Java Native Pthread for Win32 Platform

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

Dr. G.M. Nasira, Ph.D
Assistant Professor in Computer Science
Chikkanna Govt Arts College
Tirupur, India
nasiragm99@yahoo.com

Abstract— **Numerous advanced operating systems honestly hold both time-sliced and multiprocessor threading with a process scheduler. The kernel of an operating system permits programmers to control threads via the system call interface. Java incorporates threading facility within the language itself rather than managing threads as a facility of the underlying operating system. This research finding focuses on how Java can facilitate Pthreads through JNI, which can exploit in Java threads and Native Pthreads to execute in hybrid model.**

Keywords-JNI; Kernel; NativePthread; Pthread; Threads;

I. INTRODUCTION

There are two categories of threading library [1] available, implicit and explicit. Implicit threading libraries pay attention to create, manage and synchronize threads. The flexibility of implicit threading libraries can be written concurrently can take benefits of the limited scope of characteristic within the implicit libraries. There are two significant explicit threading is available. However, the majority of algorithms libraries in implicit threading are OpenMP and Intel TBB. Explicit threading libraries expects the programmer to control all features of threads, including creating threads, associating threads to functions and synchronizing and controlling the interactions between threads and shared resources. The two most important threading libraries in use today are POSIX threads (Pthreads) and Windows Threads by Microsoft.

Java thread scheduled in a perceptive of the particular virtual machine involved. There are two basic variations here: The green threads [2] are scheduled by the virtual machine itself. This is the original model for Java Virtual Machine mostly follows the priority based scheduling. This kind of thread never uses operating system threads library. The native threads [2] are scheduled by the operating system that is hosting the virtual machine. The operating system threads are logically separated into two pieces: user level threads and system level threads. The operating system itself that is the kernel of the operating system lies at system level threads. The kernel is accountable for managing system calls on behalf of programs that run at user level threads.

Pthreads [3] is a consistent model for separating a program into subtasks, whose execution can be interleaved or run in parallel. The "P" in Pthreads arrives from POSIX (Portable Operating System Interface), the family of IEEE operating system interface standards in which Pthreads is defined (POSIX Section 1003.1c to be exact).

Implementations of an API are accessible on various UNIX like POSIX compatible operating systems such as FreeBSD, NetBSD, OpenBSD, GNU/Linux, Mac OS X and Solaris. DR-DOS and Microsoft Windows implementations also exist, which provides a native implementation of POSIX APIs [4] [5] such as Pthreads-win32, which implements Pthreads on top of existing Windows API.

Windows does not support the Pthreads standard natively; therefore the Pthreads-win32 project [4] [5] seeks to provide a portable and open-source wrapper implementation. It can also be used to port UNIX software with modification to the Windows platform. The mingw-w64 “Minimalist GNU for Windows” project also contains a wrapper implementation of Pthreads, winpthreads, which attempts to use more native system calls than the Pthreads-win32 project. The Pthreads-win32 [5] is usually implemented as a dynamic link library (DLL). This has a few notable benefits from the Win32 point of view, but it is more closely models existing Pthreads libraries on UNIX, which are usually shared objects (e.g. libpthread.so). Some of the Pthreads methods [6] [7] are listed in bellow Table I.

TABLE I. PTHREAD METHODS

Method	Usage
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(*start_routine) (void),(void *arg)	Creates a new thread, initializes its attributes, and makes it runnable. The pthread_create subroutine creates a new thread and initializes its attributes using the thread attributes object specified by the attr parameter.
pthread_t pthread_self()	It returns the calling thread's identifier. The pthread_self subroutine returns the calling thread's identifier
pthread_num_processors_np()	It get the number of processors (CPUs) in use by the process
int sleep (unsigned int seconds)	It suspends the execution of the current thread until the timeout interval elapses.
exit(EXIT_SUCCESS)	It signifies the application was successful.
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset)	It sets the CPU affinity mask of the thread to the CPU set pointed to by cpuset.

