Native Pthread on Android Platform using Android NDK

Bala Dhandayuthapani Veerasamy
Research Scholar
Manonmaniam Sundaranar University
Tirunelveli, Tamilnadu, India
dhanssoft@gmail.com

Dr. G. M. Nasira, PhD
Assistant Professor/Computer Science
Chikkanna Govt Arts College
Tirupur, Tamilnadu, India
nasiragm99@ yahoo.com

Abstract: JNI is a strong feature of the Java. JNI describe a way for controlling the actual code written in the Java programming language to work together with native code that is written in C/C++. JNI permits accurate methods of Java classes to be implemented natively and still be called and used as ordinary Java methods. The Android NDK libraries authorize us to implement fractions of our Android applications via native code such as C/C++. The Android NDK provides platform specific features and relies on JNI technology to glue the native code to the Android applications. The primary motivation for considering the use of Pthreads on mobile architecture can achieve optimum performances on multi-core mobile architectures. In order for a program to get benefit of Pthreads, it must be capable to be structured into separate, individual tasks that can perform concurrently. This research finding focuses on how android applications can facilitate Pthreads through android NDK that can adventure Pthreads to execute in hybrid mode with Java threads.

Key words: Android, Applications, Android NDK, Java, JNI, Pthread

1. INTRODUCTION

Android [14] Inc was originated in Silicon Valley, California in October 2003, with the thought of providing a mobile platform [14] that is more conscious of the user's location and preferences. Google purchased Android Inc in August 2005 as it entirely possessed subsidiary of Google Inc. Google main aim take place to present a fully open platform backed by Google technologies for application developers. In November 2007, the Open Handset Alliance [13], [14] was established as a consortium to develop an open standard for mobile devices. Open Handset Alliance began its expedition by announcing the Android platform.

1.1 What is an Android?

Android [14] is a mobile operating system (OS), is based on a personalized version of Linux. Google required Android to be open and free; therefore, generally the Android code was released under the open source Apache License. The main benefit of implementing Android is that it proposes a unified approach to application development. Developers need only to create for Android applications (Android apps), has to be run on several different devices, as long as the devices are using Android.

Google thoughts about enhancing model for little powered mobile devices. Mobile devices gap behind their desktop counterparts in memory and speed by five years. They also have restricted power for computation. The performance needs on mobile devices are consequence involving to optimize all. If we look at the list of packages in Android, we could see that they are fully featured and wide.

These issues led Google to repeat the standard Java Virtual Machine (JVM) implementation in a lot of compliments. The key figure in Google's implementation of this JVM is Dan Bornstein, who composes the Dalvik Virtual Machine (DVM) - Dalvik is the name of a town in Iceland. DVM [13], [14] obtains the produced Java class files and joins them into one or more Dalvik Executable (.dex) files. The goal of the DVM is to discover every likely way to optimize the JVM for space, performance and battery life. The final executable code in an Android, as an effect of the DVM is stand not on Java byte code but on .dex files as an alternative. This means we cannot exactly carry out Java byte code; we have to begin with Java class files and then exchange them to linkable .dex files.

1.2 Android Platform Architecture

Android [13]-[15] is more about using or developing Android apps for mobile devices than an Android OS. It is a mixture of tools and technologies that are optimized for mobile needs. Android depends on the proven Linux kernel in order to afford its OS functions. For the user application, Android depends on the JVM technology by employing the DVM. The Android OS is generally divided into five sections [13], [14].

Linux kernel: This is the bottom of layer. It is the kernel based on Android. This layer holds all the low level device drivers for the numerous hardware mechanism of an Android device.

Libraries: These contain all the code that provides the main features of an Android OS.

Android runtime: At the same layer as the libraries, the Android runtime afford a set of core libraries that permit developers to develop Android apps using Java. An Android runtime system contains DVM.

Application framework: Describes the ability of the Android OS to an application developer, can create use of them in their applications.

Applications: At this top layer, we will locate applications that we download and install from the Android play store. Any applications that we write are located at this layer.

1.3 Activity class

The Activity [13]-[15] base or super class describe a sequence of methods that manage the life cycle of an activity in every Android apps. The Activity class describes the following events.

- **onCreate()** executed while the activity is created at first.
- **onStart()** executed while the activity turns into visible to the user
- **onResume()** executed while the activity begin interacting with the user
- **onPause()** executed while the present activity is being break and the previous activity is being start again
- **onStop()** executed while the activity is no longer visible to the user
- **onDestroy()** executed before the activity is destroyed by the system either physically or by the system to preserve memory
- **onRestart()** executed while the activity has been stopped and is restarting again. By default, the activity created for us hold the onCreate() event. This event handler code assist to show UI elements of our screen

2. INSTALLING ESSENTIAL TOOLS

It is important to set up all the essential tools for Android programming, such as JDK, Android SDK, Eclipse ADT, Android NDK, Cygwin.

