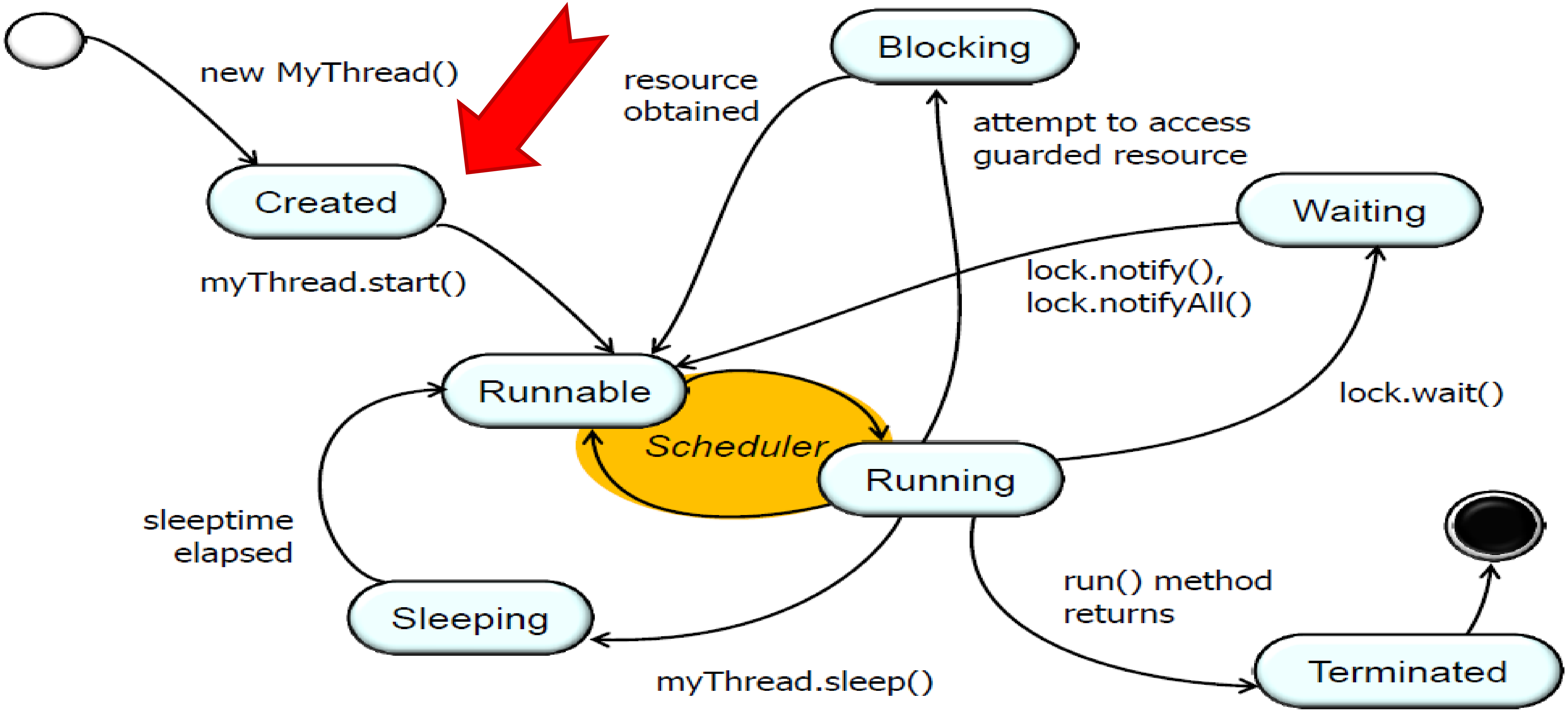
```
1.    MyThread.start()
2.    Thread.start()
      // Java method
3.    VMThread.create()
      // Native method      /src/main/java/java/lang/VMThread.java
4.    Dalvik_java_lang_VMThread_create(const u4* args, JValue* pResult)
      // JNI method         /vm/native/java_lang_VMThread.cpp
5.    dvmCreateInterpThread(Object *threadObj, int reqStackSize)
      // Dalvik method      /vm/native/java_lang_VMThread.cpp
6.    pthread_create(&threadHandle, &threadAttr, interpThreadStart, newThread)
      // pthread method     /libc/bionic/pthread.c
7.    interpThreadStart(void *arg)
      // adapter           /vm/Thread.c
8.    dvmCallMethod(self, run, self->threadObj, &unused)
      // Dalvid method      /vm/interp/Stack.cpp
9.    MyThread.run()
      // User-defined hook
```

State Machine of Threads in Android



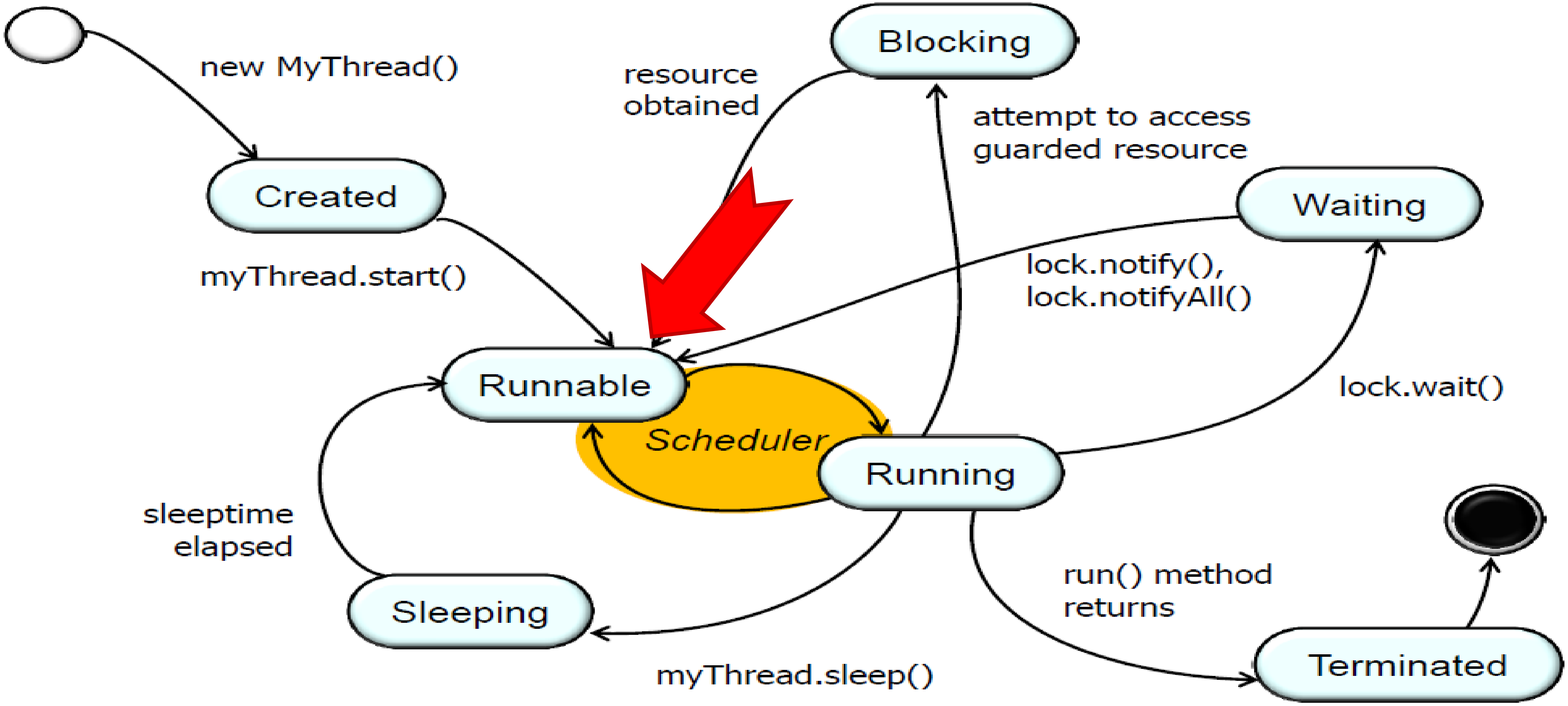
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



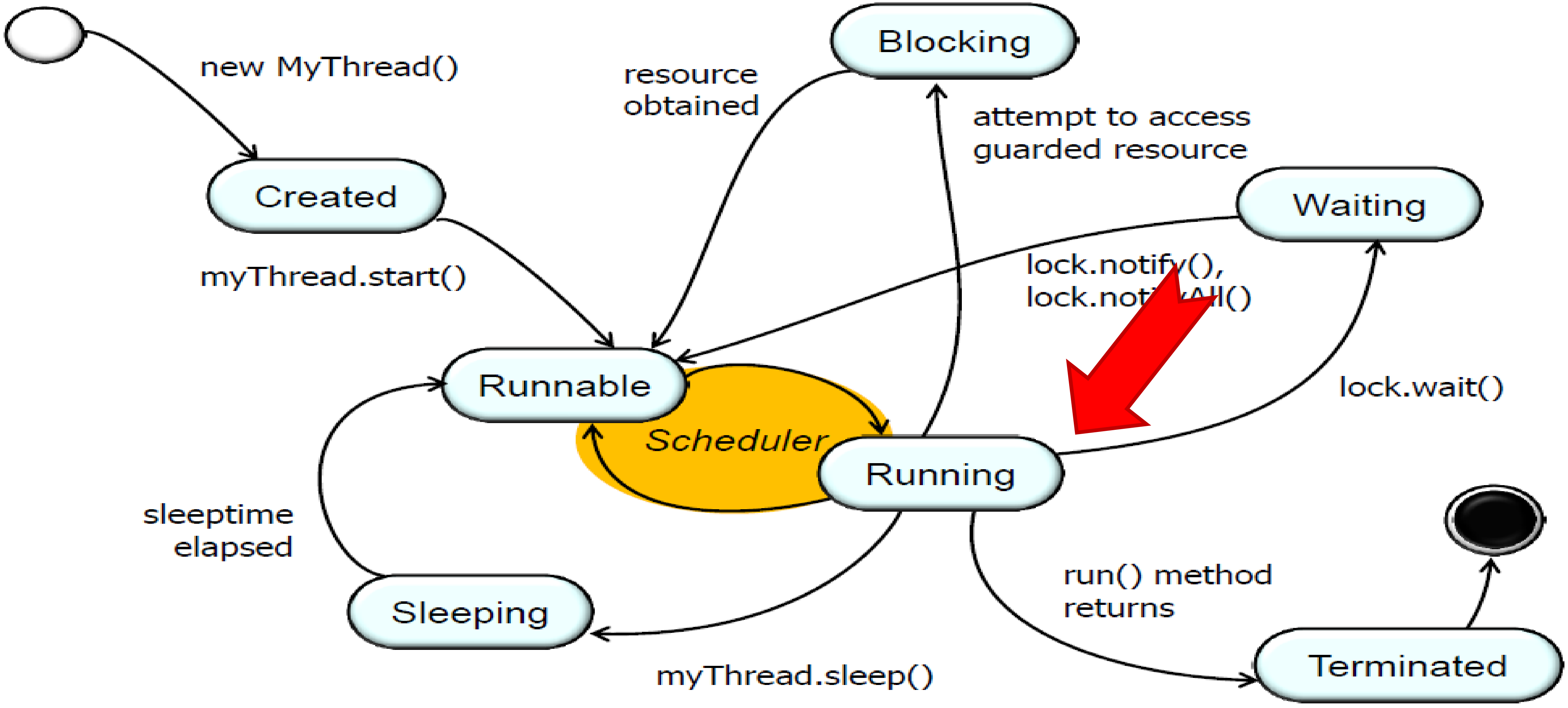
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



