Multithreading

in

Android

Álvaro Velad Galván

Index

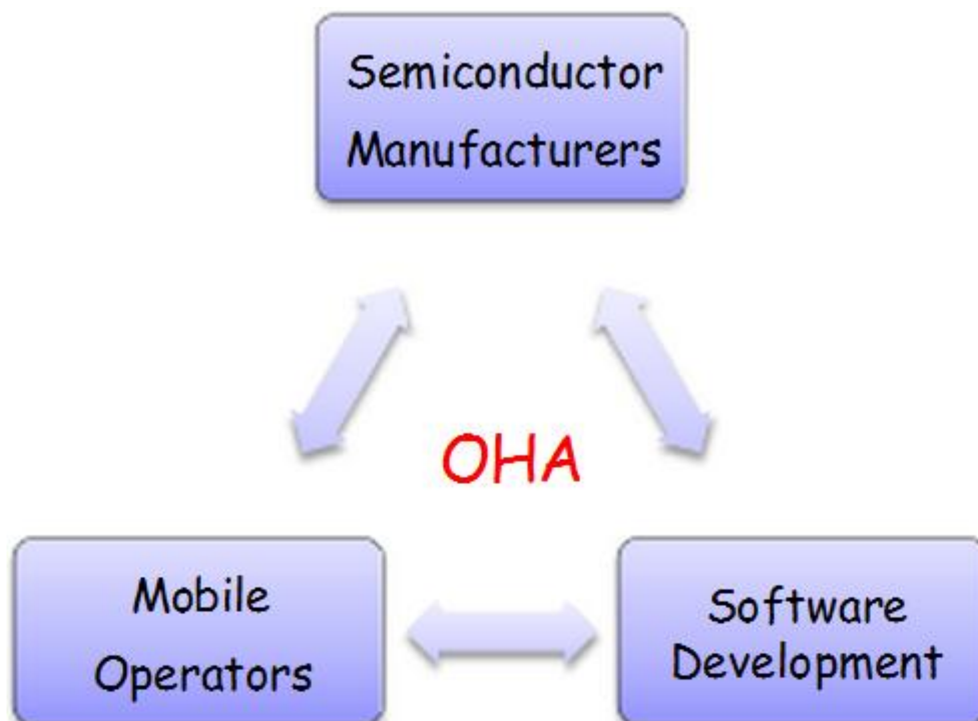
- Android Overview
- Android Stack
- Android Development Tools
- Main Building Blocks(Activity Life Cycle)
- Threading in Android
- Multithreading via AsyncTask Class
- Multithreading via Handler -> Message Passing & Post() method
- Multithreading via AsyncTask Class - Deepening

Android Overview

Awesome Ingredients:

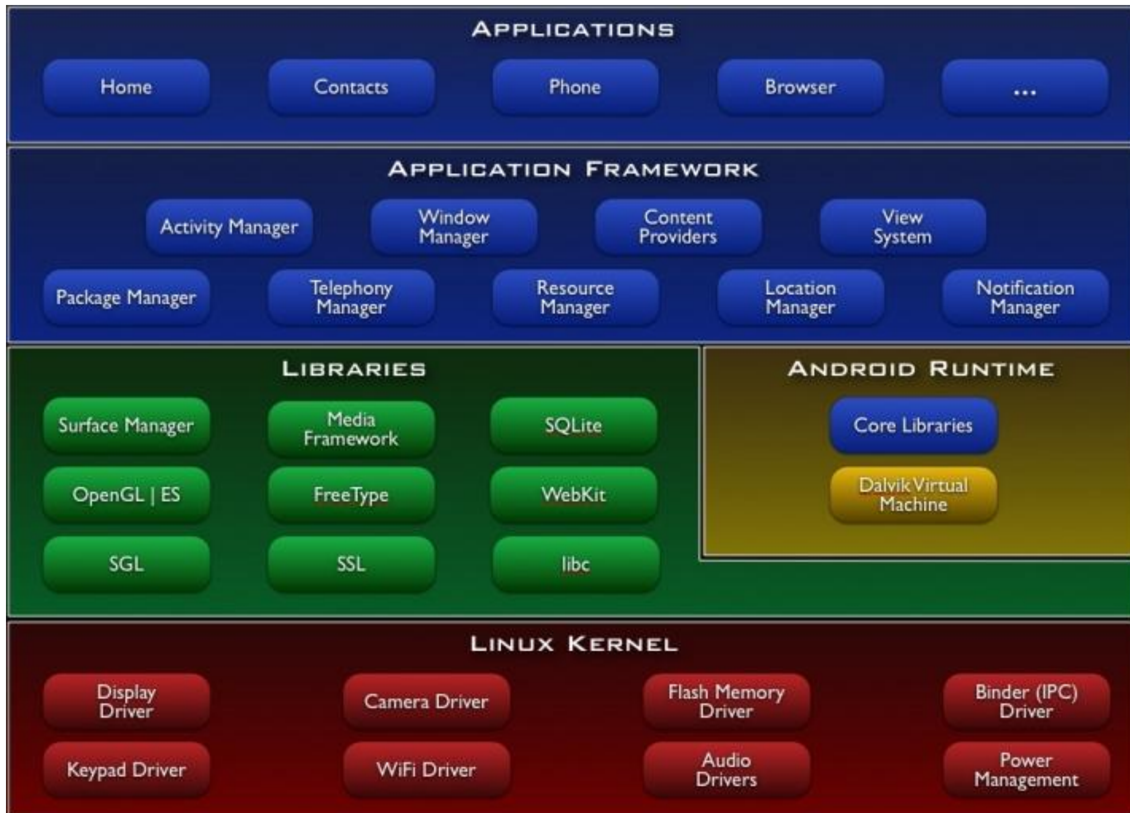
- Open Source Platform
  - Apache/MIT License
  - Third party libraries rewritten
- Designed for Mobile Devices
  - Limited Memory, Battery and Speed
- Comprehensive platform

Android is owned by Open Handset Alliance(OHA).



Android Stack Layers

- Linux Kernel
- Native Libraries
- Dalvik Runtime
- Application Framework



Linux kernel layer

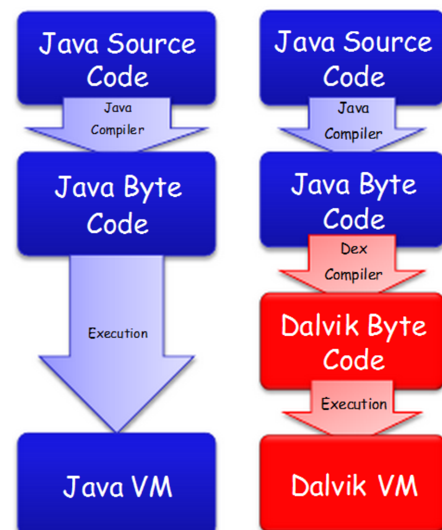
- Why linux kernel?
  - Portability
  - Security
  - Features
- Linux kernel 2.6.x or 3.x
- Kernel enhancements
  - Drivers, Pluggable modules, Power management, IPC

NativeLibraries

- Function Libraries
  - Webkit, Media framework, SQLite
- Bionic Libs
  - Optimized for embedded use:libc
- Native Servers
  - Surface flinger(manager), Audioflinger
- Hardware Abstraction Libraries(HAL)

Android RunTime

- Core Libraries
  - Mostly JavaSE(NoAWT&Swing)
  - More features
- DalvikVM
  - Low memory, slow cpu, no swap space



ApplicationFramework

- Java Libraries Built for Android
- Services(Managers)