2.1 Installing the Android SDK

Successfully installing the Android Software Development Kit (Android SDK) [13]-[15] needs the Java Development Kit (JDK) as prerequisites. Android Developer Tools (ADT) is a list of plug-ins to mix together ADT into the Eclipse Integrated Development Environment (Eclipse IDE); The Android SDK is an inclusive list of development tools including Android platform Java libraries, an application packager, a debugger, an emulator and help. In order to do thing useful to ADT, the Android SDK needs to be installed. We can able to find ADT on the Android Developers web site at http://developer.android.com. Next, we must to install the ADT plug-in. To install new software, we have to open Eclipse and choose Help menu then Install New Software, in the Install dialog box, click the Add button to add a new source of plug-ins. Give it a name (e.g., Android) and supply the following URL: https://dl-ssl.google.com/android/eclipse/. This link should prompt Eclipse to download the list of plug-ins accessible from this site.

2.2 The Android Native Development Kit (Android NDK)

The Android NDK [9]-[11] is a companion tool set for the Android SDK, intended to allow developers to implement and implant performance significant section of their applications via native code. Although the Android framework is intended simply for Java-based applications, the NDK present the required tools to develop an Android apps using native code produced through C/ C++. Although the JNI native code run and are accessed faultlessly within the Java-based application while their implementations run as machine code and are not interpreted by the DVM.

The Android NDK is accessible for major OS [14]. The installation packages are accessible from the Android NDK web site at http://developer.android.com/sdk/ndk/index.html. The Android NDK is provided as a compressed ZIP archive file for the Windows platform. Download the installation package from the Android NDK web site (http://developer.android.com/sdk/ndk/index.html). Then right-click on it and choose Extract All… from the context menu. In Eclipse, launch the Preferences dialog by choosing Window → Preferences on Windows and Linux, or Eclipse → Preferences on Mac OS X. In the Preferences dialog, expand the Android category and select Native Development. Click the Browse button and choose the NDK location.

Installing the NDK on Microsoft Windows

The Android NDK [13], [14] was originally designed to work on Linux like systems. Some of the NDK mechanisms are shell programs and they are not able to run on the Microsoft Windows OS. While the most recent version of the Android NDK shows development in making itself more independent and self packaged, it still compel Cygwin to be installed on the host machine in order to fully operate. Cygwin is a Linux like environment and command-line interface for the Windows OS. It comes with base Linux applications, including a shell that permits running the Android NDK's build system. At the time of coding, the latest version of the Android NDK for Windows is r10d and it requires Cygwin 1.7 to be pre-installed on the host machine.

Installing Cygwin

To install Cygwin, we have to visit http://www.cygwin.com and the click on Install Cygwin. The installation page will make available link to the Cygwin installer, also known as setup.exe. Cygwin is not a single application; it is a large software distribution holding multiple applications. The Cygwin installer allows installing only the certain applications to the host machine.

3. PROBLEM STATEMENT

Android apps are naturally written [9]-[11] in Java. However, at these times we want to overcome the restrictions of Java such as memory organization and performance of programming frankly into Android native interface. Android provides Android NDK to carry out native development in C/C++, as well the Android SDK, which supports Java. The NDK supply all the tools (compilers, libraries and header files) to build applications that approach the device natively. Native code (in C/C++) is needed for high performance to conquer the restrictions in Java's memory organization and performance.

Using native machine code does not forever answer in an automatic performance boost. Even though the earlier versions of Java were known to be slower than native code, the latest Java technology is highly optimized and the speed alteration is negligible in many cases. Using native code in Android apps is absolutely not a bad practice. In certain cases, it happens to highly helpful because it can afford for use again and improve the performance of some difficult applications. An application depends on a set of libraries to attain their tasks. The Android NDK permits application developers to definitely put together use of any native library with their Java based Android apps. Without the Android NDK, these native libraries necessitate to be rewritten in Java in order to be used by the Android apps. The Android NDK encourages [9]-[11] reusing of non-Java based components within Android apps and makes possible the development process.

Concerning performance, as a platform-independent language, Java does not present any procedure for using the processors specific features for enhancing the code. Compared to desktop platforms, mobile device resources are highly inadequate. The Android NDK permits development of application components as native code in order to employ these CPU features.

A thread [13] is a mechanism allowing a single process to perform multiple tasks in parallel. Threads are lightweight process sharing the similar memory and resources of the similar parent process. A single process can maintain multiple threads perform them in parallel. As part of the similar process, threads can communicate with each other and share data. Android supports threads in both Java and the native code.

Android [14] holds the practice of the Thread class to carry out asynchronous processing. Android has provisions for Java Thread [7] class and java.util.concurrent [7] package to carry out background process using the ThreadPools and Executor classes. If we vital to revise the user interface from a new Thread, we require to synchronize with the user interface thread. Android offer extra constructs to handle concurrently in comparison with standard Java. We can employ the android.os.Handler [12] class or the AsyncTasks [12] classes. More advanced approach is established on the Loader [12] class, engaged Fragments and services. The Handler class can be second-hand to register to a thread and presents a simple channel to send data to this thread. A Handler [12] class object list itself with the thread in which it is produced.

The AsyncTask [12] class summarizes the background process and the synchronization with the main thread. It also holds up reporting advancement of the running tasks. An AsyncTask is underway using execute() method. The execute() method performs the doInBackground() and the onPostExecute() method. The doInBackground() method contain the coding, which should be carried out in a background thread. It runs automatically in a distinct Thread. The onPostExecute() method synchronizes itself once more with the user interface thread and permits it to be simplified. This method is executed by the framework once the doInBackground() method finishes. The Loader class permits us to load data asynchronously in an activity or fragment. They can observe the source of the data and bring new results when the content changes. They also persists data between configuration changes. If the result is recovered by the Loader after the object detached from its parent, it can cache the data. We can employ the abstract AsyncTaskLoader [12] class as the source for our own Loader implementations. The LoaderManager [12] of an activity or fragment manages one or more Loader instances.

