SERE 2013
Secure Android Programming: Best Practices for Data Safety & Reliability

Multithreading and Java Native Interface (JNI)

Rahul Murmuria, Prof. Angelos Stavrou
rmurmuri@gmu.edu, astavrou@gmu.edu

GEORGE MASON UNIVERSITY

Multi-core processors in Android devices

- Natural Progression:

  Servers -> Workstations -> Laptops -> Phones

- Multiple serial tasks can run in parallel

  For example: Each core can handle different tasks while rendering browser pages:

  - Execute JavaScript
  - Process network connections
  - Manage protocol stack or control services

You get concurrency for free!

Multi-core processors in Android devices

Cost of multi-threading:

- Does multi-threading always bring bugs?
  - Threading should not be an after-thought
- Are multi-threaded applications always power-inefficient?
  - Well-designed multi-threaded program gives flexibility to the operating system in managing energy better

Retrofitting concurrency is a bad idea.

GEORGE MASON UNIVERSITY

Multi-threading on Android

- Normal Java threading and synchronization support is available to the App developer
- Additionally Android SDK provides additional API support for threading for both Java applications and C/C++ native code

So what's different on Android, compared to traditional software?

Use-case determines choice of API

- Responsiveness in the UI
  - Example: Performing a network operation in background, leaving the UI thread for I/O
- Speed and efficiency of long-running operations
  - Example: Decoding multiple image files to show in tiled-view on a scrollable screen

We discuss two main scenarios.

Use-case determines choice of API

- Responsiveness in the UI
  - Example: Performing a network operation in background, leaving the UI thread for I/O
- Speed and efficiency of long-running operations
  - Example: Decoding multiple image files to show in tiled-view on a scrollable screen

We discuss two main scenarios; UI first.

Badly written Android Application

```
public class MainActivity extends Activity {
        private TextView view;
        …
        protected void onCreate(…) {
                super.onCreate(…);
                setContentView(… activity_main);
                …
                try { Thread.sleep(10000); }
                catch (…) { printStack(); }
                …
                view.setText(“how are you”);
                Log.v(MyAppName, “oncreate completed”);
}
```

What’s wrong with this code?

Removing the burden from UI Thread

```
public class MainActivity extends Activity {
        ViewSetterClass task;
        String mytext = “how are you”;
        …
        protected void onCreate(…) {
                super.onCreate(…);
                setContentView(… activity_main);
                …
                task = new ViewSetterClass(view);
                task.execute(mytext);
}
```

Continued…

Removing the burden from UI Thread

```
class ViewSetterClass extends AsyncTask<String, Void, String> {
        private TextView view;
        …
        protected String doInBackground(String… params) {
                / / params come from the execute() call in previous slide
                try { Thread.sleep(10000); }
                catch (…) { printStack(); }
                return params[0];
        }
        protected String onPostExecute(String mytext) {
                view.setText(mytext);
                Log.v(MyAppName, “oncreate completed”);
        }
}
```

UI Thread does not sleep during the 10 seconds.

Memory Analysis of a production code

- We will analyze “Image Downloader” Application from Android Developer’s Blog
  - http://android-developers.blogspot.com/2010/07/multithreading-for-performance.html
- Memory Analysis can be used to find out
  - Memory leaks
  - Duplicate Strings, Weak references, etc.

Why should we care about memory analysis?

Memory Structure in Android

- Lot of shared memory between processes
  - Physical RAM is mapped to multiple processes
  - Physical memory usage is not as relevant as the scaled reading based on ratio of number of processes accessing a given page in memory (Pss value)
- Some memory analysis tools:
  - adb shell dumpsys meminfo <process-name>
  - adb shell procrank
  - Eclipse Memory Analyser

Demo of memory analysis

- Let's try analysis a few applications using the memory analysis tools described in previous slide.