II. PROBLEM STATEMENT

On a uniprocessing system [3], the concurrent execution of independent tasks will be faster than serial execution, if at least one of these tasks issues a lot of I/O requests and must wait for the device to complete each request. On a multiprocessor [3], even CPU-bound tasks can benefit from concurrency because they can proceed in parallel. Parallel computers can be separated into two kinds of major categories [8] of control flow and data flow. Control-flow parallel computers are fundamentally based on the similar principles as the sequential or von Neumann computer, except that multiple instructions can be executed at any given time. Data-flow parallel computers occasionally referred to as "non-von Neumann," is completely dissimilar in that they have no pointer to active instruction(s) or a locus of control. The control is totally distributed with the availability of operands triggering the activation of instructions. In what follows, we will concentrate exclusively on control-flow parallel computers. Flynn's MPMD is in fact a "high level" programming model. MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or dissimilar program as other tasks. All tasks may use dissimilar data.

The Pthreads standard [3] states concurrency; it permits parallelism to be at the option of system implementers. A programmer describes those tasks or threads that can occur concurrently. Whether the threads actually run in parallel is a function of the operating system and hardware on which they run. Because, Pthreads was intended in this way, a Pthreads program can run without alteration on uniprocessor as well as multiprocessor systems.

In the control flow of Java Thread API, only one public void run() method is to execute our tasks. Of course, we can partition our task within public void run() method using thread names. In order to implement with dissimilar executable method, this research focuses to encompass java thread with NativePthread. NativePthread is a native Pthread for java will be used in win32 platform. Hence, we can take advantages of Pthreads into Java program and can also hybrid with Java threads. The main advantages in this research paper paying attention in using Pthreads are, allows enclosing two different thread execution methods and setting affinity for the threads. This research finding focuses on how Java can facilitate Pthreads through JNI, which can exploit in Java threads and Native Pthreads to execute in hybrid mode.

III. METHODOLOGIES

JNI permits us to utilize native code, when an application cannot be written entirely in the Java language. It want to implements time-critical code in a lower-level, faster programming language. It has legacy code or code libraries that we want to get into from Java programs. It desires platform dependent features not supported in the standard Java class library. In order to create and work with native threads by means of Java Native Interface [9] [10] application that calls a C++ function with the following steps:

A. Declare the native method in java class

JNI begin by writing the following program in the Java programming language. The Program 1 defined a class named NativePthread that contains native Pthreads methods. The native keyword notifies the Java compiler that a method is applied in native code outside of the Java class in which it is being declared. Native methods can only be declared in Java classes, not implemented, so native methods do not have a body. The native methods have implemented using Dev-C++ program in the next following section.

Program 1- Native Pthreads method declarations

```
//Java Native Pthread
package JNT.Win32.Kernel;
class NativePthread {
       public native void createPthread();
       public native int getCurrentThreadId();
       public native int getNumberOfProcessor();
       public native void Pthreadsleep(final long wait);
       public native void setPthreadAffinityMask(final int mask);
       public void executeNativePthread()
       {       /* parallel code*/
       }
  }
```

The NativePthread is defined in package named JNT.Win32.Kernel. The class NativePthread declared with native Pthread methods, which will be implemented in C++ using JNI. Since createPthread() declared as final native, can be called without creating an object for NativePthread class. The sleep (final long wait) is used to give waiting time. The int getNumberOfProcessor() will return number of processor available in the computer system. The int getCurrentThreadId() will return current threaded. The setPthreadAffinityMask(final int mask) method is used to affinity mask for a current thread and the executeNativePthread() will assist to write parallel tasks.