Most of the Android library packages are the abstraction of Java library packages. Android platform utilizes the same library that Java has. Now a day's, mobile devices manufactured with multi-core processor architectures. In particular to multi-core architecture, threads are expected to utilize all the idle cores. Android support libraries do not have certain method to set affinity for thread to utilize multi-core processors such as AsyncTasks, Handler and Loader. On the other side, Java support flexible and easy use of threads; yet, Java libraries do not have certain methods for thread affinity to utilize the core processors.

The execution time of a parallel program depends on a variety of factors [7] jointly with the architecture of the execution platform, the compiler and OS employed, the parallel programming environment and the parallel programming model on which the environment is supported, as well as properties of the application program such as locality of memory references or dependencies between the computations to be carried out. There are performance analysis tools [7] on hand to enhancing an application's performance. They can support you in understanding what our program is really doing and suggest how program performance should be enhanced. Multi-core programming is deepest part of computing world. There are so many researches going on to improve speedup and performance of the program through parallel programs.

Even though, there are classes available to schedule threads through java.util.concurrent package [7], the performance of NativePthread that we shown in our earlier research was good. The Pthread [13] has affinity threads, which can schedule on specific core processors. In order to implement with dissimilar executable method, we proposed to encompass Android apps with native Pthread [13] through Android NDK. Hence, we can take advantages of Pthreads [13] into Android apps. One of the main advantages of using Pthreads [13] is, setting affinity for the threads to utilize multi-core processors. Our earlier researches concentrated on task parallelism; they are setting as affinity for task [6] , Parallel: One Time Pad using Java [5], JNT - Java Native Thread for Win32 Platform [12], Java Native Pthread for Win32 Platform [2], Java Native Intel Thread Building Blocks for Win32 Platform [14], Overall Aspects of Java Native Threads on Win32 Platform [15] , Performance Analysis of Java NativeThread and NativePthread on Win32 Platform [12] . Our previous research experiences assist us to facilitate native Pthreads through Android NDK for Android apps which can exploit in Java threads and native Pthreads to execute in hybrid mode in Android apps.

4. METHODOLOGIES

JNI allows us to employ native code when an application cannot be written completely in the Java. It wants to implements difficult code in a lower-level language. It needs the platform features not carried in Java class library. In order to create and work with native program using Android NDK [9]-[11] application that calls a C++ functions with the following steps:

1. Write an Android JNI program
2. Generating C/C++ Header File using "javah" Utility
3. C Implementation - NativePthread.cpp
4. Create an Android makefile - Android.mk and Application.mk
5. Build NDK
6. Run the Android App

4.1 Write an Android JNI program

We make an activity that describe a native method to obtain a create Pthread, get thread Id and set affinity for thread. The Pthread execution results will be displayed as string on a TextView. Create an Android project called "NativePthread", with application name "NativePthread" and package "com.example.nativepthread". Create an activity called "NativePthread" with Layout name "activity_main" and Title "NativePthread". Replaced the "NativePthread.java" as follows.

Program 1. NativePthread.java

```
package com.example.nativepthread;
import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.TextView;

public class NativePthread extends Activity {
          TextView tv;
   long startTime,stopTime,elapsedTime;
   int i=0;

   @Override
   protected void onCreate(Bundle
savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      tv=(TextView)findViewById(R.id.textView1);
   }
   @Override
   public boolean onCreateOptionsMenu(Menu
menu) {
       getMenuInflater().inflate(R.menu.main, menu);
       return true;
   }
   public void clickStart(View v){
           setPthreadAffinityMask(0);
           createPthread();
   }
   public native void createPthread();
   public native int getCurrentThreadId();
   public native void setPthreadAffinityMask(final
int mask);

   public void executeNativePthread(){
   try{
     startTime = System.currentTimeMillis();
     for(i=0;i<100;i++){
      stopTime = System.currentTimeMillis();
      elapsedTime = (stopTime - startTime);
      tv.post(new Runnable(){
         public void run(){
          tv.setText("NativePthread-ID\t"
            +getCurrentThreadId() + "\tValue " + i + "
                          used ms " + elapsedTime);
                             }
                      });
             }
}catch(Exception e){       System.out.println("Error
in NativePthread"+e); }
   }

   static {
           System.loadLibrary("NativePthread");
   }
}
```

This JNI program uses a static initializer to load a shared library "libNativePthread.so" in Linux. It declares a native method called createPthread(), which create Pthread by calling executeNativePthread() and returns a String to be as the TextView's message. The onCreate() method already declared with a TextView. The NativePthread extends Activity class contain following native methods

```
public native void createPthread();
public native int getCurrentThreadId();
public native void setPthreadAffinityMask(final int mask);
```

The native keyword indicates that the method is implemented natively. Although the virtual machine now knows that the method is implemented natively, it still does not know where to find the implementation. These native methods are compiled into a shared library. This shared library needs to be loaded first for the virtual machine to find the native method implementations. The java.lang.System [9]-[11] class provides the loadLibrary method for Java applications to load shared libraries during runtime. Assuming that the native method is compiled into a shared library called libNativePthread.so, the following method call should be added to the code.

```
static {
System.loadLibrary(“NativePthread”);
}
```

The loadLibrary method is called within the static context, because we would like to have it loaded only once during the virtual machine's lifetime. The layout the Activity will use is.