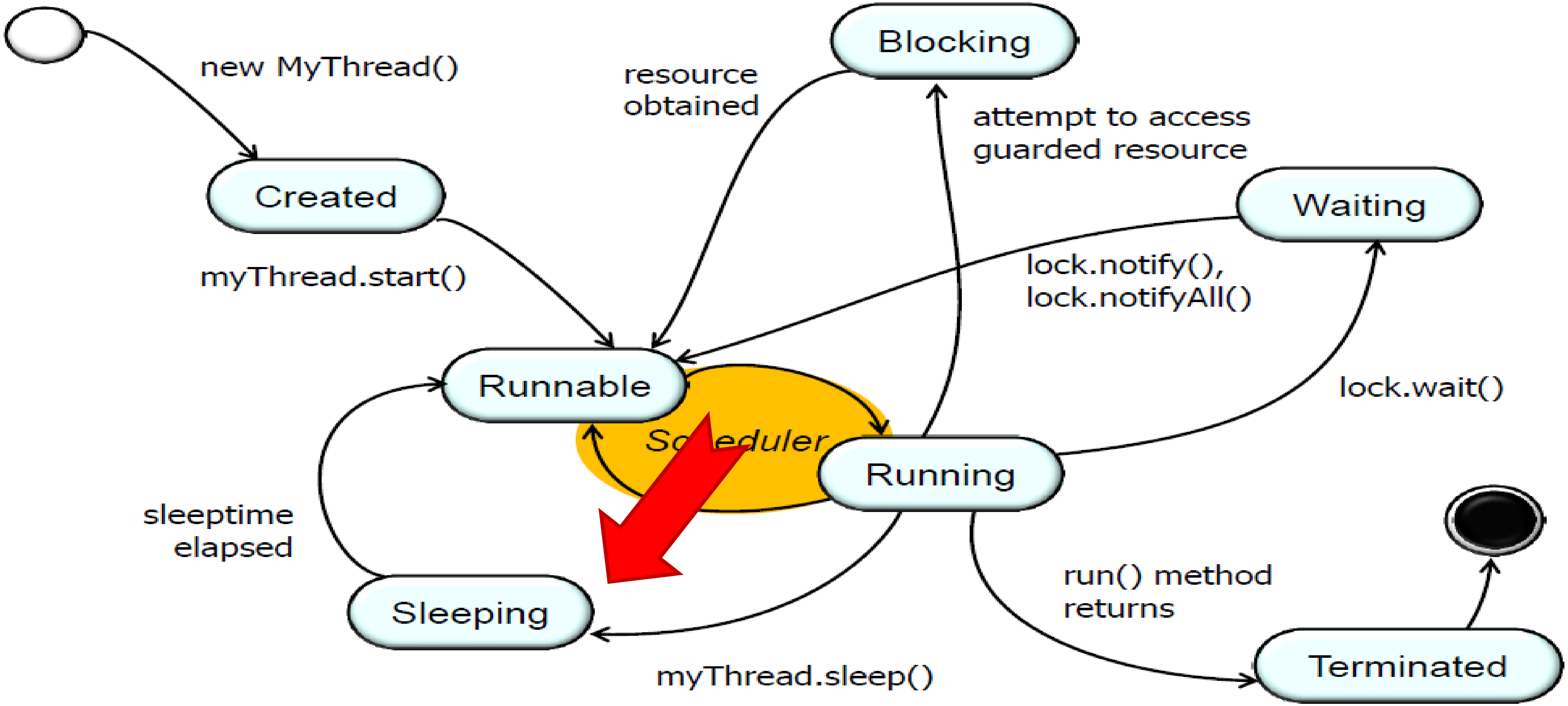
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



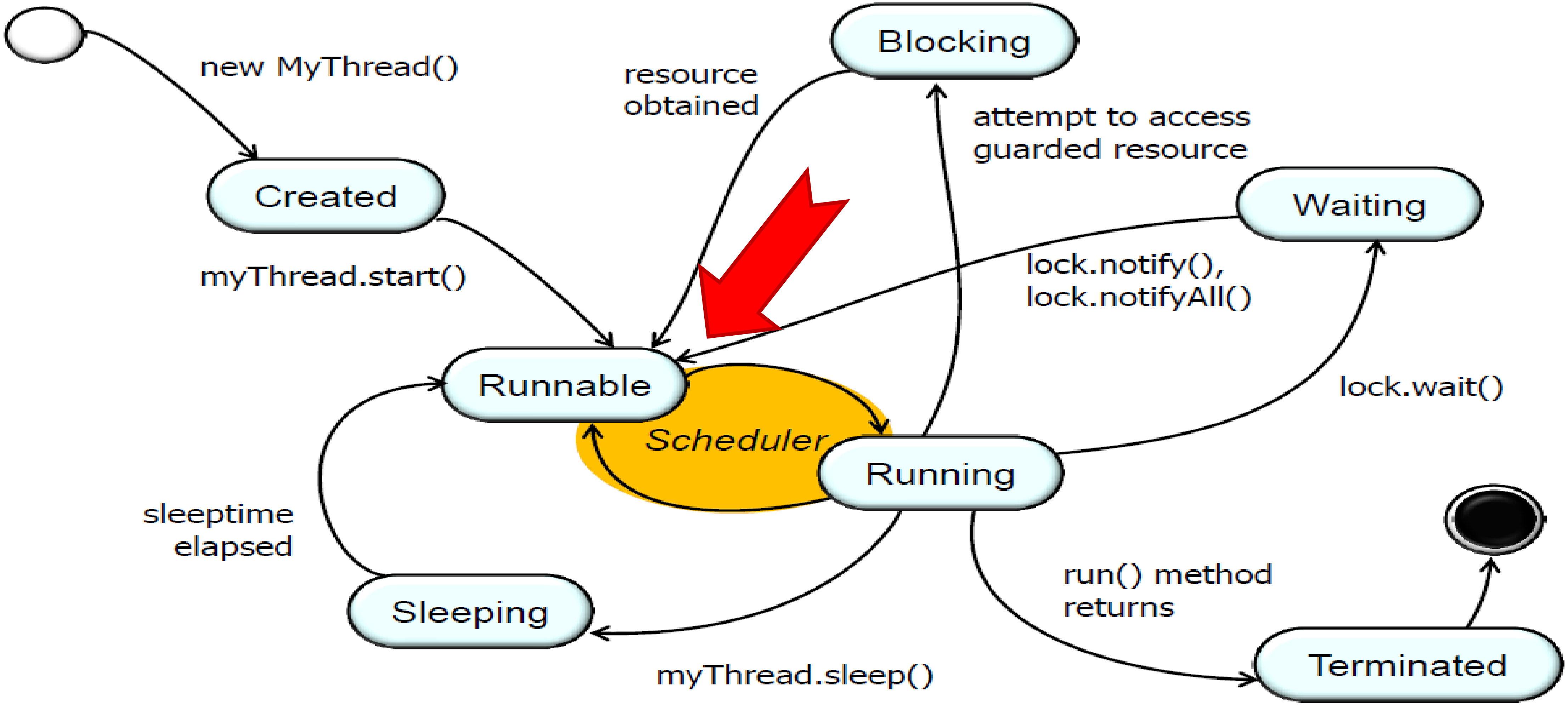
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



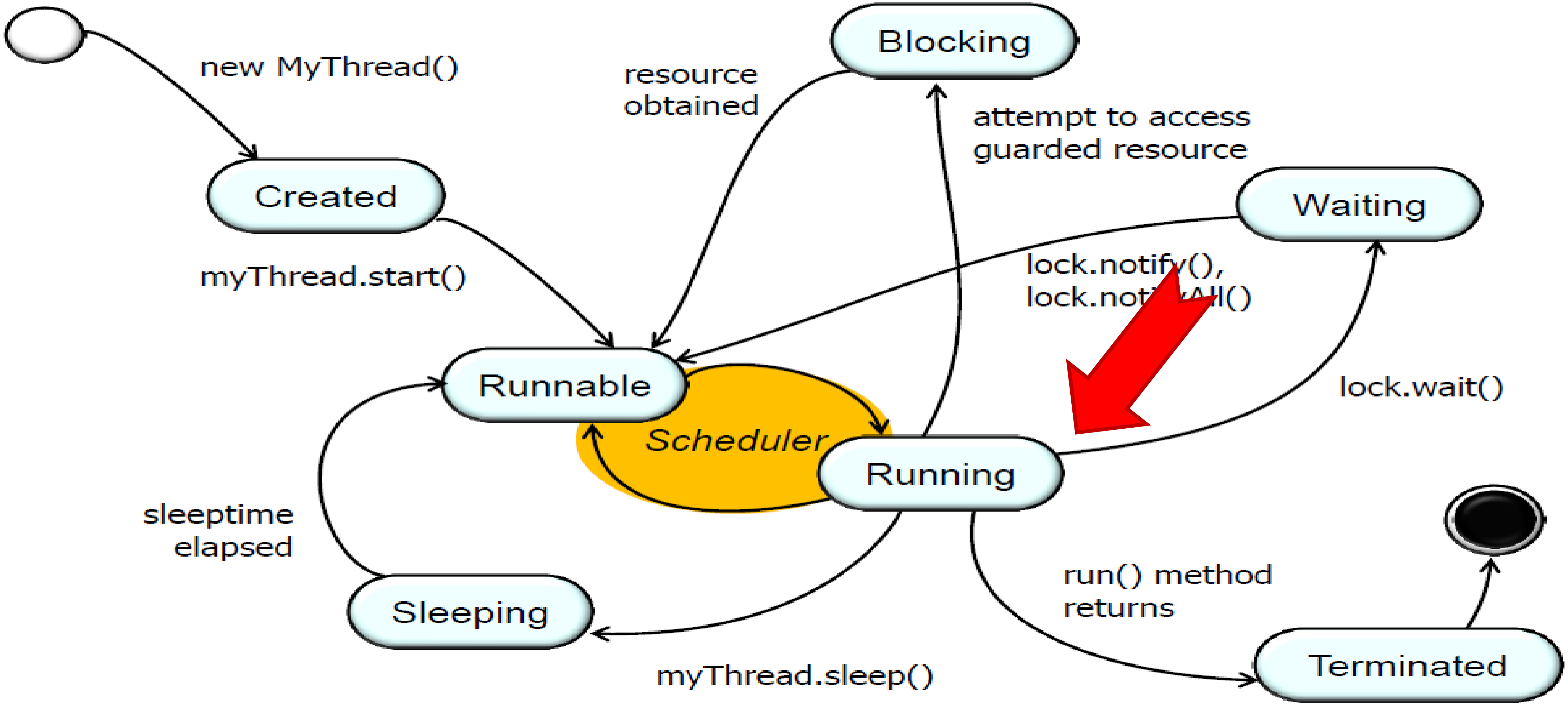
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



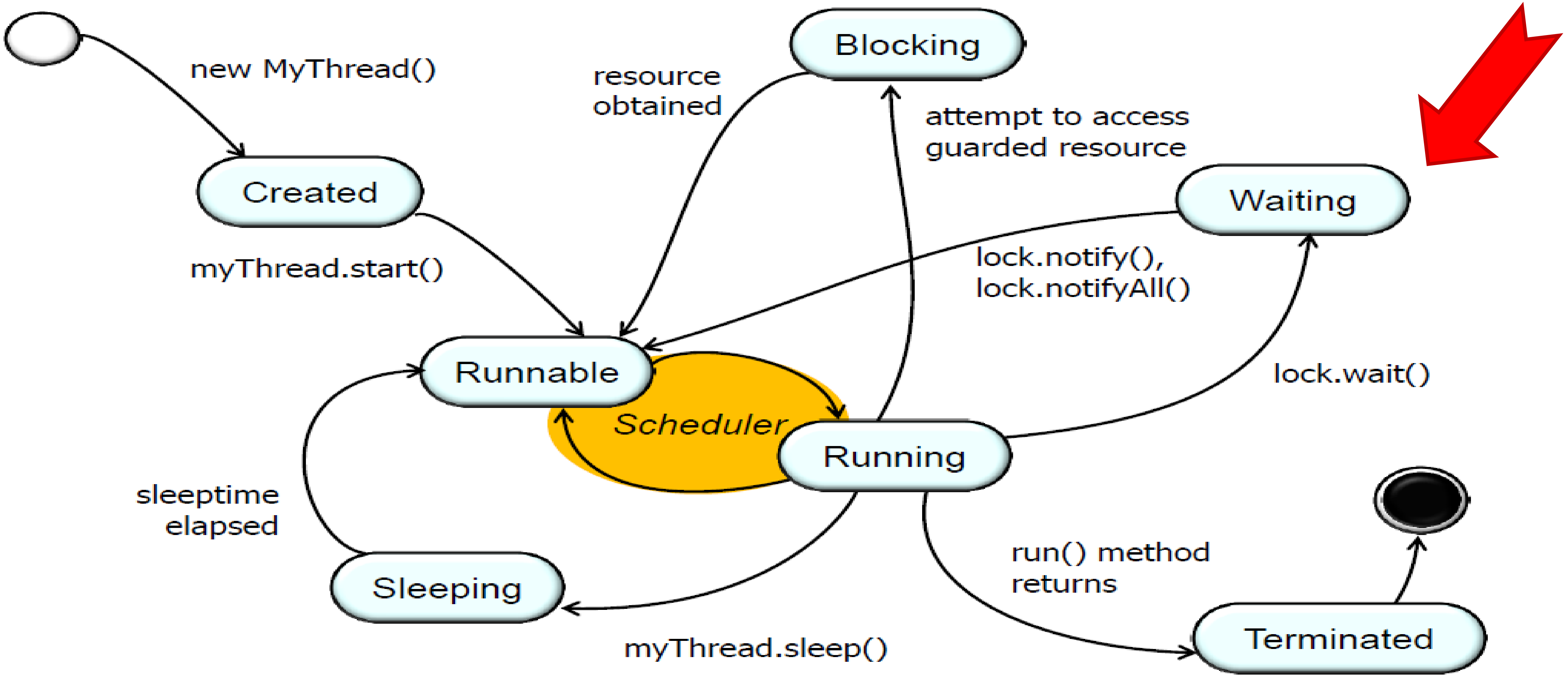
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