B. Compiling java class and creating native method header file

We have compiled the Java code down to byte code. One way to do this is to use the Java compiler javac, which comes with the SDK. The command we used to compile our Java code to byte code is:

```
javac NativePthread.java
```

This command generated a NativePthread.class file in the JNT.Win32.Kernel directory. The next step, we created C/C++ header file that defines native function signatures. One way to do this is we used the native method C stub generator tool javah.exe, which comes with the SDK. This tool is designed to create a header file that defines C-style functions for each native method it found in a Java source code file. The command we used on JNT.Win32.Kernel directory is:

```
javah NativePthread
```

The name of the header file is the class name with ".h" appended to the end of it. The command shown above generates a file named JNT_Win32_Kernel_NativePthread.h, which is show bellow in Program 2.

Program 2 - JNT_Win32_Kernel_Nativepthread.H

```
#include <jni.h>
#ifndef _Included_JNT_Win32_Kernel_NativePthread
#define _Included_JNT_Win32_Kernel_NativePthread
#ifdef __cplusplus
extern "C" {
#endif
/ * Class:    JNT_Win32_Kernel_NativePthread
 * Method:   createPthread                    Signature: ()V  */
JNIEXPORT void JNICALL
Java_JNT_Win32_Kernel_NativePthread_createPthread (JNIEnv *,
jobject);
/ * Class:    JNT_Win32_Kernel_NativePthread
   Method:   getCurrentThreadId               Signature: ()I  */
JNIEXPORT jint JNICALL
Java_JNT_Win32_Kernel_NativePthread_getCurrentThreadId
  (JNIEnv *, jobject);
/ * Class:    JNT_Win32_Kernel_NativePthread
   Method:   getNumberOfProcessor             Signature: ()I  */
JNIEXPORT jint JNICALL
Java_JNT_Win32_Kernel_NativePthread_getNumberOfProcessor
  (JNIEnv *, jobject);
/ * Class:    JNT_Win32_Kernel_NativePthread
   Method:   Pthreadsleep                     Signature: (J)V  */
JNIEXPORT void JNICALL
Java_JNT_Win32_Kernel_NativePthread_Pthreadsleep
  (JNIEnv *, jobject, jlong);
/*  Class:    JNT_Win32_Kernel_NativePthread
   Method:   setPthreadAffinityMask           Signature: (I)V  */
JNIEXPORT void JNICALL
Java_JNT_Win32_Kernel_NativePthread_setPthreadAffinityMask
  (JNIEnv *, jobject, jint);
#ifdef __cplusplus
}
#endif
#endif
```

The C/C++ function signatures in JNT_Win32_Kernel_NativePthread.h are quite different from the Java native method declarations in NativePthread.java. JNIEXPORT and JNICALL is compiler-dependent specifier for export functions. The return types are C/C++ types that map to Java types. The parameter lists of all these functions have a pointer to a JNIEnv and a jobject, in addition to normal parameters in the Java declaration. The pointer to JNIEnv is actually a pointer to a table of function pointers. These functions provide the various faculties to manipulate Java data in C and C++. The jobject parameter refers to the current object. Thus, if the C or C++ code needs to refer back to the Java side, this jobject acts as a reference or pointer, back to the calling Java object. The function name itself is made by the "Java_" prefix, followed by the fully qualified package name followed by an underscore and class name followed by an underscore and the method name.

C. Implementing native methods

The JNI-style header file generated by javah helped us to write C++ implementations for the native method. When it comes to writing the C++ function implementation, the important thing to keep in mind is that our signatures must be exactly like the function declarations from JNT_Win32_Kernel_NativePthread.h.

The function that we write must follow the prototype specified in the generated header file. We implemented the method in C++ file named Dllmain.cpp as shown in the following Program 3.

