Setting CPU Affinity in Windows Based SMP Systems Using Java

Bala Dhandayuthapani Veerasamy, Dr. G.M. Nasira

Abstract—Multithreaded program involves multiple threads of control within a single program on single or multiple environments. In multithreaded programming model, a single process can have multiple, concurrent execution paths on CPUs. *Thread affinity* benefit a thread to run on a specific subset of processors. Multithreaded programming is written in many programming languages with an improvement of setting an affinity to threads. Java supports to develop multithreaded programming, while it does not contain any method to set an affinity for threads on CPU. This paper articulates the method to setting CPU affinity for threads in windows based SMP systems using Java.

Index Terms— Affinity mask, JNA, Multithread, Thread, Windows programming

1 INTRODUCTION

MOST concurrent applications [3] are organized around the execution of tasks: abstract, discrete units of work. Dividing the work of an application into tasks simplifies program organization, facilitates error recovery by providing natural transaction boundaries, and promotes concurrency by providing a natural structure for parallelizing work. The first step in organizing a program around task execution is identifying sensible task boundaries. Ideally, tasks are independent activities: work that doesn't depend on the state, result, or side effects of other tasks. Independence facilitates concurrency, as independent tasks can be executed in parallel if there are adequate processing resources. For greater flexibility in scheduling and load balancing tasks, each task should also represent a small fraction of your application's processing capacity.

A multithreaded program [5] contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Each thread in a multithreaded process [9] can be dispatched to a different processor in a multiprocessor system. This collaboration across multiple processors improves single-application performance. Multithreading enables [5] you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. Threads exist in several states. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed. Thread priority determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch. Problems can arise from the differences in the way that operating systems context-switch thread of equal priority. Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor.

Windows supports concurrency among processes because threads in different processes may execute concurrently. Moreover, multiple threads within the same process may be allocated to separate processors and execute simultaneously. A multithreaded process achieves concurrency without the overhead of using multiple processes. Threads within the same process can exchange information through their common address space and have access to the shared resources of the process. Threads in different processes can exchange information through shared memory that has been set up between the two processes. Multiprocessor programming is challenging because modern computer systems are inherently asynchronous: activities can be halted or delayed without warning by interrupts, preemption, cache misses, failures, and other events. These delays are inherently unpredictable, and can vary enormously in scale: a cache miss might delay a processor for fewer than ten instructions, a page fault for a few million instructions, and operating system preemption for hundreds of millions of instructions.

The general-purpose process and thread facility must support the particular process and thread structures of the

- *Bala Dhandayuthapani Veerasamy is currently a Research Scholar in Information Technology, Manonmaniam Sundaranar University, India. E-mail: dhanssoft@gmail.com*
- *Dr. G.M. Nasira is currently working as an Assistant Professor-Computer Applications, Department of Computer Science, Govt Arts College (Autonomous), Salem -636007, Tamilnadu, India. E-mail: nasiragm99@yahoo.com*

various OS clients. It is the responsibility of each OS subsystem to exploit the Windows process and thread features to emulate the process and thread facilities of its corresponding OS. An application client process issues its process creation request to the OS subsystem; then a process in the subsystem in turn issues a process request to the Windows executive. Windows enables the subsystem to specify the parent of the new process. The new process then inherits the parent's access token, quota limits, base priority, and default processor affinity.

1.1 Win32 API and Dynamic-Link Libraries

Windows programming [12] is useful to start off with some appreciation of some new terms intrinsic to Windows: objects, handles, instances, messages, and callback functions. These give us the mechanics of programming in this environment, that is, they are tools that we need to use. A Dynamic Link Library (DLL) is a module that contains functions and data that can be used by another module (application or DLL). DLLs provide a way to modularize applications so that their functionality can be updated and reused more easily. DLLs also help reduce memory overhead when several applications use the same functionality at the same time, because although each application receives its own copy of the DLL data, the applications share the DLL code. The Windows application programming interface (API) is implemented as a set of DLLs, so any process that uses the Windows API uses dynamic linking. Dynamic linking allows a module to include only the information needed to locate an exported DLL function at load time or run time.