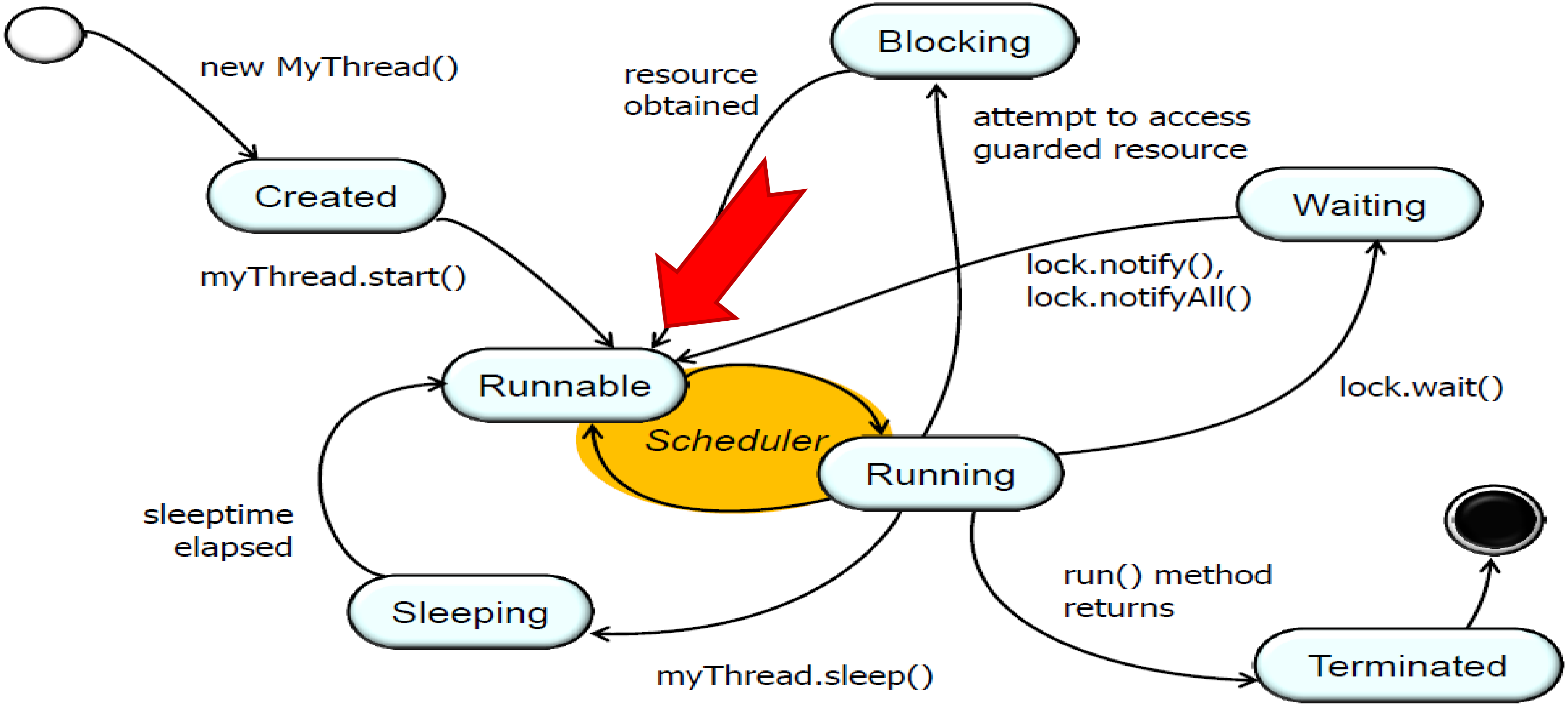
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



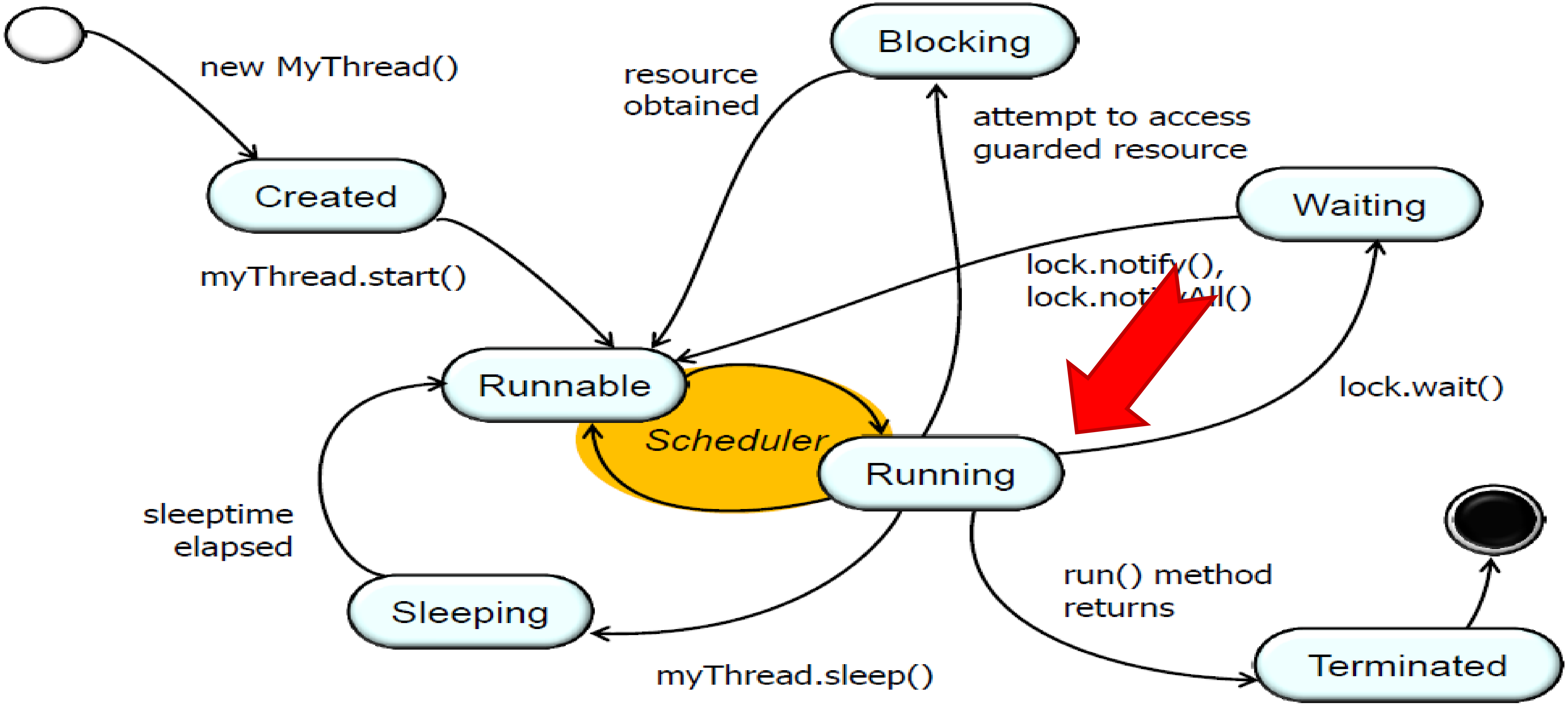
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



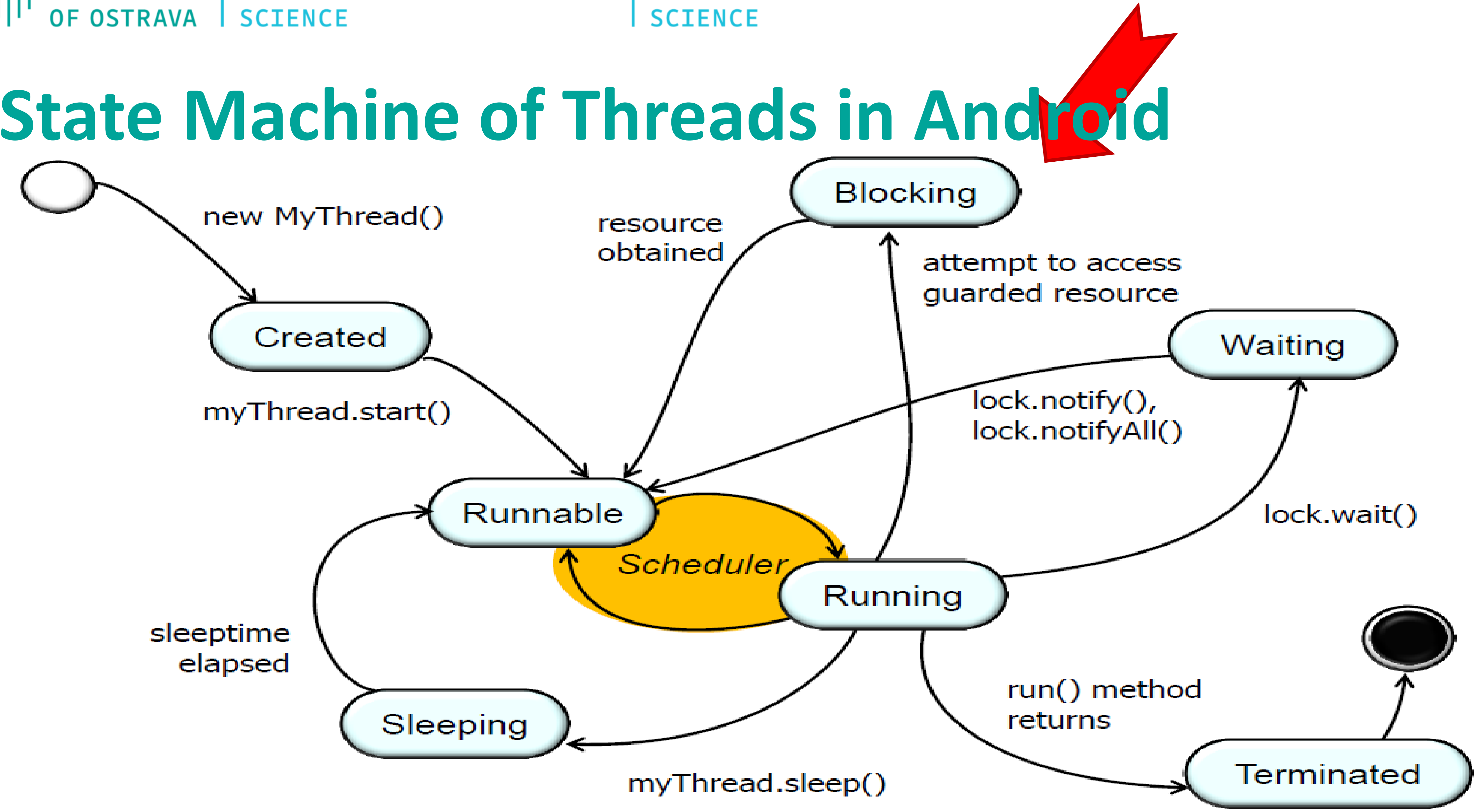
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