Program 3 - Dllmain.cpp

```
#include <jni.h>
#include "JNT_Win32_Kernel_NativePthread.h"
#include <pthread.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;
}
void *NativePthread(void *argv){
   javaVM->AttachCurrentThread((void **)&env, NULL);
   jclass cls = env-> GetObjectClass(aThreadObj);
   jmethodID method = env->GetMethodID(aThreadCls,
"executeNativePthread", "()V");
   env->CallVoidMethod(aThreadObj, method);
   exit(EXIT_SUCCESS);
   javaVM->DetachCurrentThread();
   return NULL;
 }
JNIEXPORT void JNICALL
Java_JNT_Win32_Kernel_NativePthread_createPthread
  (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);
if(pt != 0) {
 printf("Error: JNT.Win32.Kernel.NativePthread.createPthread() method
failed\n");
 }
}
JNIEXPORT jint JNICALL
Java_JNT_Win32_Kernel_NativePthread_getCurrentThreadId
  (JNIEnv *, jobject){
 return (jint)gettid();
}
JNIEXPORT void JNICALL
Java_JNT_Win32_Kernel_NativePthread_Pthreadsleep
  (JNIEnv *env, jobject obj, jlong wait){
 Sleep(wait);
}
JNIEXPORT jint JNICALL
Java_JNT_Win32_Kernel_NativePthread_getNumberOfProcessor
  (JNIEnv *env, jobject obj){
        int num_cores =  pthread_num_processors_np();
        return (jint)num_cores;
}
JNIEXPORT void JNICALL
Java_JNT_Win32_Kernel_NativePthread_setPthreadAffinityMask
  (JNIEnv *env, jobject obj, jint mask){
  int num_cores =  pthread_num_processors_np();
   if (mask >= num_cores)
     printf("The core processor is not exists with :%d",mask);
      cpu_set_t cpuset;
  CPU_ZERO(&cpuset);   CPU_SET(mask, &cpuset);
  pthread_t current_thread = pthread_self();
  int s=pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset);
}
```

The Dllmain.cpp program 3 included with pthread.h, thus libpthread.a can be used to create NativePthread. A Java_JNT_Win32_Kernel_NativePthread_create Pthread(JNIEnv *env, jobject obj) 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. The Java_JNT_Win32_Kernel_NativePthread_ getNumberOfProcessor (JNIEnv *env, jobject obj) get number of processor in the current system using pthread_num_processors_np(). The Java_JNT_Win32_Kernel_NativePthread_getCurrentThread Id (JNIEnv *env, jobject obj) calls getid() method to get current threadid using pthread_self().

The JNIEXPORT void JNICALL Java_JNT_Win32_Kernel_NativePthread_setPthreadAffinity Mask (JNIEnv *env, jobject obj, jint mask) calls pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset) to set the affinity for the current thread.

Once NativePthread is created, it will start running public void executeNativePthread() method immediately. This research finding has openings to have methods of scheduling, synchronization and resource sharing such as mutex locks, critical sections.

D. Compiling C++ and creating native library

Dev-C++ allows creating Dynamic Link Library (DLL), is a shared library that contains the native code. In order to create dll, we should choose DLL project, which should be linked libpthread.a(pthreads-win32) in project option then parameter tab on linker section. The dllmain.cpp and JNT_Win32_Kernel_NativePthread.h program should be added in DLL project. The library we have created is NativePthread.dll, which can be accessed in java program through System.load("NativePthread.dll").

IV. TESTING NATIVE PTHREAD PROGRAM

At this point, we have the three components ready to run the program (see Fig.1). The class file (NativePthread.class) calls a native method, native library (NativePthread.dll) and pthread.dll (pthreads-win32). The following program 4 is the testing program, TestMain class allowed us to run the program on Win32 platform, because we have used Pthreads library through JNI. First of all the program should be imported with JNT.Win32.Kernel.NativePthread; this package library included a line of code that loaded a native library into the program through System.load ("NativePthread.dll"). The pthread.dll (pthreads-win32) should be placed in windows/system32 folder. When program started execution the public static void main(String[] args) is called JVM to creates the main thread. Inside the TestMain class constructor, we called createPthread() to create Pthreads, will automatically call the public void executeNativePthread() to execute our parallel task. Of course, we can create any number of supported threads based on the computer system memory stack availability.