Android Development

Development requirements

- Android SDK

http://developer.android.com/sdk/index.html

- Eclipse IDE for Java Developers

http://www.eclipse.org/downloads/packages/eclipse-classic-372/indigosr2

- Android Development Plugin

http://developer.android.com/sdk/eclipse-adt.html

IDE&Tools

- Android SDK
  - Class Library
  - Emulator and System Images
  - Documentation and Sample Code
  - Developer Tools
    - dx– Dalvik Cross - Assembler
    - aapt – Android Asset Packaging Tool
    - adb – Android Debug Bridge
    - ddms – Dalvik Debug Monitor Service
- Eclipse IDE + ADT plugin
  - UI creation easier, reduces development time

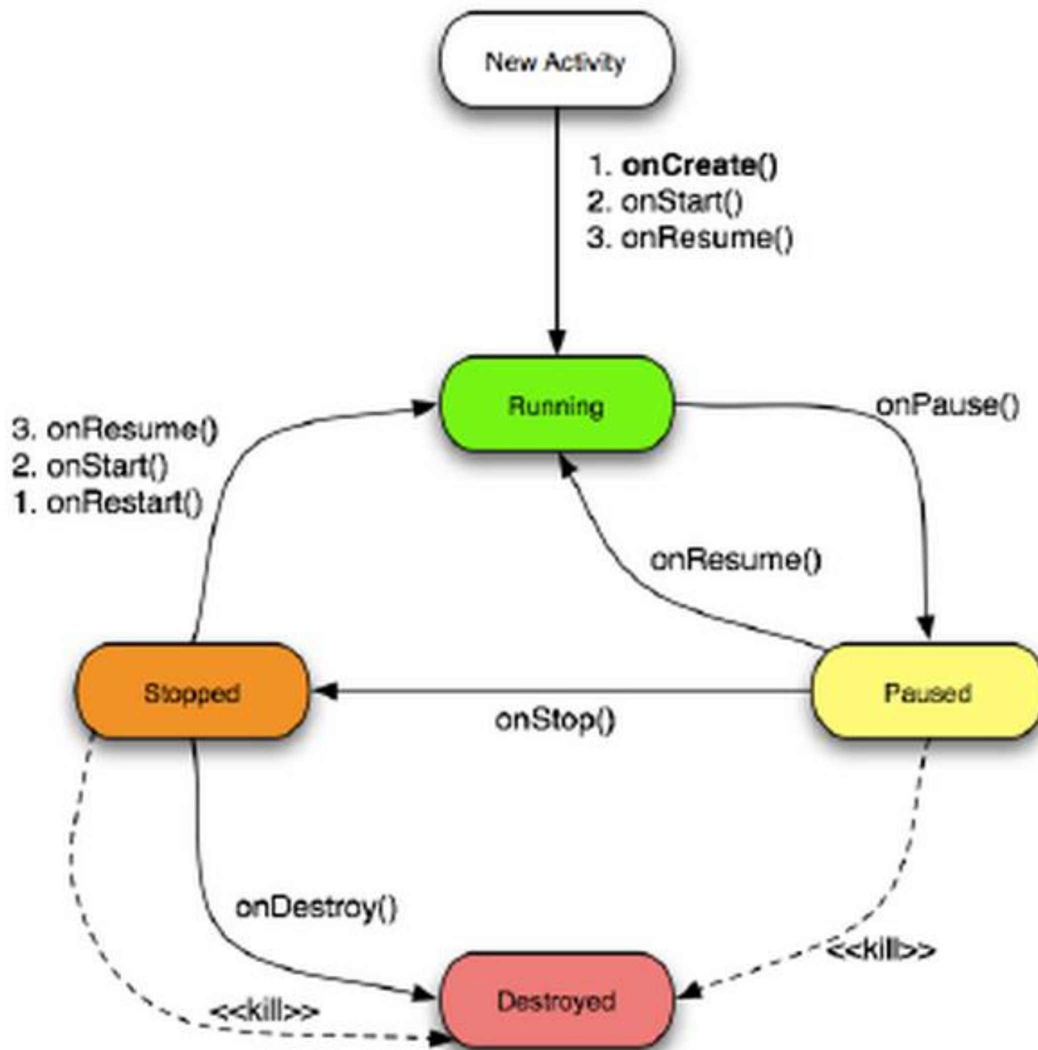
Programming Languages

- Java
- C/C++(Restricted official support)