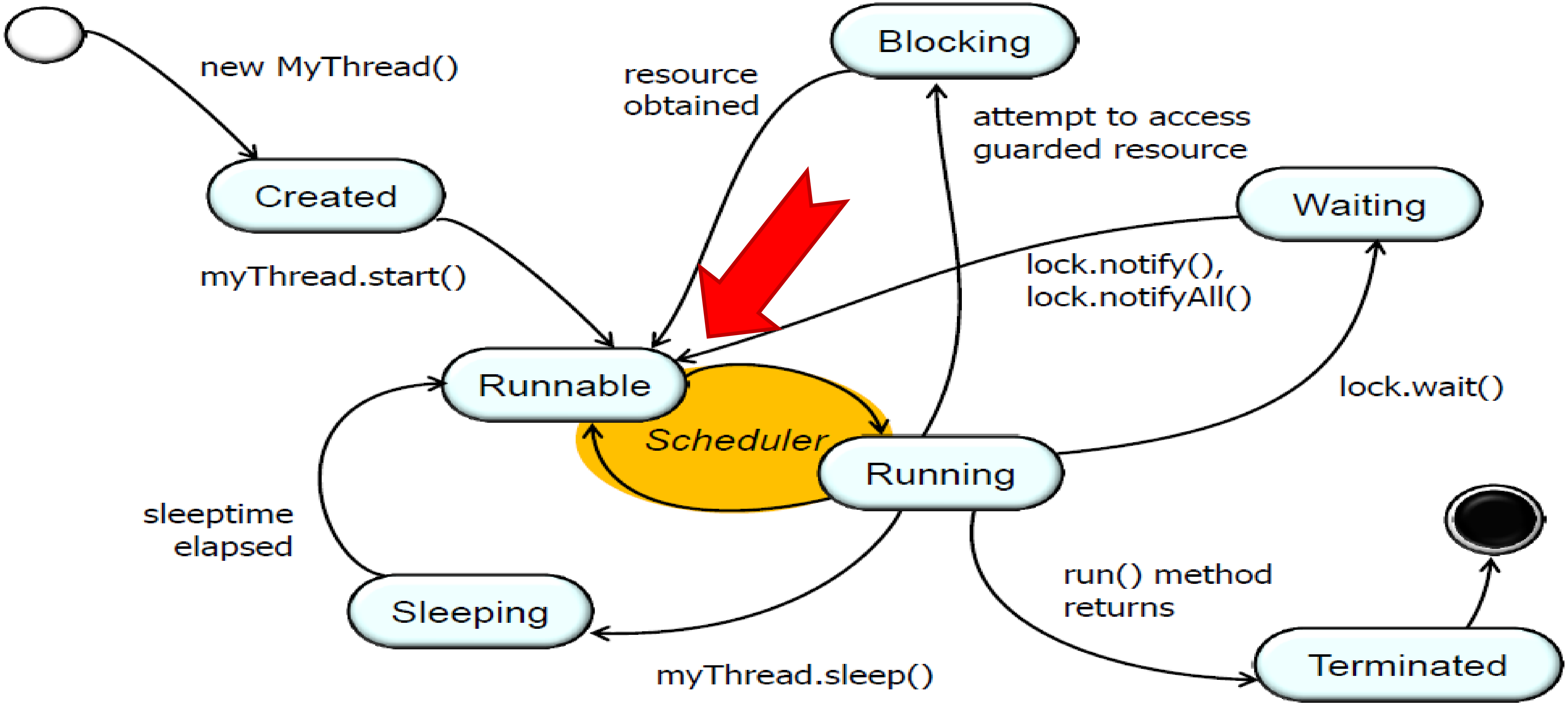
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



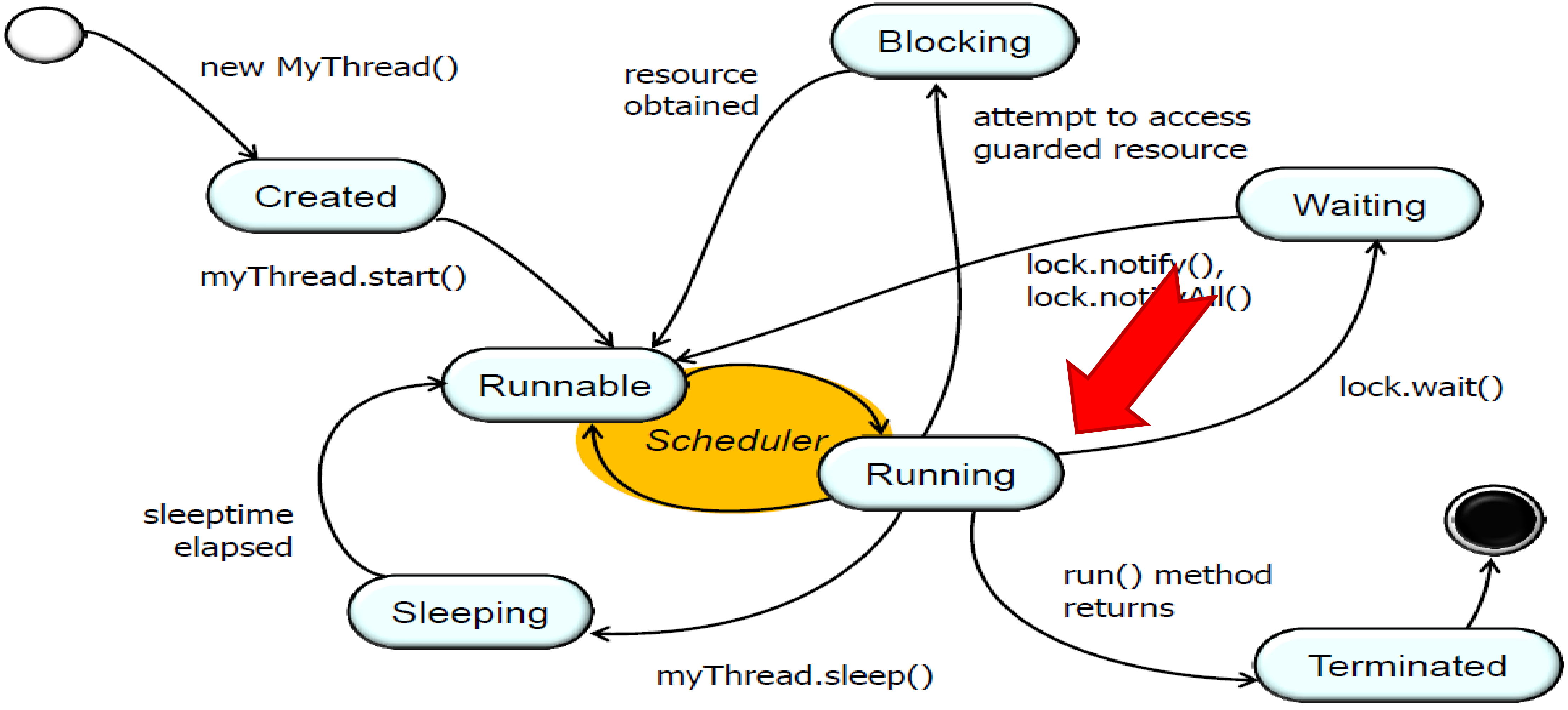
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



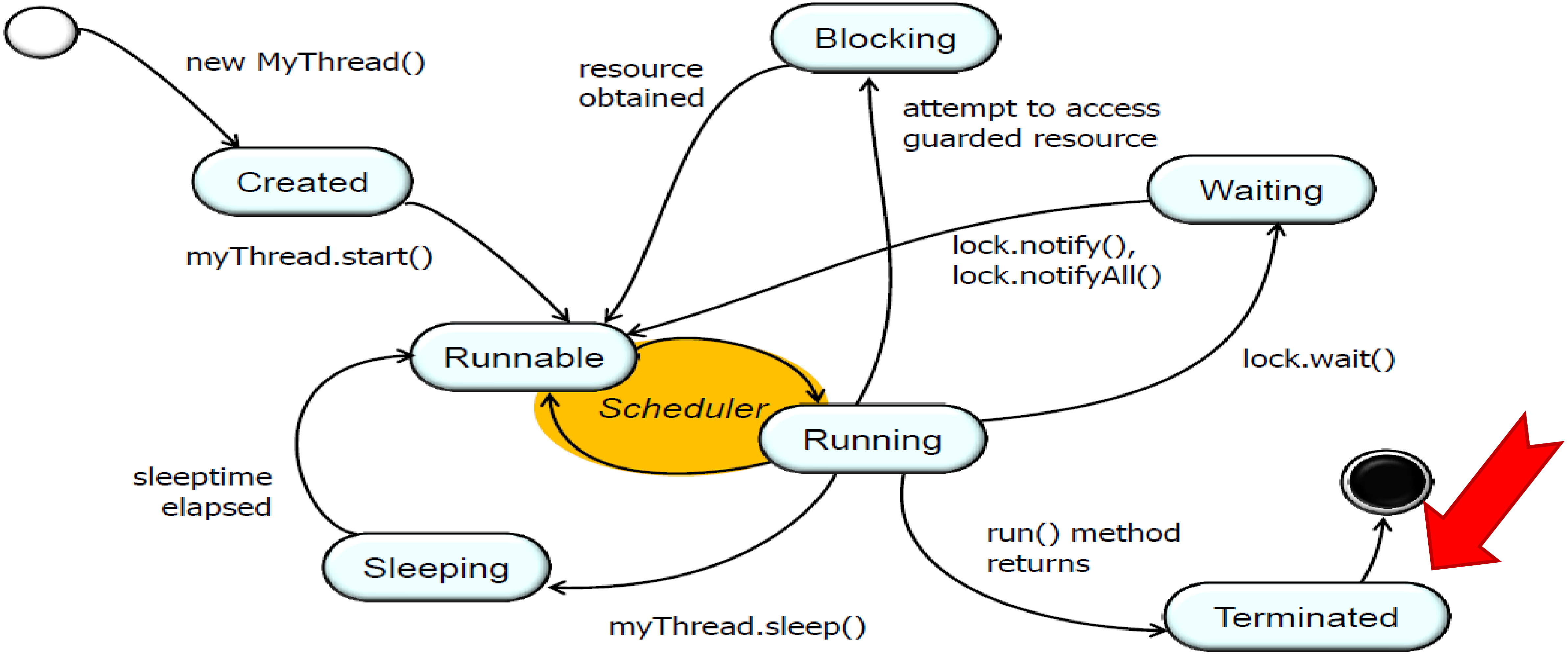
Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

State Machine of Threads in Android



Source: https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

Additional Methods

- **yield()**
  - Causes the calling Thread to yield execution time to another Thread that is ready to run.

- **join()**
  - Blocks the current Thread until the receiver finishes its execution and dies. Threads that have completed are “not alive” as are threads that have not yet been started.
  - Or specific time expires **join(long millis)**

- **sleep(long time)**
  - Causes the thread which sent this message to sleep for the given interval of time (given in milliseconds).

Yield Example

```
class MyThread extends Thread {
    private String name;
    public MyThread(String name) {
        this.name = name;
    }
    public void run() {
        for (;;) {
        System.out.println(name + ": hello world");
        }
    }
}

public class Main2 {
    public static void main(String [] args) {
        MyThread t1 = new MyThread("thread1");
        MyThread t2 = new MyThread("thread2");
        t1.start(); t2.start();
    }
}
```

```
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread2: hello world
thread1: hello world
```

Yield Example

```
class MyThread extends Thread {
    private String name;
    public MyThread(String name) {
        this.name = name;
    }
    public void run() {
        for (;;) {
        System.out.println(name + ": hello world");
        yield();
        }
    }
}

public class Main2 {
    public static void main(String [] args) {
        MyThread t1 = new MyThread("thread1");
        MyThread t2 = new MyThread("thread2");
        t1.start(); t2.start();
    }
}
```

```
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
```

Thread Scheduling

- **Scheduling**
  - Priority inherited from parent, but can be changed
  - Higher priority threads generally run before lower priority threads
  - For equal priority threads, best to call yield() intermittently to handle JVM's with user-level threading (i.e., no time-slicing)
- Modifying the priority
  - **setPriority**(int newPriority)
    - public static final int MAX_PRIORITY
    - public static final int MIN_PRIORITY
  - **getPriority()**

stop() was deprecated, because stopping a thread in this manner is unsafe and can leave your application and the VM in an unpredictable state.

suspend(), resume() may cause deadlocks.

Stopping Threads

- **Interrupt method**
  - Sends an **interrupt** request to the designated thread
  - Check **Thread.interrupted()** if the thread was stopped

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

Stopping Threads

- **Flags**
  - Add a volatile boolean is_running and check its state

```
Thread t = new Thread(new Runnable() {
      private volatile boolean is_running = true;
      public void stop() {
       is_running = false;
      }

      public void run() {
          while (is_running) {
              …
          }
      }
}
```

Synchronization Methods

- **Atomic action happens all at once**
  - **Cannot be interleaved**
  - Some atomic operations
    - Read/Write are atomic for reference variables
    - Read/Write for all variables declared volatile
  - Changes to volatile are **visible to all threads**

If we have multiple threads accessing shared data,
how do we synchronize access to ensure it remains in a consistent state?

- **Synchronized methods, objects**
- **Wait, notify, notifyAll**

Synchronized Methods - Monitors

- **synchronized** keyword used with a method
  - Example

```
public synchronized void SetValue() {
    // Update instance data structure.
    // When the thread executes here,  it exclusively has the monitor lock
}
```

  - Provides *instance-based* mutual exclusion
    - A lock is implicitly provided-- allows at most one thread to be executing the method at one time
  - Used on a per method basis; not all methods in a class have to have this
    - But, you’ll need to design it right!!