Every process that loads the DLL maps it into its virtual address space. After the process loads the DLL into its virtual address, it can call the exported DLL functions. The system maintains a per-process reference count for each DLL. When a thread loads the DLL, the reference count is incremented by one. When the process terminates, or when the reference count becomes zero, the DLL is unloaded from the virtual address space of the process. Like any other function, an exported DLL function runs in the context of the thread that calls it. Therefore, the following conditions apply: 1) The threads of the process that called the DLL can use handles opened by a DLL function. Similarly, handles opened by any thread of the calling process can be used in the DLL function. 2) The DLL uses the stack of the calling thread and the virtual address space of the calling process. 3) The DLL allocates memory from the virtual address space of the calling process. Programs can be written in different programming languages can call the same DLL function as long as the programs follow the same calling convention that the function uses. The calling convention (such as C, C++, .Net and etc) controls the order in which the calling function must push the arguments onto the stack, whether the function or the calling function is responsible for cleaning up the stack, and whether any arguments are passed in registers. The Kernel32.dll is an important DLL for doing multithreaded programs; here the following important function we used in our research.

- Public Declare Function SetThreadAffinityMask Lib "kernel32" Alias "SetThreadAffinityMask" (ByVal hThread As Long, ByVal dwThreadAffinityMask As Long) As Long

1.2 Symmetric Multiprocessing Support

Symmetric Multiprocessing (SMP) is a multiprocessing architecture in which multiple CPUs, residing in one cabinet, share the same memory. It is the tightly coupled process of program tasks being shared and executed, in true parallel mode, by multiple processors who all work on a program at the same time. Typically, these have large, single units with multiple processors that utilize shared memory, I/O resources, and, of course, a single operating system. The term SMP is so closely associated with shared memory that it is sometimes misinterpreted as standing for "shared memory parallel". SMP systems range from two to as many as 32 or more processors. However, if one CPU fails, the entire SMP system is down. Clusters of two or more SMP systems can used to provide high availability (fault resilience). If one SMP system fails, the others continue to operate.

Windows supports an SMP hardware configuration. The threads of any process, including those of the executive, can run on any processor. In the absence of affinity restrictions, the microkernel assigns a ready thread to the next available processor. This assures that no processor is idle or is executing a lower-priority thread when a higher-priority thread is ready. Multiple threads from the same process can be executing simultaneously on multiple processors. As a default, the microkernel uses the policy of soft affinity in assigning threads to processors: The dispatcher tries to assign a ready thread to the same processor it last ran on. This helps reuse data still in that processor's memory caches from the previous execution of the thread. It is possible for an application to restrict its thread execution to certain processors (hard affinity).

Thread affinity forces a thread to run on a specific subset of processors. Setting thread affinity should generally be avoided, because it can interfere with the scheduler's ability to schedule threads effectively across processors. This can decrease the performance gains produced by parallel processing. An appropriate use of thread affinity is testing each processor. The system represents affinity with a bitmask called a processor affinity mask. The affinity mask is the size of the maximum number of processors in the system, with bits set to identify a subset of processors. Initially, the system determines the subset of processors in the mask. You can obtain the current thread affinity for all threads of the process by calling the GetProcessAffinityMask function. Use the SetProcessAffinityMask function [8] to specify thread affinity for all threads of the process. To set the thread affinity for a single thread, use the SetThreadAffinityMask function. The thread affinity must be a subset of the process affinity. On systems with more than 64 processors, the affinity mask initially represents processors in a single processor group. However, thread affinity can be set to a processor in a different group, which alters the affinity mask for the process.

2 IDENTIFYING PROBLEM

2.1 Multithreading in java

The Java run-time system [4] depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles. Java's multithreading system is built upon the Thread class, its methods, and its companion interface Runnable. Thread encapsulates a thread of execution. The Thread class defines several methods that help on managing threads.

After a computational job is designed and realized as a set of tasks, an optimal assignment of these tasks to the processing elements in a given architecture needs to be determined. This problem is called the scheduling problem [6] and is known to be one of the most challenging problems in parallel and distributed computing. The goal of scheduling is to determine an assignment of tasks to processing elements in order to optimize certain performance indexes. Performance and efficiency are two characteristics used to evaluate a scheduling system [9]. We should evaluate a scheduling system based on the quality of the produced task assignment (schedule) and the efficiency of the scheduling algorithm (scheduler). The produced schedule is judged based on the performance criterion to be optimized, while the scheduling algorithm is evaluated based on its time complexity.

In java, most of the executor [2] implementations in java.util.concurrent use thread pools. Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks. A ThreadPoolExecutor can additionally schedule commands to run after a given delay or to execute periodically. ScheduledThreadPoolExecutor class is preferable to Timer when multiple worker threads are needed, or when the additional flexibility or capabilities of ThreadPoolExecutor are required.