  http://developer.android.com/sdk/ndk/index.html

Android Activity Life Cycle

- Activity -> GUI / Window on screen
- Activity Life Cycle



Threading in Android

- A Thread is a concurrent unit of execution.
- A thread has its own call stack for methods being invoked, their arguments and local variables.
- Each virtual machine instance has at least one main Thread running when it is started; typically, there are several others for housekeeping.
- The application might decide to launch additional Threads for specific.
- 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.
    - One is providing a new class that extends Thread and overriding its run() method.
    - The other is providing a new Thread instance with a Runnable object during its creation.
- In both cases, the start() method must be called to actually execute the new Thread.
- Threads share the process' resources but are able to execute independently.
- Applications responsibilities can be separated
    - Main thread runs UI, and
    - 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.

Multithreading Via AsycTask

Using the AsyncTask Class

- AsyncTask enables proper and easy use of the UI thread.
- This class allows performing background operations and publishing results on the UI thread without having to manipulate threads and/or handlers.
- An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.
- An asynchronous task is defined

3 Generic Types	4 Main States	1 Auxiliary Method
Params, Progress, Result	onPreExecute, doInBackground, onProgressUpdate onPostExecute.	publishProgress

Using the AsyncTask Class

```
private class VerySlowTask extends AsyncTask<String, Long, Void> {

    // Begin - can use UI thread here
    protected void onPreExecute() {

    }

    // this is the SLOW background thread taking care of heavy tasks
    // cannot directly change UI
    protected Void doInBackground(final String... args) {
    ... publishProgress((Long) someLongValue);
    }

    // periodic updates - it is OK to change UI
    @Override
    protected void onProgressUpdate(Long... value) {

    }

    // End - can use UI thread here
    protected void onPostExecute(final Void unused) {

    }
}
```

Multithreading Via Handler

Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue

- Each Handler instance is associated with a single thread and that thread's message queue
  - to schedule messages and runnables to be executed as some point in the future; and
  - to enqueuean action to be performed on a different thread than your own.
- Post(Runnable) & sendMessage(Message)

Multithreading via AsyncTask Class – Deepening

Android application follows single thread model i.e. when android application is launched, a thread is created for running that application. This single thread model works fine for normal execution, but for the instance like network call which are usually long operations, UI hangs waiting for the response from the server.

One of the methods that promptly come to our mind is to create a new thread and implements its run method for performing time consuming long operations.

```
public void onClick(View v) {
    Thread t = new Thread(){
    public void run(){
        // Long time comsuming operation
          }
   };
   t.start();
}
```

The above approach of creating new thread for performing time consuming long operations would have worked fine but since Android implements single thread model and Android UI toolkit is not thread safe i.e. UI must always be updated in UI thread, updating UI view at the end of long operation may lead to some issues like UI hangs.

Following are the various mechanisms provided by android via which we can access UI thread from other threads.

- Activity.runOnUiThread(Runnable)
- View.post(Runnable)

- View.postDelayed(Runnable, long)
- Handler

Let us look at each of them.

runOnUIThread

This activity method will run on UI thread. If the current thread is the UI thread, then the action is executed immediately. Or else the action is posted to the event queue of the UI thread.

```
public void onClick(View v) {
        Thread t = new Thread(){
                public void run(){
                        // Long time consuming operation
                        _activity.runOnUiThread(new Runnable() {

                                @Override
                                public void run() {
                                        _activity.setStatus("Long
Operation Completed");

                                }
                        });
                }
        };
        t.start();
}
```

post & postDelayed

These methods are of view and are use for updating the view. They place action (Runnable) in the message queue and this runnable action runs on UI thread.