Synchronized Methods - Wait

- Does a blocking (**not busy) wait**
- Relative to an Object
  - E.g., Used within a synchronized method
- Releases lock on Object and waits until a condition is true
  - Blocks calling process until notify() or notifyAll() is called on same object instance (or exception occurs)
- Typically used within a loop to re-check a condition
  - wait(long millis); // bounded wait

Synchronized Methods - Notify

- Stop a process from waiting– wakes it up
- Relative to an Object
  - E.g., Used within a synchronized method
- Wakes up a blocked thread (**notify**) or all blocked threads (**notifyAll**)
  - One woken thread reacquires lock; The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
- For notify, if more than one thread available to be woken, then one is picked

Guarded Blocks

- Block begins by polling a condition that must be true before the block can proceed.
- **Object.wait** is used to suspend current thread. **Object.notifyAll** informing all threads waiting on that lock that something important has happened

```
while (condition not true) {
    try {
        wait(); // this.wait();
    } catch {
        System.out.println(“Interrupted!”);
    } }
// After loop, condition now true & thread
// has monitor lock for this object instance
```

Lock Objects

- **Lock objects** work very much like the implicit locks used by synchronized code.
- Ability to back out of an attempt to lock
  - tryLock
  - lockInterruptibly

```
try {
    myLock = lock.tryLock();
    yourLock = bower.lock.tryLock();
}
    finally
{
    if (! (myLock && yourLock)) {
           if (myLock) { lock.unlock(); }
       if (yourLock) { bower.lock.unlock(); } }
}
```

Semaphores

- 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.

Kotlin Concurrency – Coroutines Basics

Atomic Variables

- Kotlin provides atomic variable classes such as **AtomicInt**, **AtomicLong**, **AtomicReference**, etc.,
- Perform operations atomically **without the need for explicit synchronization.**

```
val atomicCounter = AtomicInt(0)

fun incrementCounter() {
    atomicCounter.incrementAndGet()
}
```

Kotlin Concurrency – Coroutines Basics

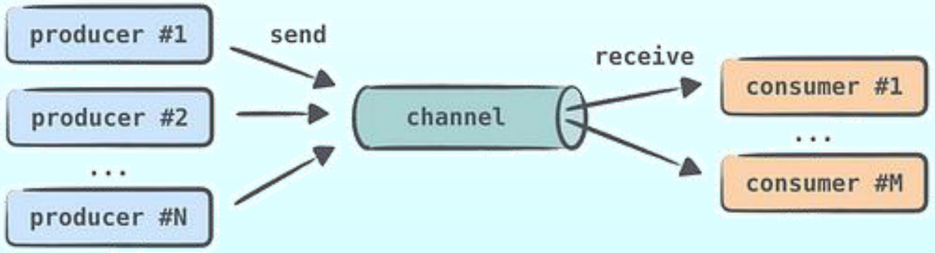
Channels

- Multiple coroutines to **communicate** with each other in a **producer-consumer fashion**
- Channels can be used for both **thread-to-thread** communication and **coroutine-to-coroutine** communication.

```
val channel = Channel<Int>()

GlobalScope.launch {
    val value = channel.receive()
    // Process the received value
}

// Sending a value to the channel
channel.send(42)
```



Kotlin Concurrency – Coroutines Basics

Actor

- Entity that can **receive messages** and **process them sequentially**. Can handle concurrent operations in a structured manner.
- Encapsulate mutable state and ensure that it is **accessed by only one coroutine at a time**.

```
data class Message(val content: String)

val actor = GlobalScope.actor<Message> {
    for (msg in channel) {
        // Process the message
        println(msg.content)
    }
}
// Sending a message to the actor
actor.send(Message("Hello, Actor!"))
```

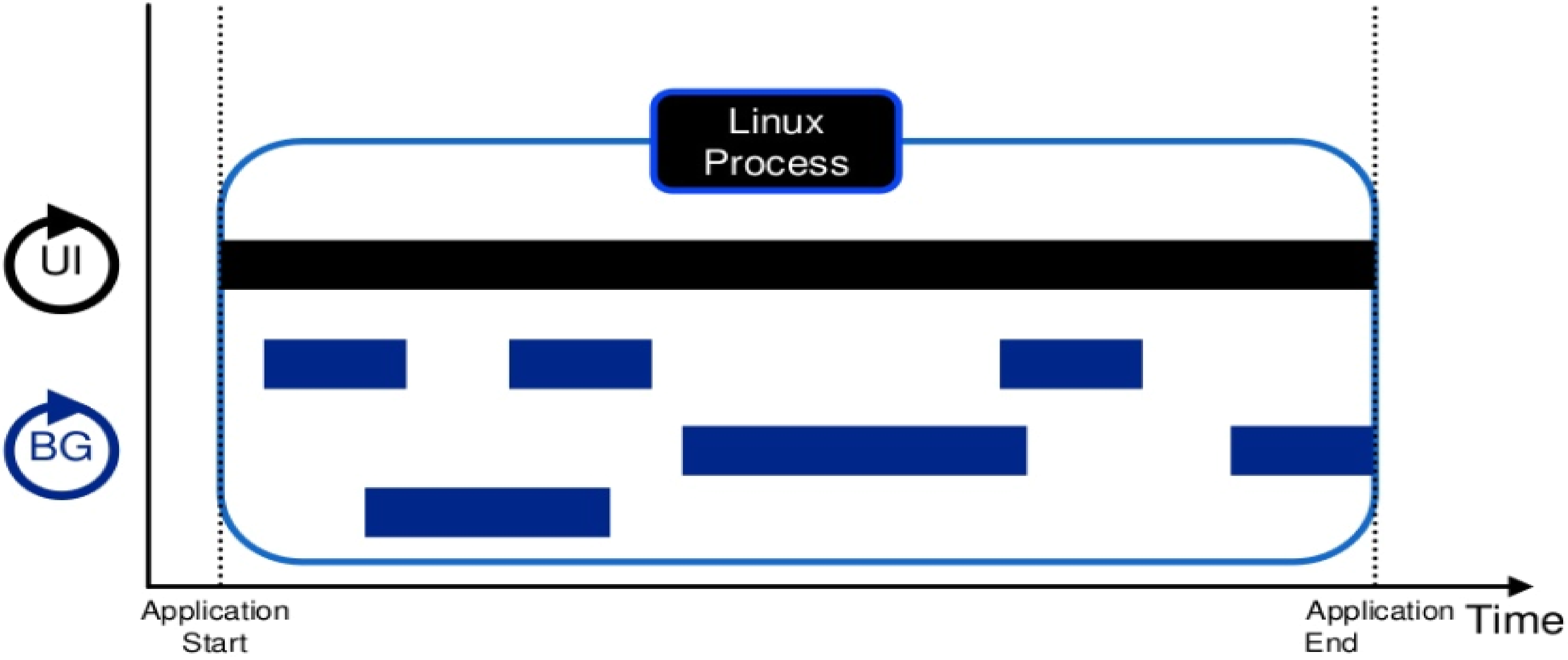
Kotlin Concurrency – Coroutines Basics

Flow

- **Asynchronous data stream** that sequentially **emits values**.
- Designed for handling streams of data asynchronously and reactively. Work with asynchronous data in a structured and declarative manner.

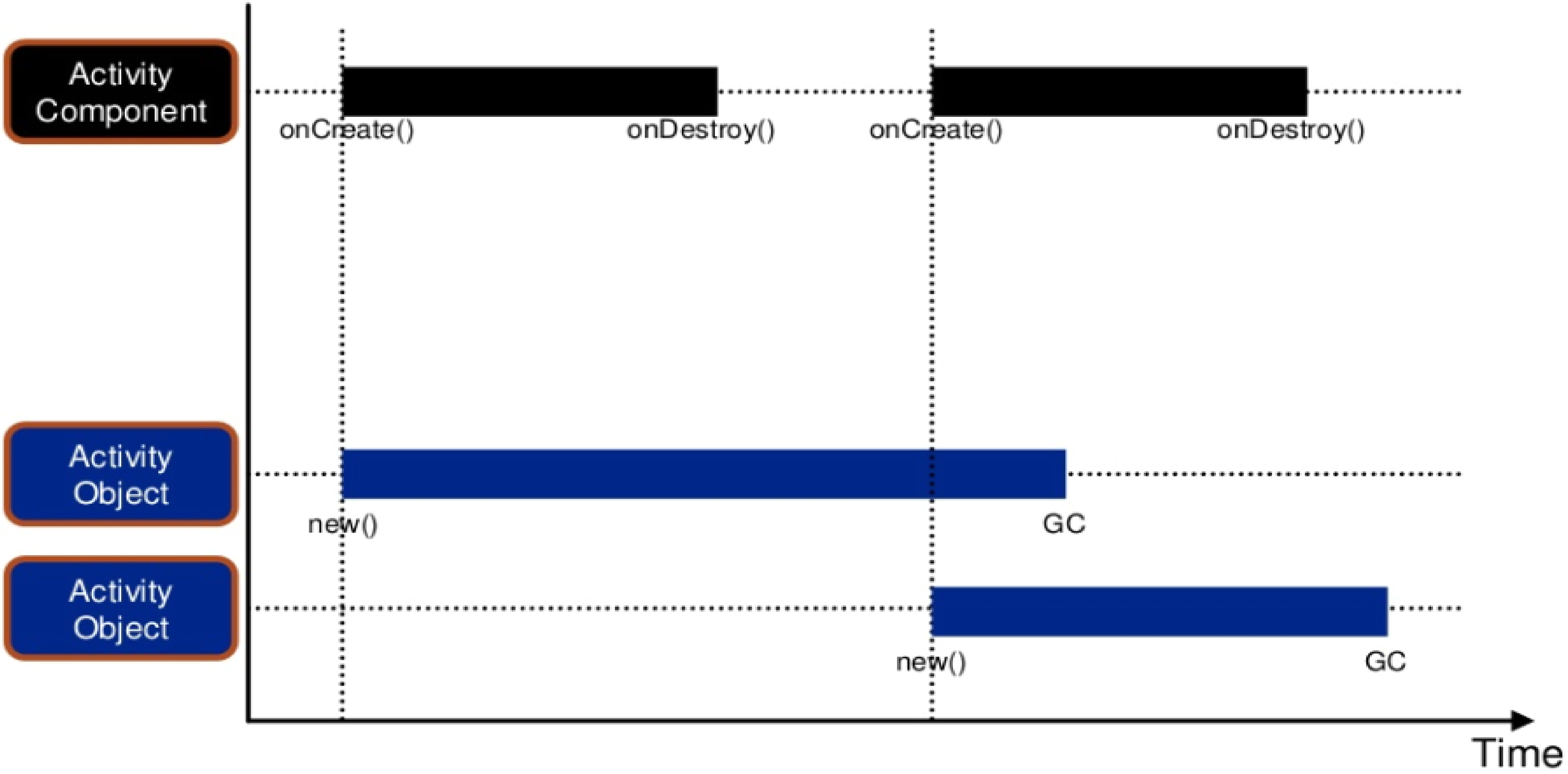
```
fun simpleFlow(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)          // Simulating a long-running operation
        emit(i)             // Emitting values from the flow
    }
}
GlobalScope.launch {        // Collecting values from the flow
    simpleFlow().collect { value ->
        println(value)
    }
}
```