Program 2. activity_main.xml

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/
android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
android:paddingBottom="@dimen/activity_vertical_
margin"
android:paddingLeft="@dimen/activity_horizontal_
margin"
android:paddingRight="@dimen/activity_horizontal_
margin"
android:paddingTop="@dimen/activity_vertical_mar
gin"
 tools:context=".MainActivity" >

   <TextView
     android:id="@+id/textView1"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/hello_world" />

   <Button
     android:id="@+id/button1"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_below="@+id/textView1"
     android:layout_centerHorizontal="true"
     android:layout_marginTop="100dp"
     android:text="@string/start"
     android:onClick="clickStart"/>
</RelativeLayout>
```

4.2 Generating C/C++ Header File using "javah" Utility

The javah tool produces C header and source files, which are necessary to apply in native methods. It takes the compiled class files and parses them for native methods and generates the necessary header and source files. Although this can be achieved without using the javah tool, it makes the process more robust and much easier. It is most often used tools during native development.

Create a folder "jni" below the project's root folder by right-clicking on the project → New → Folder. Then run "javah" utility from a Command prompt to create C/C++ header called "NativePthread.h".

```
> javah -classpath
../../bin/classes;<ANDROID_SDK_HOME>\platforms\android-<xx>\android.jar
   -o NativePthread.h
com.example.nativepthread.NativePthread
```

The header file contains a function prototype shown in the following program.

Program 3. NativePthread.h

```
/* DO NOT EDIT THIS FILE - it is machine
generated */
#include <jni.h>
/* Header for class
com_example_nativepthread_NativePthread */

JNIEXPORT void JNICALL
Java_com_example_nativepthread_NativePthread_createPthread
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL
Java_com_example_nativepthread_NativePthread_getCurrentThreadId
  (JNIEnv *, jobject);

JNIEXPORT void JNICALL
Java_com_example_nativepthread_NativePthread_setPthreadAffinityMask
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
#endif
#endif
```

The header file first includes the jni.h header file This header file contains definitions of JNI data types and functions. The native function takes two parameters, the first parameter, JNIEnv, is an interface pointer that points to a function table of available JNI functions. The JNIEnv interface pointer is always provided with each native function call. The second parameter can either be an object reference for member methods or a class reference for static methods. Using the automatically generated header file, we will provide the native implementation in a C/C++ source file.

4.3 C Implementation - NativePthread.cpp

If we are familiar with JNI, the Android NDK is greatly based on JNI concepts. It is basically JNI with a limited set of headers for CPP compilation. Create the following CPP program called "NativePthread.cpp" under the "jni" directory by right-clicking on the "jni" folder → New → File.

Program 4. NativePthread.cpp

```
#include <jni.h>
#include "NativePthread.h"
#include <sys/syscall.h>
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#include <unistd.h>

JavaVM* javaVM = NULL;
JNIEnv* env=0;
jclass aThreadCls;
jobject aThreadObj;

 int gettid() {
        pthread_t ptid = pthread_self();
        int threadId = 0;
        memcpy(&threadId, &ptid,sizeof(ptid));
        return threadId;
 }

 JNIEXPORT jint JNICALL
JNI_OnLoad(JavaVM* vm, void * reserved)  {
        javaVM = vm;
        return JNI_VERSION_1_6;
 }

 void JNI_OnUnload(JavaVM *vm, void
*reserved)  {
        javaVM = NULL;
 }

 void *NativePthread(void *argv){
         int res = javaVM-
        >AttachCurrentThread(&env, NULL);
         jclass cls = env->
        GetObjectClass(aThreadObj);
         jmethodID method = env-
        >GetMethodID(aThreadCls,
        "executeNativePthread", "()V");
         env->CallVoidMethod(aThreadObj,
        method);
```

```
            javaVM->DetachCurrentThread();
            pthread_exit(NULL);
            return NULL;
 }