Java support flexible and easy use of threads; yet, java does not contain methods for thread affinity to the processors. Setting an affinity thread to multiprocessor [9] is not new to research, since it was already sustained by other multiprogramming languages for example C in UNIX platform [10] and C# in Windows platform [11]. This paper illustrates how java multithreaded program adapt with an affinity thread on multiprocessors in windows platforms.

3 PROBLEM SOLVING METHODS

3.1 Just Peculiar Algorithm (JPA)

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime() method. This method returns the runtime object associated with the current Java application. Most of the methods of class Runtime are instance methods and must be invoked with respect to the current runtime object. The number of processors available to the Java virtual machine can be obtained through the availableProcessors() method. This value may change during a particular invocation of the virtual machine. Java applications are sensitive to the number of available processors should therefore occasionally poll this property and adjust their resource usage appropriately. Multithreaded programming is written in many programming languages with an improvement of setting an affinity to threads. However, Java does not contain any method to set an affinity for threads on CPU. Hence, we carried this research to synchronize all threads associated with the available processors. The following Program 1 is inherited Thread class to assign the affinity settings. The following is Just Peculiar Algorithm (JPA) algorithm shows how to set thread affinities by our own steps.

1. Get the available number of processor (Processors) in system
2. Select the processor number (ProcessorNum), where we assign thread to execute
3. Check whether selected processor number (ProcessorNum) is greater than and equal to the available processors (Processors) in the system or not. If selected processor is greater than and equal to available processor (Processors), throw IllegalArgumentException.
4. Initialize the incremental for loop with variable=0 and check the variable is less than available number of processor (Processors)
5. Check if the variable is equal to selected processor number (ProcessorNum) then create a thread.

The following Program 1 developed, based on the above algorithm with an affinity thread named as JThreadCore class and it packaged in javax.core, which can be exploit in user's applications. The JThreadCore class constructors and methods discussed bellow.

Constructors:

JThreadCore(String name)
JThreadCore(String name, int ProcessorNum).
The first constructor with the String name argument will assign the name of the thread. And the second constructor with String name argument will assign the name of the thread and int ProcessorNum will select the processor to execute the thread. To assign the processor, ultimately it calls the setAffinity(int ProcessorNum) to select the processor.

Methods:

synchronized void setAffinity(int ProcessorNum)
int getAffinity()
When two or more threads need to access to a shared processor, they need some way to ensure that the processor will be used by only one thread at a time. Consequently, the

setAffinity() method implemented with synchronized method, it can allow one thread can own a monitor at a given time. The argument of setAffinity(), int ProcessorNum will select the processor number to execute the Thread. And the getAffinity() method will retune the affinity of the running thread.

Program 1: JThreadCore.java

```
// package name
package javax.core;
// inherited with Thread class
public class JThreadCore extends Thread{
// get the available system processor
private int Processors ;
// select the processor to execute the thread
private int ProcessorNum;
//assign the name of the thread
private static  String name;
Thread t;
public JThreadCore(String name){
//interchange the thread name value to class variable
this.name=name;
}
public JThreadCore(String name,int ProcessorNum){
//interchange the thread name value to class variable
this.name=name;
//interchange the processor number to class variable
this.ProcessorNum=ProcessorNum;
//call setAffinity method
setAffinity(ProcessorNum);
}
//synchronized method to select the processor
public synchronized void setAffinity(int ProcessorNum){
// interchange processor number to class variable
this.ProcessorNum=ProcessorNum;
//get the available processor in the system
Processors  = Runtime.getRuntime().availableProcessors();
//check selected processor is greater than equal to the selected
//processor
if (ProcessorNum>=Processors )
throw new IllegalArgumentException("This processor is not available");
//create threads using loop
for(int i=0; i < Processors ; i++)
//check i value is equal to user selected processor
 if (i==ProcessorNum){
            //create thread
             t=new Thread(name); }
}
//get the affinity of the thread
public int getAffinity(){
//return the core  number, in which current thread is running
 return ProcessorNum;
}
}
```

The following Program 2 developed to exercise the JThreadCore library. The Test class main main function has been created the object for Test("one",5,1,0) constructor. Once object created the constructor Test(String name,int wait,int pri,int afi) automatically called. This assigns the name of thread as "one", waiting time as 5, priority as 1 and processor 1 to execute. The start() method in the constructor start the thread. Once thread started, it automatically call public void run() method, will execute until it reach the Thread.sleep() method. This Thread.sleep() will give chance, if other thread assigned on the same processor. Likewise, "two", "three" and "four" thread will execute on the assigned processor.