Threading in Android - Memory



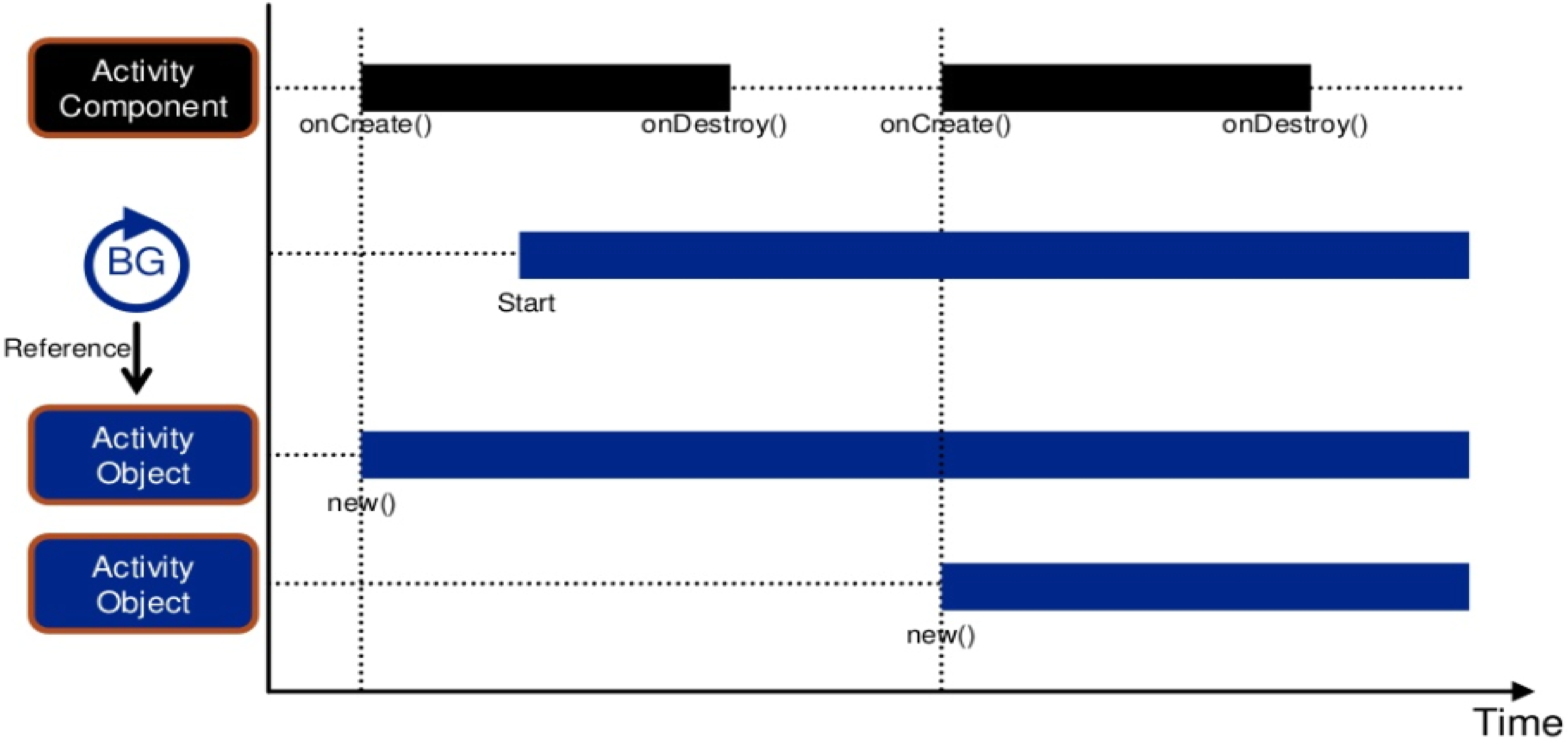
Source: http://www.slideshare.net/andersgoransson/efficient-android-threading#btnNext

Threading in Android - Memory



Source: http://www.slideshare.net/andersgoransson/efficient-android-threading#btnNext

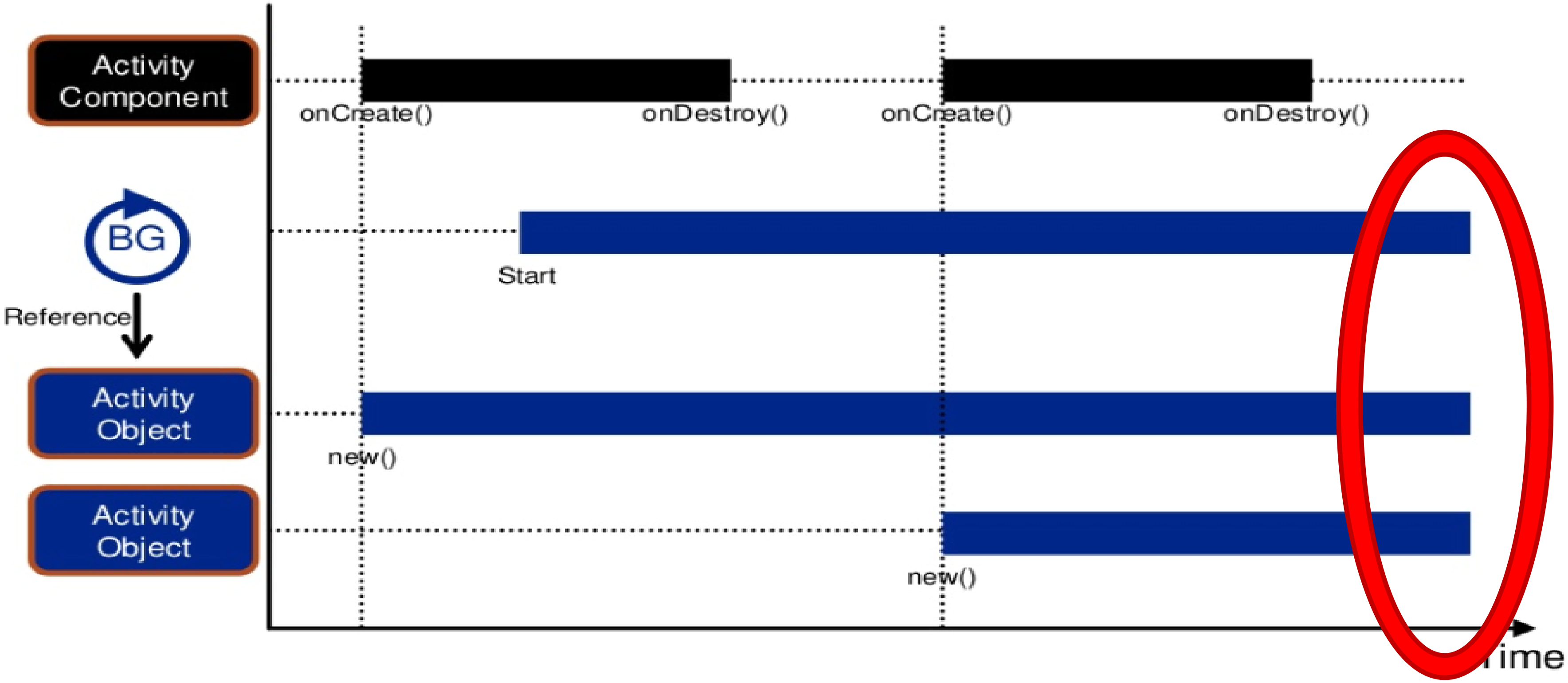
Threading in Android - Memory



Source: http://www.slideshare.net/andersgoransson/efficient-android-threading#btnNext

Threading in Android - Memory

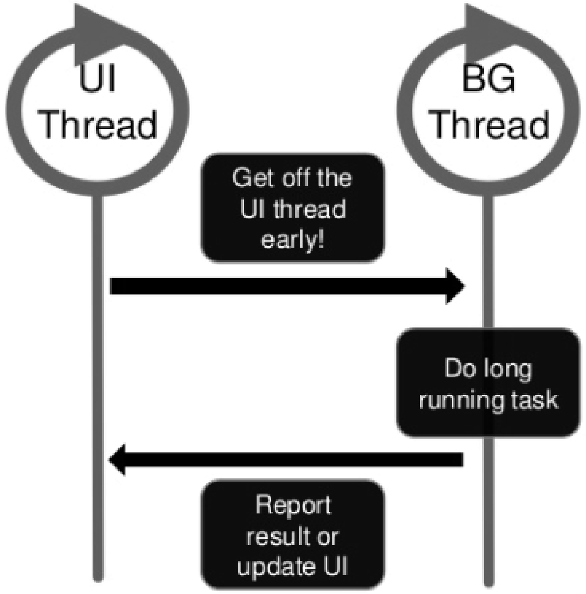
- **A thread is a Garbage Collector root**
  - Implement cancellation policy for your background threads



Source: http://www.slideshare.net/andersgoransson/efficient-android-threading#btnNext

Android Threading Techniques (Java)

- **Thread**
- **Executor**
- **HandlerThread**
- **AsyncTask**
- Service
- IntentService
- AsyncQueryHandler
- Loader



AsyncTask

- Enable proper and **easy use of the UI thread**.
- Allows to perform background operations and publish results on the UI thread without having to manipulate threads or handlers.
- Defined by a computation that runs on a background thread and whose result is published on the UI thread.
- Async task is defined by
  - 3 generic types – **Params, Progress, Result**
  - 4 main states – **onPreExecute, doInBackground, onProgressUpdate, onPostExecute**
  - 1 auxiliary method – **publishProgress**

AsyncTask States

- onPreExecute()
  - invoked on the **UI thread** before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
- doInBackground(Params...)
  - invoked on the **background thread** immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take a long.
- onProgressUpdate(Progress...)
  - invoked on the **UI thread**.This method is used to display any form of progress in the user interface while the background computation is still executing.
- onPostExecute(Result),
  - invoked on the **UI thread** after the background computation finishes. The result of the background computation is passed to this step as a parameter.