JNIEXPORT void JNICALL
Java_com_example_nativepthread_NativePthread_cr
eatePthread
  (JNIEnv *env, jobject obj){
            env->GetJavaVM(&javaVM);
            jclass cls = env->GetObjectClass(obj);
            aThreadCls = (jclass) env-
            >NewGlobalRef(cls);
            aThreadObj = env->NewGlobalRef(obj);
            int pt;
            pthread_t Pthread;
            pt =  pthread_create(&Pthread, NULL,
            &NativePthread, NULL);
            pthread_join(Pthread, NULL);
            if(pt != 0) {
                        printf("Error:
            NativePthread.createPthread() method
            failed\n");
            }
}
JNIEXPORT jint JNICALL
Java_com_example_nativepthread_NativePthread
_getCurrentThreadId
  (JNIEnv *, jobject){
            return pthread_self();
}
JNIEXPORT void JNICALL
Java_com_example_nativepthread_NativePthread_se
tPthreadAffinityMask
 (JNIEnv *env, jobject obj, jint mask){
          cpu_set_t cpuset;
          CPU_ZERO(&cpuset);
          CPU_SET(mask, &cpuset); //assign
          sched_setaffinity(gettid(), sizeof(cpuset),
&cpuset);
   }
```

The JVM [13], [14], [4] offers the "invocation interface" functions, which permit us to make and demolish a JavaVM. In theory we can have multiple JavaVM per process, but Android only permits only one JVM. The JNIEnv [13], [14], [4] presents most of the JNI functions. In our native functions all receives from JNIEnv as the first argument. The JNIEnv is exploit for thread-local storage. This is the reason, why we cannot distribute a JNIEnv between threads. If a part of code retains no other way to acquire JNIEnv, we should distribute the JavaVM and employ GetEnv to find out the thread's JNIEnv. The C statements of JNIEnv and JavaVM are dissimilar from the C++ statements. The "jni.h" include file presents different type definition depending on whether it's incorporated into C or C++. For this reason it's a wicked idea to comprise JNIEnv arguments in header files incorporated by both languages. All threads are Linux threads, arranged by the kernel.

Thread begun with pthread_create [13], [14] can be connecting either through JNI AttachCurrentThread [13], [14] or through AttachCurrentThreadAsDaemon functions. Until a thread is put together, it has no JNIEnv and cannot make JNI calls. Android does not delay threads completing native code. If garbage collection is in move forward or the debugger has issued a delay request, Android will be paused the thread the next time it makes a JNI call. Threads connected through JNI should be called DetachCurrentThread [13]-[15] before terminated.

A JNIEXPORT void JNICALL Java_com_example_nativepthread_NativePthread_createPthread (JNIEnv *, jobject) method creates thread by calling pthread_create(&Pthread, NULL, &NativePthread, NULL). The parameter &Pthread initializes the pthread and &NativePthread parameter will call void *NativePthread (void *argv), the env->GetMethodID(aThreadCls, " executeNativePthread", "()V") in void *NativePthread (void *argv), is used to call public void executeNativePthread() method (program 1) though its signature "()V". Hence, this method can have parallel code implementation at Java application program side, which will be called by native method and link with windows kernel to create thread. Once NativePthread is created, it will start running public void executeNativePthread() method immediately from android program.

A JNIEXPORT jint JNICALL Java_com_example_nativepthread_NativePthread_getCurrentThreadId (JNIEnv *, jobject) calls getid() method to get current threadid using pthread_self(). JNIEXPORT void JNICALL Java_com_example_nativepthread_NativePthread_setPthreadAffinityMask (JNIEnv *, jobject, jint) calls sched_setaffinity(gettid(), sizeof(cpuset), &cpuset) to set the affinity for the current thread.

4.4 Create an Android makefile - Android.mk and Application.mk

Although the project contains the native code, ADT will not be competent to run it. We need to first add native code support to the project to allow building the native code as a part of the Android apps build process. Right-click the project and choose Android Tools ➢ Add Native Support from the context menu. It creates an Android makefile called "Android.mk" [13]-[15] below the "jni" folder as follows.

Program 5. Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE    := NativePthread
LOCAL_SRC_FILES := NativePthread.cpp
include $(BUILD_SHARED_LIBRARY)
include $(BUILD_SHARED_LIBRARY)
```

Create an Android makefile called "Application.mk" [13]-[15] under the "jni" directory by right-clicking on "jni" folder → New → File, as follows.

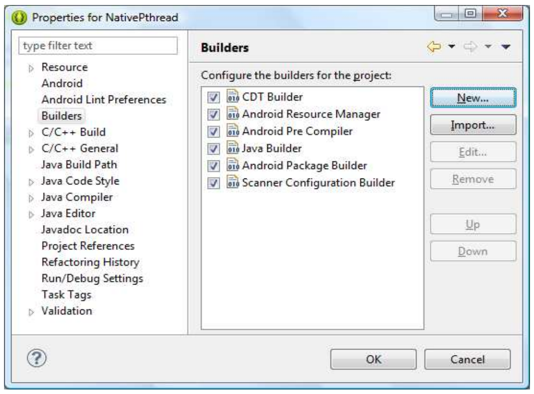
Program 6. Application.mk

```
APP_ABI := all
APP_PLATFORM=android-21
```

In the above makefile, "NativePthread" is the name of our shared library (used in System.loadLibrary()) and "NativePthread.cpp" is the source file.

4.5 Build NDK

To edit native JNI code in an Android project using the Android NDK require to configure [13]-[15] Eclipse in our project by editing native code as it does for java. The below steps shows, how to perform the essential configuration? Start by right clicking on our android project with JNI resources and select Properties. In the dialog, choose the Builders list to the left and press the New... button.


Properties for NativePthread

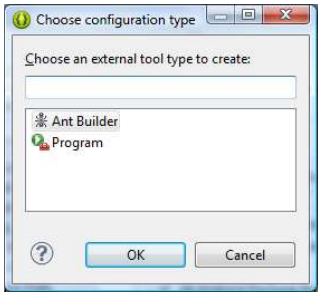
Builders

Configure the builders for the project:

☑ CDT Builder
☑ Android Resource Manager
☑ Android Pre Compiler
☑ Java Builder
☑ Android Package Builder
☑ Scanner Configuration Builder

Fig. 1. Project Properties

A new dialog will open displaying a list of builder types. Choose the Program and then press OK.