Threading data-intensive operations

- Multiple threads in an Android app using a thread pool object
  - You can also communicate between the threads
- Create a pool of threads:

```
private PhotoManager() {
    ...
    // Sets the amount of time an idle thread waits before terminating
    private static final int KEEP_ALIVE_TIME = 1;
    // Sets the Time Unit to seconds
    private static final TimeUnit KEEP_ALIVE_TIME_UNIT = TimeUnit.SECONDS;
    // Creates a thread pool manager
    mDecodeThreadPool = new ThreadPoolExecutor(
            NUMBER_OF_CORES,       // Initial pool size
            NUMBER_OF_CORES,       // Max pool size
            KEEP_ALIVE_TIME,
            KEEP_ALIVE_TIME_UNIT,
            mDecodeWorkQueue);
}
```

External Link to Sample Application

- For ThreadPool Google has a sample application at their developer website:
  - https://developer.android.com/training/multiple-threads/index.html

Motivation for Native Code

- Access Low-level OS features – ioctl, poll, etc.
- Explore advanced CPU features – NEON instruction set for signal and video processing
- Reuse large or legacy C/C++ programs
- Improve performance of computationally intensive operations
- OpenGL
- OpenSL ES

Multiple Ways to Program Native

- Using Java Application to present a UI to the user, and perform parts of logic in native code
  - Interfacing between Java and C is done using : Java Native Interface (JNI)
- Create purely native Activity with UI designed using OpenGL
  - Not common practice
- Android has a C Library called Bionic, custom built for use on mobile phones.

We focus on the first method using JNI

JNI Example – Step by Step

- Make new application called
  - Project: HelloJni
  - Package: edu.gmu.HelloJni
  - Activity Name: HelloJni
- The Java sources are under folder “HelloJni/src”
- Make new subdirectory in project folder called “jni”
  - i.e., HelloJni/jni
- In jni directory make new file called
  - MyHelloJni.cpp

JNI Example (p2)

- In this file, MyHelloJni.cpp, put

```
#include <string.h>
#include <jni.h>
extern "C" {
    JNIEXPORT jstring JNICALL
    Java_edu_gmu_HelloJni_HelloJniActivity_stringFromJNI
            ( JNIEnv* env, jobject thiz )
    {
        return env->NewStringUTF("Hello from JNI!");
    }
}
```

- Important: There is a logic to that complicated function name, and it is required to follow the convention.

JNI Example (p3)

- In HelloJni/jni make new file called Android.mk
- Put the following in Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE    := HelloJni
LOCAL_SRC_FILES := MyHelloJni.cpp

include $(BUILD_SHARED_LIBRARY)
```

- Note that LOCAL_MODULE defines the module name

JNI Example (p4)

- Build library
  - Open terminal.
  - “cd” to <workspace>/HelloJni/jni
  - Run build
    - <android-ndk-r7b>/ndk-build
- Check that libHelloJni.so is created

Java code compiles using the Android SDK

Native code compiles using the Android NDK

On startup – Working of JNI

- The JNI library is loaded when System.loadLibrary() is called.

- Every function in the native C code maps to a function declaration on the Java side.
  - The declarations are defined as “native”
  - public native int getNextFrame(*parameters*);

Demo! : Let’s look at some samples in code

Ref: http://developer.att.com/developer/forward.jsp?passedItemId=11900170

GEORGE MASON UNIVERSITY

Security implications of C code

- Java Virtual Machine (JVM) does a lot of work to make the Java code secure:
  - Protects against buffer overruns, and stack smashing attacks
  - It performs bounds checking on all arrays
- Jni code is a blackbox to the JVM
  - Native code also runs with same privileges as the Java code that spawned it, however, sandboxing is weaker

Other tips for Reliable Development

- If using pthreads in C for native threads, remember to detach each of the threads before exiting
- All arguments passed to and from the native code are local references to the JNI functions
  - There is API to define global references explicitly
- Make use of onPause/onResume to save or close resources that are not needed in the background
  - Specially useful if you have multiple threads, or content listeners which are not for other applications to use

Thank you!

Extra Slides

Tutorial Links

- **JNI**: http://marakana.com/s/post/1292/jni_reference_example
- **Multithreading**: http://developer.android.com/training/multiple-threads/index.html

In java HelloJni

- After public class HelloJniActivity extends Activity {
  - public native String stringFromJNI(); // the c++ function name
  - static {
  - System.loadLibrary("HelloJni"); // shared lib is called libHelloJni.so.
    - // this name is from the LOCAL_MODULE part of the Android.mk file
  - }
- In onCreate, after setContentView(R.layout.main); put
  - Log.e("debug","calling jni");
  - Log.e("debug",stringFromJNI()); // last part of name of c++ function
  - Log.e("Debug","done");

- Run and check log
- Note: public native … allows any function to be defined. But when this function is called, the shared library must have already been loaded (via System.loadLibrary)