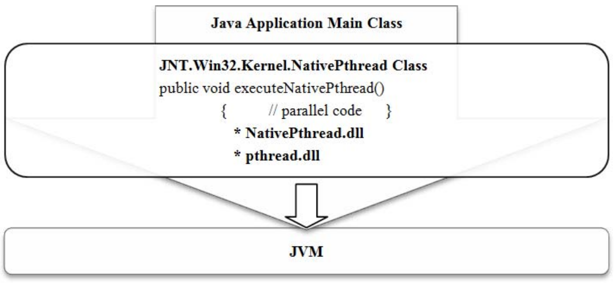
Java Application Main Class
JNT.Win32.Kernel.NativePthread Class
public void executeNativePthread()
{ // parallel code }
* NativePthread.dll
* pthread.dll
JVM

Figure 1. Native Pthread model using JNI

Program 4 – Testing Native Threads

```
package JNT.Win32.Kernel;
import JNT.Win32.Kernel.*;
public class TestMain extends NativePthread {
static {   System.load("NativePthread.dll");   } //loading library
public TestMain() {
  System.out.println("Available core processor
is:"+getNumberOfProcessor());
  createPthread();  //creating first thread
  createPthread();  //creating second thread
  }
@Override
public void executeNativePthread(){ // runs parallel task
try{
    for(int i=0;i<5;i++){
     System.out.println("\t\t"+ getCurrentThreadId() + "\t" + i);
     Pthreadsleep(1);  // thread waiting   }
  }catch(Exception e){ System.out.println("Error in Native Pthread"+e); }
}
public static void main(String[] args) {
 try{ TestMain t=new TestMain();
      }catch(Exception e){ System.out.println("Error in Main Thread"+e); }
   }
}
```

In Program 4, public void executeNativePthread() method from NativePthread class executes two different threads in parallel. The sample output of the program as follows.

```
Available core processor is:2
Native PthreadID 12025800       Value 0
Native PthreadID 12025976       Value 0
Native PthreadID 12026544       Value 0
Native PthreadID 12025976       Value 1
Native PthreadID 12025800       Value 1
Native PthreadID 12026544       Value 1
Native PthreadID 12025800       Value 2
Native PthreadID 12025976       Value 2
Native PthreadID 12026544       Value 2
Native PthreadID 12025976       Value 3
Native PthreadID 12025800       Value 3
Native PthreadID 12025800       Value 4
Native PthreadID 12026544       Value 3
Native PthreadID 12025976       Value 4
```

V. HYBRID THREAD MODEL

The Java thread and NativePthread are merged in this model. Java thread (green thread) can be created through java.lang.Thread class. The NativePthread (NativePthread.dll library) can be attached with Java Virtual Machine (JVM) (see Fig 2.). Therefore, the java program has NativePthread as well as green thread. This articulate more than one thread execution method can be explicitly used in Java, which means public void run() and public void executeNativePthread() method implemented in parallel. Hence, Flynn's Multiple Program Single Data (MPSD) and Multiple Program Multiple Data (MPMD) programming models can be utilized well though Java threads program. The following Program 5 executed with hybrid model thread by creating thread using Thread class and NativePthread class.