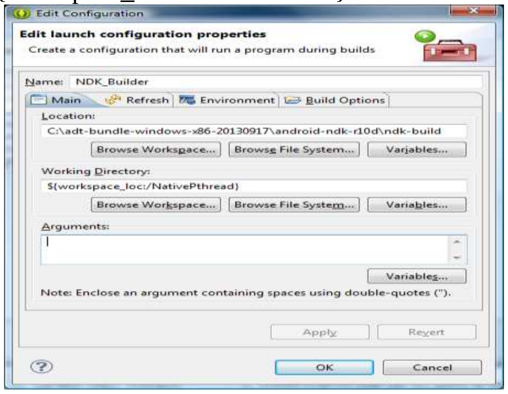

Choose configuration type

Choose an external tool type to create:

Ant Builder
Program

Fig. 2. Configuration Type

In the Main tab, we should fill in the following.
Name: NDK Builder
Location: ../android-ndk/ndk-build
Working Directory: ${workspace_loc:/NativePthread}


Edit Configuration

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: NDK_Builder

Location:
C:\adt-bundle-windows-x86-20130917\android-ndk-r10d\ndk-build

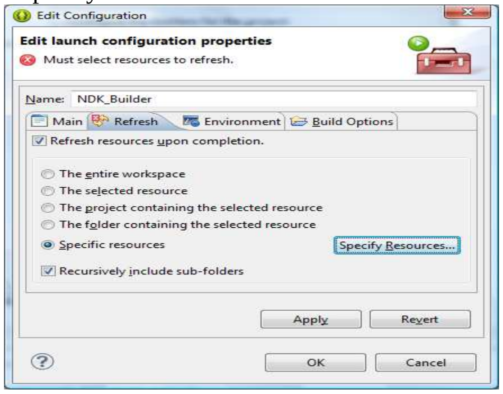
Working Directory:
${workspace_loc:/NativePthread}

Arguments:

Note: Enclose an argument containing spaces using double-quotes (").

Fig. 3. NDK Builder

Now carry on with the refresh tab. Select the checkbox with Refresh resources upon completion. Choose the Specific resources radio button and press the Specify Resources... button.


Edit Configuration

Edit launch configuration properties

Must select resources to refresh.

Name: NDK_Builder

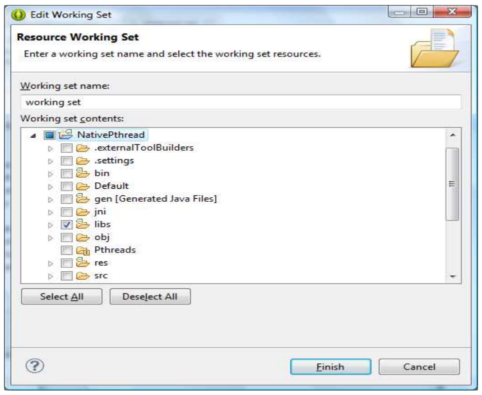
☑ Refresh resources upon completion.

○ The entire workspace
○ The selected resource
○ The project containing the selected resource
○ The folder containing the selected resource
● Specific resources

☑ Recursively include sub-folders

Fig. 4. Specify Library Resources

Since the ndk-build procedure will produce files in the lib folder, we desire Eclipse to determine alteration made there without having to refresh manually. Select the libs folder in the project and press the Finish button.



Edit Working Set

Resource Working Set

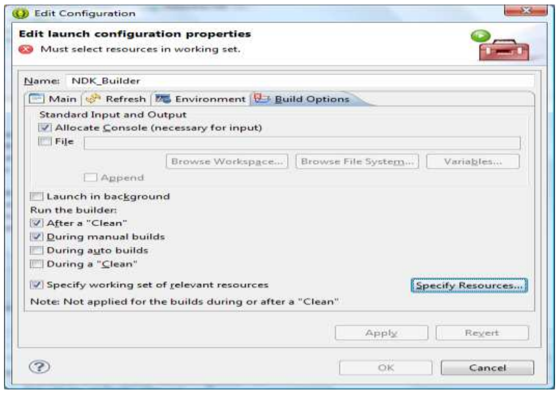
Enter a working set name and select the working set resources.

Working set name: working set

Working set contents:
- NativePthread
  - [ ] .externalToolBuilders
  - [ ] .settings
  - [ ] bin
  - [ ] Default
  - [ ] gen [Generated Java Files]
  - [ ] jni
  - [✓] libs
  - [ ] obj
  - [ ] Pthreads
  - [ ] res
  - [ ] src

Fig. 5. Choose libs folder

Now we have to go to Build Options tab then make sure the Specify working set of relevant resources checkbox is checked or not.



Edit Configuration

Edit launch configuration properties

Must select resources in working set.

Name: NDK_Builder

Main | Refresh | Environment | Build Options

Standard Input and Output
- [✓] Allocate Console (necessary for input)
- [ ] File
- [ ] Append

- [ ] Launch in background

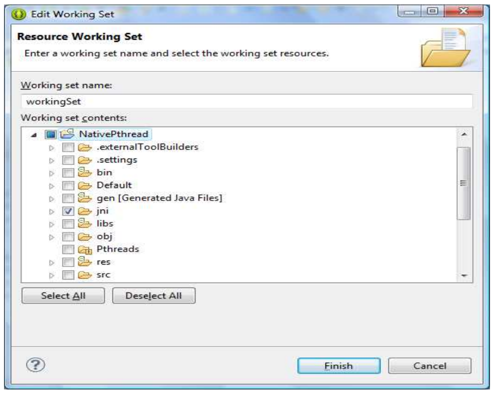
Run the builder:
- [✓] After a "Clean"
- [✓] During manual builds
- [ ] During auto builds
- [ ] During a "Clean"

- [✓] Specify working set of relevant resources

Note: Not applied for the builds during or after a "Clean"

Fig. 6. Specify JNI Resources

Since the NDK build only wants to occur when editing files in the jni folder, check "jni" folder and press the Finish button.