AsyncTask Example

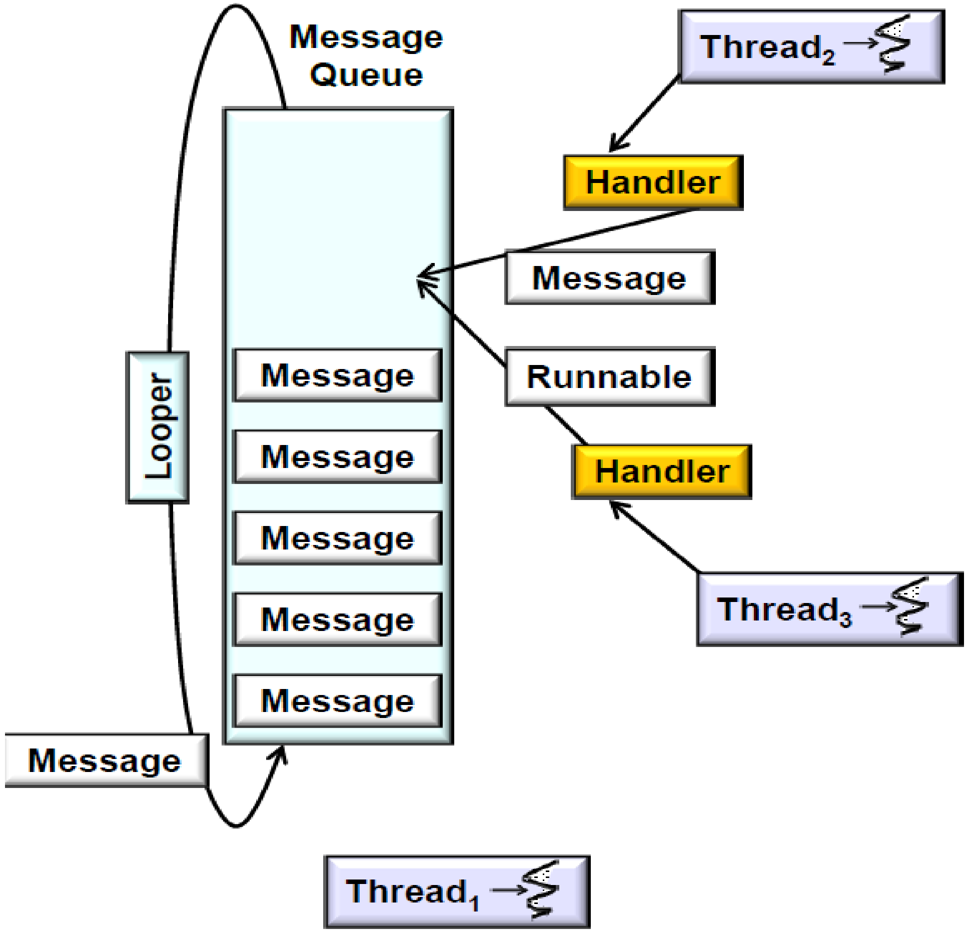
```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length; long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            if (isCancelled()) break; // Escape early if cancel() is called
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

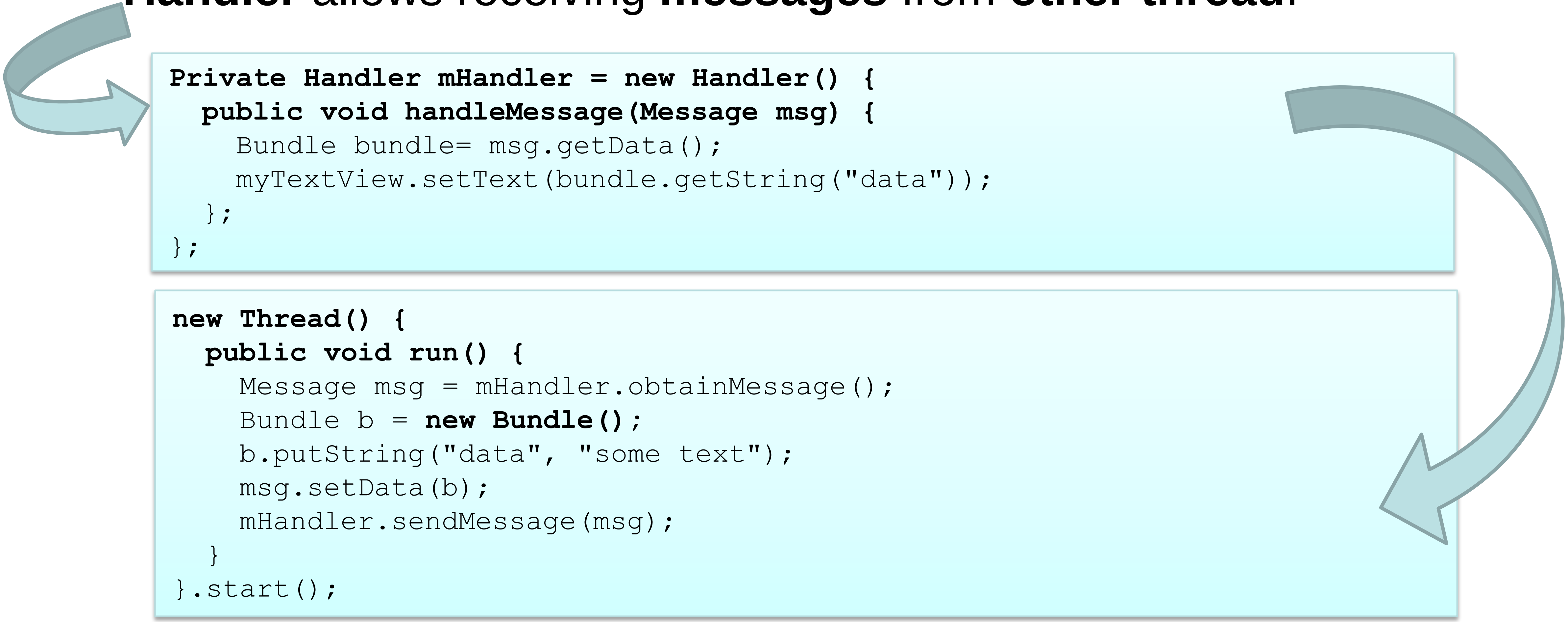
Handlers and Messages

- Allows an app to spawn threads that perform background operations and publish results on the UI thread
- A **Looper** provides a message queue to a thread
- Only **one Looper is allowed per Thread**
- **UI Thread has a Looper**



Handlers and Messages

- **Handler** allows receiving **messages** from **other thread**.



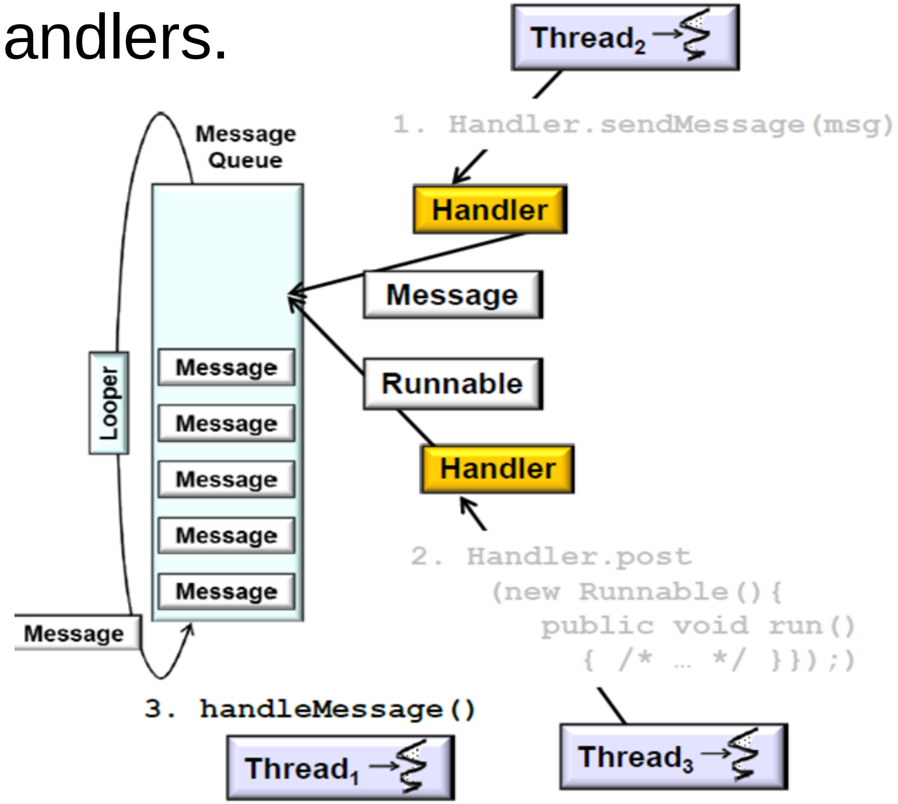
```
Private Handler mHandler = new Handler() {
  public void handleMessage(Message msg) {
    Bundle bundle= msg.getData();
    myTextView.setText(bundle.getString("data"));
  };
};
```

```
new Thread() {
  public void run() {
    Message msg = mHandler.obtainMessage();
    Bundle b = new Bundle();
    b.putString("data", "some text");
    msg.setData(b);
    mHandler.sendMessage(msg);
  }
}.start();
```

Handlers and Messages

- A Looper provides a message queue to a thread
- The **Looper.loop()** methods runs a Thread's main event loop.
- Waits for **Messages** and dispatches them to their Handlers.

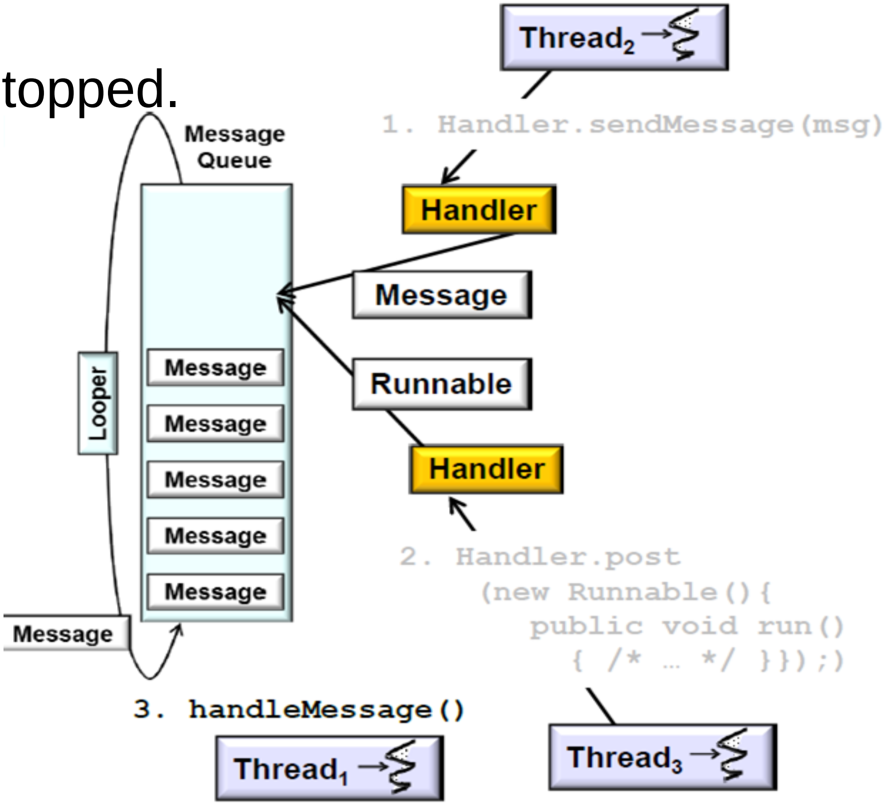
```
public class Looper {
    …
    final MessageQueue mQueue;
    public static void loop() {
    …
    while(true) {
        Message msg = queue.next();
        …
        msg.target.dispatchMessage(msg);
        …
    }
```



Handlers and Messages

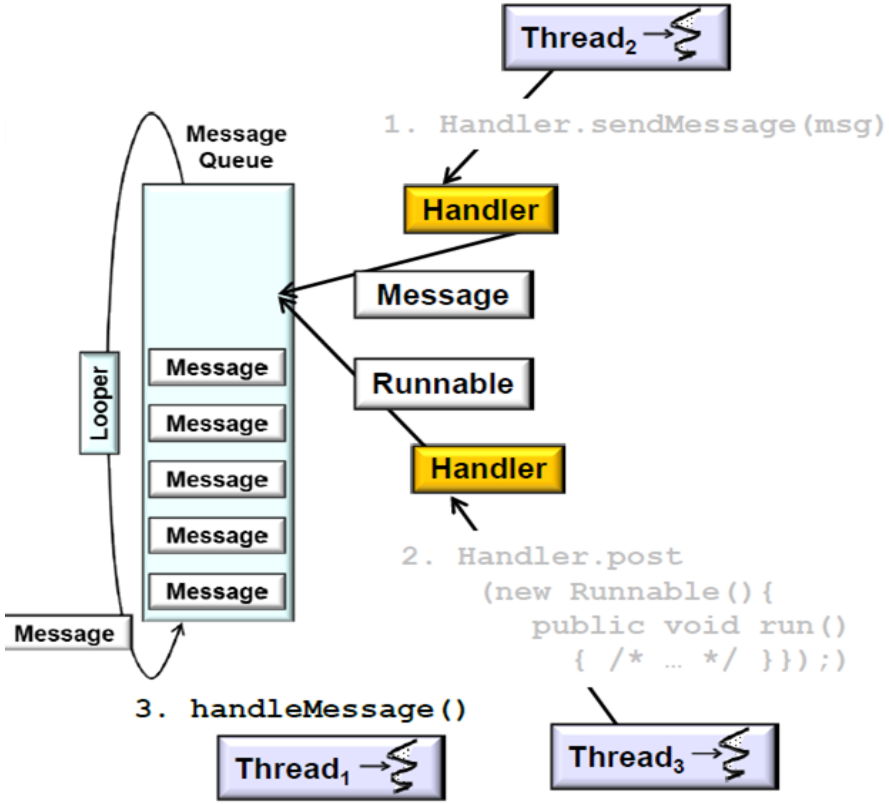
- By default Threads don’t have a message loop associated with them.
  - To create one, call prepare() in the thread that is to run loop
  - Create Handlers to process incoming messages
  - Call loop() to have it process messages until the loop is stopped.