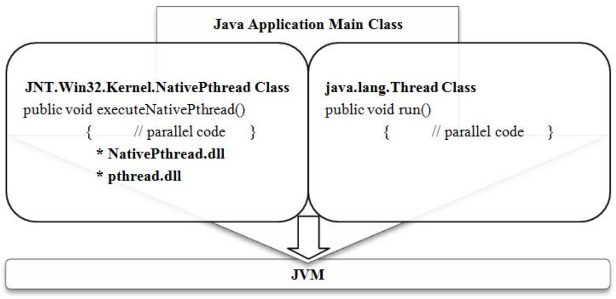

Figure 2. Hybrid Thread using JNT in Java

Program 5 - Hybrid thread model

```
package JNT.Win32.Kernel;
import JNT.Win32.Kernel.*;
class HybridTest extends NativePthread implements Runnable{
Thread t;
static {  System.load("NativePthread.dll");  }
HybridTest(){  t=new Thread(this);  t.start();  createPthread();  }
@Override
public void executeNativePthread(){ // call from NativePthread class
try{   for(int i=0;i<3;i++)   {
          System.out.println("Native PthreadID\t" +
          getCurrentThreadId() + "\tValue " + i);
          Pthreadsleep(1);  }
  }catch(Exception e){ System.out.println("Error in NativePthread"+e); }
}
@Override
public void run(){ // call from Thread class
try{ for(int i=0;i<3;i++)  {
       System.out.println("Green ThreadID\t\t" + t.getId() + "\t\tValue " + i);
          t.sleep(1);            }
  } catch(Exception e){  System.out.println("Error in Green Thread"+e);  }
}
public static void main(String[] args) {
try{      HybridTest ht = new HybridTest();
   }catch(Exception e){  System.out.println("Error in Main Thread"+e);  }
  }
}
```

In Program 5, the public void executeNativePthread() method from NativePthread class executes native Pthread and the public void run() from Thread class executes green thread. In a same program both native Pthreads and Java green threads are executed in hybrid mode. The sample output of the program as follows.

```
Green ThreadID       8              Value 0
Native PthreadID     10568616       Value 0
Green ThreadID       8              Value 1
Native PthreadID     10568616       Value 1
Green ThreadID       8              Value 2
Native PthreadID     10568616       Value 2
```

In Win32 there are two references [7] to thread objects, a thread handle which is local to the process and most similar to a POSIX TID and a thread ID, which is a system-wide referent to the thread. The getCurrentThreadId()method from NativePthread class returns "10568616" (POSIX TID) as its present thread and the values are printed though its loop. Likewise, the t.getId() method from Thread class returns "8" (Java thread ID) as its present thread and the values are printed through its loop. In this program, we have created one native Pthread and one green thread; Of course, we can create any number of supported threads based on the computer system memory stack availability.

VI. CONCLUSION

The native threads are scheduled by the operating system that is hosting the virtual machine. The user level Pthreads has used the kernel of the operating system lies at system level threads. The kernel is accountable for managing system calls on behalf of programs that run at user level Pthreads. This research finding focused on how Java can facilitate Pthreads through JNI, which enables Java threads and native threads to schedule and execute in hybrid mode. As a result, this research brings up opening to exploit Pthreads within Java program, it is strongly recommending for Flynn's Multiple Program Multiple Data (MPMD) and Multiple Program and Single Data (MPSD) through method level concurrency.

REFERENCES

[1] Clay Breshears, "The Art of Concurrency", O'Reilly Media,2009.

[2] Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, "JNT - Java Native Thread for Win32 Platform", International Journal of Computer Applications, Foundation of Computer Science, USA, Vol 70 num 24,pp1-9, May 2013.

[3] Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell, "Pthreads Programming", O'Reilly & Associates, Inc.,1996.

[4] "POSIX Threads", http://en.wikipedia.org/wiki/POSIX_Threads

[5] Pthread-w32, https://sourceware.org/pthreads-win32

[6] Davud R.Butenhof, "Programming with POSIX Thread", Addison Wesley Longman Inc, 1997.

[7] Bil Lewis, Daniel J. Berg, "PThreads Primer", Sun Microsystems Inc, 1996

[8] Bala Dhandayuthapani Veerasamy, "Concurrent Approach to Flynn's MPMD Classification through Java", International Journal of Computer Science and Network Security, Korea, Vol. 10 No. 2 pp. 164-167 , February 2010.

[9] "Java Native Interface", http://java.sun.com/javase/6/docs/technotes /guides/jni/index.html

[10] Sheng Liang, The Java™ Native Interface Programmer's Guide and Specification, Sun Microsystems Inc, 1999.