Edit Working Set

Resource Working Set

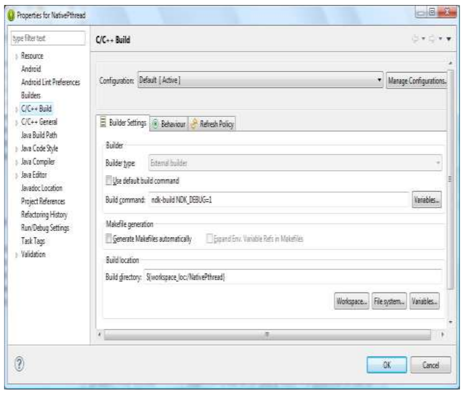
Enter a working set name and select the working set resources.

Working set name: workingSet

Working set contents:
- NativePthread
  - [ ] .externalToolBuilders
  - [ ] .settings
  - [ ] bin
  - [ ] Default
  - [ ] gen [Generated Java Files]
  - [✓] jni
  - [ ] libs
  - [ ] obj
  - [ ] Pthreads
  - [ ] res
  - [ ] src

Fig. 7. Choose JNI folder

Before running android app, we should establish C/C++ Build Configurations. In Build Settings tap, uncheck use default build command then type ndk-build NDK_DEBUG=1 in Build command textbox area. Likewise, uncheck Generate Make files automatically and then select present workspace project shown bellow.



Properties for NativePthread

C/C++ Build

Configuration: Default [Active]

Builder Settings | Behaviour | Refresh Policy

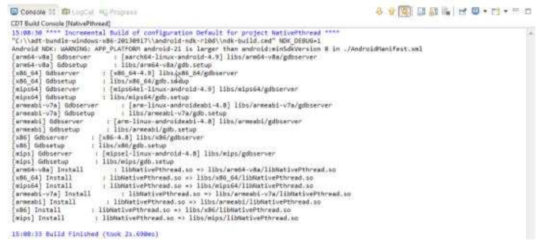
Builder
- Builder type: External builder
- [ ] Use default build command
- Build command: ndk-build NDK_DEBUG=1

Makefile generation
- [ ] Generate Makefiles automatically
- [ ] Expand Env. Variable Refs in Makefiles

Build location
- Build directory: ${workspace_loc:/NativePthread}

Fig. 8. C/C++ Build

Finally, click on OK button in the properties dialog window. Now, Android NDK builder settings is ready to run.



```
CDT Build Console [NativePthread]
15:08:30 **** Incremental Build of configuration Default for project NativePthread ****
"C:\\adt-bundle-windows-x86-20130917\\android-ndk-r10d\\ndk-build.cmd" NDK_DEBUG=1
Android NDK: WARNING: APP_PLATFORM android-21 is larger than android:minSdkVersion 8 in ./AndroidManifest.xml
[arm64-v8a] Gdbserver      : [aarch64-linux-android-4.9] libs/arm64-v8a/gdbserver
[arm64-v8a] Gdbsetup       : libs/arm64-v8a/gdb.setup
[x86_64] Gdbserver      : [x86_64-4.9] libs/x86_64/gdbserver
[x86_64] Gdbsetup       : libs/x86_64/gdb.setup
[mips64] Gdbserver      : [mips64el-linux-android-4.9] libs/mips64/gdbserver
[mips64] Gdbsetup       : libs/mips64/gdb.setup
[armeabi-v7a] Gdbserver      : [arm-linux-androideabi-4.8] libs/armeabi-v7a/gdbserver
[armeabi-v7a] Gdbsetup       : libs/armeabi-v7a/gdb.setup
[armeabi] Gdbserver      : [arm-linux-androideabi-4.8] libs/armeabi/gdbserver
[armeabi] Gdbsetup       : libs/armeabi/gdb.setup
[x86] Gdbserver      : [x86-4.8] libs/x86/gdbserver
[x86] Gdbsetup       : libs/x86/gdb.setup
[mips] Gdbserver      : [mipsel-linux-android-4.8] libs/mips/gdbserver
[mips] Gdbsetup       : libs/mips/gdb.setup
[arm64-v8a] Install        : libNativePthread.so => libs/arm64-v8a/libNativePthread.so
[x86_64] Install        : libNativePthread.so => libs/x86_64/libNativePthread.so
[mips64] Install        : libNativePthread.so => libs/mips64/libNativePthread.so
[armeabi-v7a] Install        : libNativePthread.so => libs/armeabi-v7a/libNativePthread.so
[armeabi] Install        : libNativePthread.so => libs/armeabi/libNativePthread.so
[x86] Install        : libNativePthread.so => libs/x86/libNativePthread.so
[mips] Install        : libNativePthread.so => libs/mips/libNativePthread.so

15:08:33 Build Finished (took 2s.698ms)
```

Fig. 9. NDK Build

4.6 Run the Android App

Run our Android apps, via "Run As" → "Android Application" [13]-[15]. We can see all the communications from the native programs on the displays. Check the "LogCat" panel to verify that the shared library "libNativePthread.so" is loaded or not. At last, build and run our application as usual way. The Android SDK build tools will put together into shared libraries in "NativePthread.apk" file.