post

```
public void onClick(View v) {
        // TODO Auto-generated method stub
        Thread t = new Thread(){
                @Override
                public void run() {
                        // Long time consuming operation
                        status.post(new Runnable() {

                                @Override
                                public void run() {
                                        status.setText("Long Operstion
Completed");
```

```
                                }
                        });
                }
        };
        t.start();
}
```

postDelayed

```
public void onClick(View v) {
        // TODO Auto-generated method stub
        Thread t = new Thread(){
                @Override
                public void run() {
                        // Long time consuming operation
                        status.postDelayed(new Runnable() {

                                @Override
                                public void run() {
                                        status.setText("Long Operstion
Completed");

                                }
                        }, 1000);
                }
        };
        t.start();
}
```

As we can see the difference between the post and postDelayed is that postDelayedaccepts one more parameter defining the delayed for the execution of action.

Handler

```
private Handler handler = new Handler() {
@Override
        public void handleMessage(Message msg) {
                // Code to process the response and update UI.
        }
};
```

Above code snippets shows mechanism to create Handler. Here we have to override method handleMessage for Handler which accepts object of Message as paarameter.

Now look at the code snippet for long operation and see how this Handler is called.

```
public void onClick(View v) {
        Thread t = new Thread(){
                public void run(){
                        // Long time comsuming operation
                        Message myMessage=new Message();
                        Bundle resBundle = new Bundle();
                        resBundle.putString("status", "SUCCESS");
                        myMessage.obj=resBundle;
                        handler.sendMessage(myMessage);
                }
        };
        t.start();
}
```

After the execution of long operation result is set in Message and passed to sendMessage of handler. Handler will extract the response from Message and will process and accordingly update the UI. Since Handler is part of main activity, UI thread will be responsible for updating the UI.

All the above said approaches enforces creation of thread and implementing its run() method. All this make code more complex and difficult to read. To simplify this process of performing long operation and updating UI, Android introduced AsyncTask.

AsyncTask

With AsyncTask android takes care of thread management leaving business login writing to us.

```
public class LongTimeConsumingOperation extends AsyncTask<String,
Void, String> {

        @Override
        protected String doInBackground(String... params) {
                // perform Long time consuming operation
                return null;
        }

        @Override
        protected void onPostExecute(String result) {
                // TODO Auto-generated method stub
                super.onPostExecute(result);
        }

        @Override
        protected void onPreExecute() {
                // TODO Auto-generated method stub
                super.onPreExecute();
```

```
        }

        @Override
        protected void onProgressUpdate(Void... values) {
                // TODO Auto-generated method stub
                super.onProgressUpdate(values);
        }

}
```

For executing AsyncTask, call execute method as shown in onClick method.

```
public void onClick(View v) {
        new LongTimeConsumingOperation().execute("");
}
```

Let us understand AsyncTask with the help of above example:

- Extend class AsyncTask and implements its methods.
- onPreExecute: This method is called before doInBackground method is called.
- doInBackground: Code to perform long operations goes here.
- onPostExecute: As the name suggest this method is called after doInBackground completes execution.
- onProgressUpdate: Calling publishProgress anytime from doInBackground call this method.

onPostExecute, onPreExecute and onProgressUpdate is optional and need not be overridden.

Also look at how we have extended LongTimeConsumingOperation withAsyncTask and understand AsyncTask generics.

AsynTask<String, void, String>

Things to note here are:

- String represents Params i.e the type of parameter doInBackground method will accept and also represents the type of parameter execute method will accept.

- void represent Progress i.e the parameter type for onProgressUpdate method

- String represents Result i.e the type of parameter accepted by onPostExecute method

Rules to Remember

While implementing AsyncTask we need to remember following rules:

- Task can be executed only once.
- Instance of AsyncTask needs to be created in UI thread. As shown above in onClick method a new instance of LongTimeConsumingOperation is created.
- execute method with parameters should be called from UI thread.
- Methods onPostExecute, onPreExecute and onProgressUpdate should not be called explicitly.

Conculsion

Through AsyncTask android provides an robust way of performing task and handling UI updating.