```
public class myThread extends Thread {
    …
    public Handler mHandler;
    public void run() {
        Looper.prepare();
        mHandler = new Handler() {
                public void handleMessage(Message msg) {
                        // do something with msg
                }
        }
        Looper.loop();
}
```



Handlers and Messages

- Handler sends Messages and posts Runnables to a Thread
- Thread's MessageQueue enqueues and schedules them for future execution
- Implements **thread-safe processing of Messages**
  - In current Thread or different Thread



Service

- Component that can perform long-running operations in the background, and it doesn't provide a user interface.
- **Foreground**
  - Operation is noticeable to the user. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.
- **Background**
  - Operation that isn't directly noticed by the user.
    - API level 26 and higher imposes restrictions on running background services when the app itself isn't in the foreground.
- **Bound**
  - Client-server interface that allows components to interact with the service, send requests, receive results

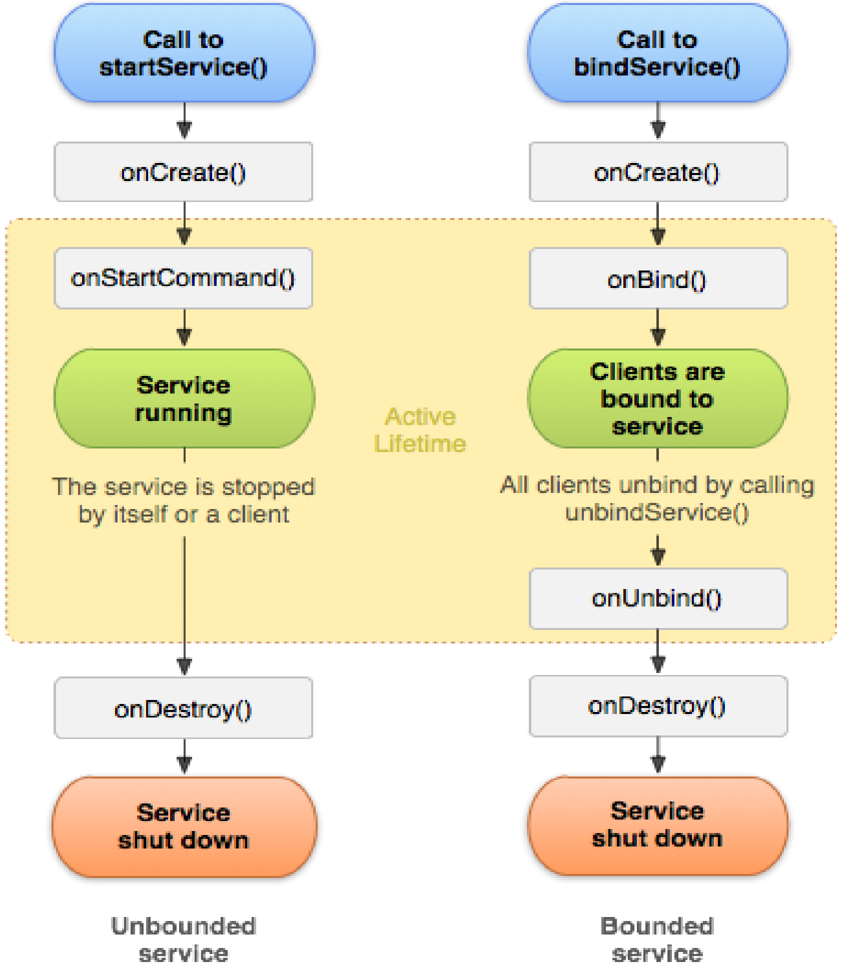
Service Life Cycle

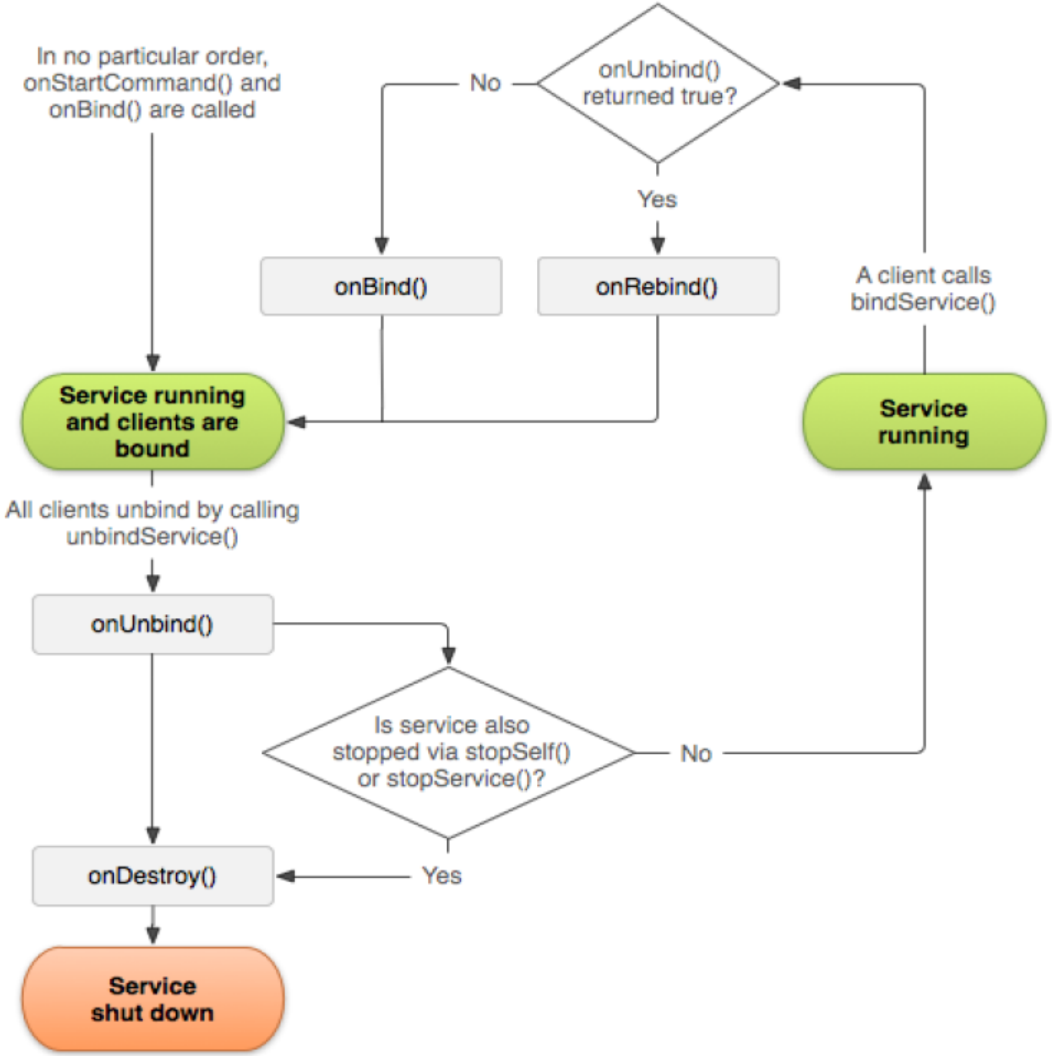
- The service is created when another component calls startService().

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

- The **service then runs indefinitely** and must stop itself by calling stopSelf().
  - Another component can also stop the service by calling stopService().
- Callbacks
  - **onCreate()** – called when Service is created (anyhow)
  - **onStartCommand()** – called after startService()
  - **onBind()** – after client calls bindService()
  - **onDestroy()** – Service will be removed

Service Life Cycle




In no particular order, onStartCommand() and onBind() are called
onUnbind() returned true?
No
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
Yes
onDestroy()
Service shut down

Start Service

```
public class MyService extends Service{
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
  //do something
  return START_STICKY;
}
```

- **START_STICKY**
  - If the system kills the service after onStartCommand() returns, recreate the service and call onStartCommand(), but do not redeliver the last intent.
- **START_NOT_STICKY**
  - If the system kills the service after onStartCommand() returns, do not recreate the service unless there are pending intents to deliver.
- **START_REDELIVER_INTENT**
  - If the system kills the service after onStartCommand() returns, recreate the service and call onStartCommand() with the last intent that was delivered to the service.

Binding Service to Activity

```
private final IBinder binder = new MyBinder();
Public class MyBinder extends Binder {
   MyService getService() {
      return MyService.this;
   }
}

@Override
public IBinder onBind(Intent intent) {
  return binder;
}
```

- **IBinder** defines the interface for communication with the service.
- **bindService()** retrieves the interface and begin calling methods on the service.

Binding Service to Activity

```
public class myActivity extends Activity {

  private MyService service;

  private Service Connection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, Ibinder service)
       service = ((MyService.MyBinder) service).getService();
  }

  public void onServiceDisconnected(ComponentName className) {
     service = null;
  }
};
```

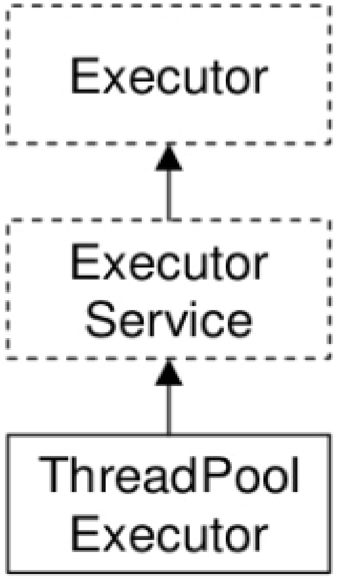
```
Intent i = new Intent(this, MyService.class);
bindService(i, mConnection, BIND_AUTO_CREATE);
```

Binding Service to Activity

- Interactions between Service and Activity
  - **Callback object or Listener interface**
  - **Broadcast Intents**
  - **Pending Intents**
    - `Context.startActivity(Intent)`
      `getActivity(context, requestCode, intent, flags)`
    - `Context.startService(Intent)`
      `getService(context, requestCode, intent, flags)`
    - `Context.sendBroadcast()`
      `getBroadcast(context, requestCode, intent, flags)`

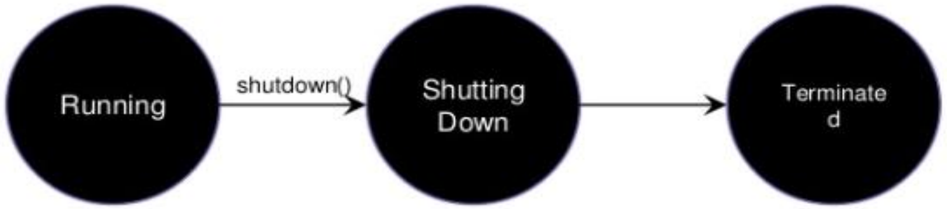
Executor

- An **Executor** that provides methods to manage termination and methods that can produce a **Future** for tracking progress of one or more asynchronous tasks.
- A **Future** represents the **result of an asynchronous computation**. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.



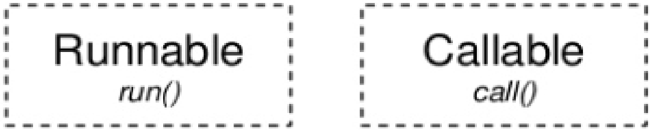
ExecutorService

- **Task submission**
  - executorService.submit(MyTask);
  - executorService.invokeAll(Collection<Tasks>);
  - executorService.invokeAny(Collection<Tasks>);
- **Lifecycle management**
  - executorService.shutdown();
  - executorService.shutdownNow();
- **Lifecycle observation**
  - executorService.isShutdown();
  - executorService.isTerminated();
  - executorService.awaitTermination();



Task / Execution Environment

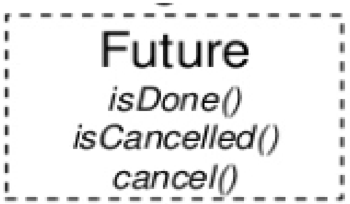
- **Task**
  - Independent unit of work executed anywhere



Runnable
run()

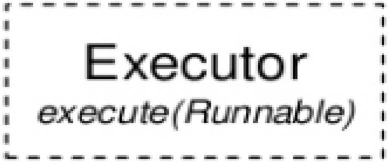
Callable
call()

- **Task manager/observer**



Future
isDone()
isCancelled()
cancel()

- **Execution Environment**
  - Technique used to execute the task



Executor
execute(Runnable)

Thread Pools

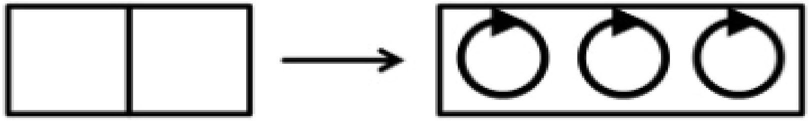
- **Execute tasks concurrently**
  - Multiple HTTP requests
  - Concurrent image processing
  - Use cases gaining performance from concurrent execution.
- **Lifecycle management** and observation of task execution.
- Maximum platform utilization

- Cons
  - Lost thread safety when switching from sequential to concurrent execution

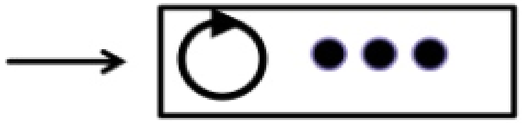
Thread Pools

- Executor managing a pool of threads and a work queue.
- Reduces overhead of thread creation.

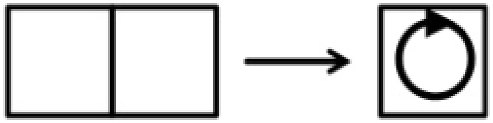
- **Fixed Thread Pool**



- **Cached Thread Pool**



- **Single Thread Pool**



- **Custom Thread Pool**

Thread Pools

- **Execute tasks concurrently**
  - Multiple HTTP requests
  - Concurrent image processing
  - Use cases gaining performance from concurrent execution.
- **Lifecycle management** and observation of task execution.
- Maximum platform utilization

- Cons
  - Lost thread safety when switching from sequential to concurrent execution

Resources

- https://kotlinlang.org/docs/coroutines-basics.html
- http://docs.oracle.com/javase/tutorial/essential/concurrency/
- http://www.slideshare.net/andersgoransson/efficient-android-threading#btnNext
- http://www.vogella.com/articles/AndroidPerformance/article.html
- http://www.d.umn.edu/~cprince/courses/cs5631spring07/lectures/JavaThreads.ppt
- Douglas C. Schmidt, Android Concurrency & Synchronization, https://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/Concurrency-and-Synchronization-parts1-2-and-3.pdf

Thank you for your attention

Mgr. Ing. Michal Krumnikl, Ph.D.

+420 597 325 867

michal.krumnikl@vsb.cz

www.vsb.cz