5. CONCLUSION

The JNI is a part of the Java standard that enables developers to write methods in languages that are compiled to native code, such as C and C++ and call those methods from Java code. JNI is also what connects the Java runtime environment to the underlying OS. JNI is especially helpful when we desire to employ platform-specific features or obtain benefit of hardware in the platform that cannot be approached through Android APIs. The Android NDK makes it further suitable to compile native code that can be used with Android programs. The Android NDK presents system headers for an extremely incomplete set of native APIs and libraries maintained by the Android platform. The primary motivation for considering the use of Pthreads on mobile architecture can achieve optimum performances on multi-core mobile architectures. Pthreads should capable to be structured into separate, independent tasks that can carry out concurrently. Since, the Android NDK affords system headers for an extremely incomplete set of native APIs and libraries maintained by the Android platform at present. Hence, we are expecting full-fledged version of native APIs for Pthread access into Android apps. We implemented Android NDK to create and work with Pthread, which can exploit Pthreads to execute in hybrid mode with Java threads. Also, we exercised in setting affinity for a Pthread. At present with the limitations of android-21 platform, while a typical Android system incorporates many native shared libraries, we should believe them an implementation detail that might alter drastically between platform releases.

REFERENCES

[1] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, Java Native Intel Thread Building Blocks for Win32 Platform, Asian Journal of Information Technology, Accepted on for publication on March 03, 2014. Medwell Publishing, ISSN: 1682-3915.

[2] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, Java Native Pthreads for Win32 Platform, “World Congress on Computing and Communication Technologies (WCCCT’14), vol., no., pp.195-199, Feb. 27 2014-March 1 2014, Tiruchirappalli, published in IEEE Xplore, ISBN: 978-1-4799-2876-7, DOI: 10.1109/WCCCT.2014.13

[3] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, Overall Aspects of Java Native Threads on Win32 Platform, Second International Conference on Emerging Research in Computing, Information, Communication and Applications (ERCICA-2014), Vol. II, pp.667-675, August 01-02, 2014, Bangalore, published in ELSEVIER in India. ISBN: 9789351072621.

[4] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira,JNT-Java Native Thread for Win32 Platform, International Journal of Computer Applications, USA, Volume 71, Issue 1, May 2013, ISSN 0975-8887. DOI: 10.5120/12212-8249

[5] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira,Parallel: One Time Pad using Java, International Journal of Scientific & Engineering Research, USA, Volume 3, Issue 11, PP.1109-1117, November 2012, ISSN 2229-5518 11

[6] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira,Setting CPU Affinity in Windows Based SMP Systems Using Java, International Journal of Scientific & Engineering Research, USA, Volume 3, Issue 4, PP. 893-900, April 2012, ISSN 2229-5518 11

[7] Bala Dhandayuthapani Veerasamy, An Introduction to Parallel and Distributed Computing through java, First Edition, Penram International Publishing (India) Pvt, Mumbai, India, ISBN-10: 81-87972-84-X, ISBN-13: 978-81-87972-84-6.

[8] Bala Dhandayuthapani Veerasamy, Dr. G. M. Nasira, PhD, Performance Analysis of Java NativeThread and NativePthread on Win32 Platform (unpublished work)

[9] Yeong-Jun Kim, Seong-Jin Cho, Kil-Jae Kim, Eun-Hye Hwang, Seung-Hyun Yoon, Jae-Wook Jeon, Benchmarking Java application using JNI and native C application on Android, 12th International Conference on Control, Automation and Systems (ICCAS), 17-21 Oct. 2012, 284 - 288, ISBN: 978-1-4673-2247-8.

[10] Cheng-Min Lin, Jyh-Horng Lin, Chyi-Ren Dow, Chang-Ming Wen, Benchmark Dalvik and Native Code for Android System, Second International Conference on Innovations in Bio-inspired Computing and Applications (IBICA),16-18 Dec. 2011,320 - 323,ISBN: 978-1-4577-1219-7

[11] Jae Kyu Lee, Prof. Jong Yeol Lee, Android Programming Techniques for Improving Performance, 3rd International Conference on Awareness Science and Technology (iCAST), 27-30 Sept. 2011,386 - 389,ISBN: 978-1-4577-0887-9.

[12] Lars Vogel (accessed since 2014), Android background processing with Handlers, AsyncTask and Loaders - Tutorial, http://www.vogella.com/tutorials/AndroidBackgroundProcessing/article.html

[13] Onur Cinar Pro Android C++ with the NDK, Apress, December, 2012, ISBN:978-1-4302-4827-9

[14] Onur Cinar, Android Apps with Eclipse, Apress, Jun, 2012, ISBN13: 978-1-4302-4434-9

[15] Zigurd Mednieks, Laird Dornin, G. Blake Meike and Masumi Nakamura, Programming Android, Second Edition, Zigurd Mednieks, O’Reilly Media Inc, 2012