This program measured execution time using System.currentTimeMillis(). The formula to calculate the execution time is elapsedTime = (stopTime - startTime) / wait. Here startTime is starting time of the thread execution, stopTime is end of thread execution and wait is waiting time of each threads (same waiting time for all the threads). If suppose different waiting time assigned to thread, we should hold summation of all thread waiting time.

Program 2: Testing JThreadCore

```
import javax.core.JThreadCore;
class Test extends JThreadCore{
private static String name;
private static int wait;
private static int pri,afi;
Test(String name,int wait,int pri,int afi){
super(name);
this.name=name;        this.wait=wait;
this.pri=pri;          this.afi=afi;
setPriority(pri);
setAffinity(afi);
start();
}
public void run(){
 try{
  for(int i=0;i<5;i++){
   System.out.println(i+"\t\t"        + getId()+"\t\t\tCPU "+
   getAffinity()+ "\t\t" +currentThread()); sleep(wait);
  }
  }catch(InterruptedException e){System.out.println(e);    }
 }
public static void main(String s[]){
long startTime,stopTime,elapsedTime;
try{
 startTime = System.currentTimeMillis();
 System.out.println("Value\tThread Id\tCPU #\t\t\tThread");
 System.out.println("-----------------------------------------------------");
Test a1= new Test("one",5,1,0);
Test a2= new Test("two",5,1,1);
Test a3= new Test("three",5,10,0);
Test a4= new Test("four",5,10,1);
a1.join(); a2.join(); a3.join(); a4.join();
stopTime = System.currentTimeMillis();
elapsedTime = (stopTime - startTime) / wait;
System.out.println("Used exection time in ms: " +elapsedTime);
  }catch(Exception e){System.out.println(e);                    }
 }
}
```

This program is tested with with Inter Core™ 2 Duo CPU

T8100 @ 2.10 GHz (see Figure 1).Threads can run concurrently on assigned CPU absolutely, wherein threads "one" and "three" assigned on CPU 0, threads "two" and "four" assigned on CPU 2. The performance of this program evaluated on different multi-core environment by selecting different affinity on CPU, which is described on the Result and Discussion section. This research exercised with the available Java Thread library only to solve affinity thread problem. Since Java is a platform independent language, this program benefited to execute on any platform. The assigned thread names, not properly fixed on running threads are a big weakness of this research.

General Output

```
--------------------Configuration: <Default>--------------
Value   Thread Id   CPU #           Thread
-----------------------------------------------------------
0       12          CPU 0       Thread[Thread-2,10,main]
0       14          CPU 1       Thread[Thread-3,10,main]
0       8           CPU 0       Thread[Thread-0,1,main]
0       10          CPU 1       Thread[Thread-1,1,main]
1       12          CPU 0       Thread[Thread-2,10,main]
1       10          CPU 1       Thread[Thread-1,1,main]
1       14          CPU 1       Thread[Thread-3,10,main]
1       8           CPU 0       Thread[Thread-0,1,main]
2       12          CPU 0       Thread[Thread-2,10,main]
2       14          CPU 1       Thread[Thread-3,10,main]
2       8           CPU 0       Thread[Thread-0,1,main]
2       10          CPU 1       Thread[Thread-1,1,main]
3       14          CPU 1       Thread[Thread-3,10,main]
3       8           CPU 0       Thread[Thread-0,1,main]
3       10          CPU 1       Thread[Thread-1,1,main]
3       12          CPU 0       Thread[Thread-2,10,main]
4       12          CPU 0       Thread[Thread-2,10,main]
4       14          CPU 1       Thread[Thread-3,10,main]
4       8           CPU 0       Thread[Thread-0,1,main]
4       10          CPU 1       Thread[Thread-1,1,main]
Used exection time in ms: 5
```

Fig. 1. Output of Test.java

3.2 Java Native Access (JNA)

Java Native Access provides [4] Java programs easy access to native shared libraries without using the Java Native Interface. JNA's design aims to provide native access in a natural way with a minimum of effort. The JNA library uses a small native library called foreign function interface library (libffi) to dynamically invoke native code. The JNA library uses native functions allowing code to load a library by name and retrieve a pointer to a function within that library, and uses libffi library to invoke it, all without static bindings, header files, or any compile phase. The developer uses a Java interface to describe functions and structures in the target native library. This makes it quite easy to take advantage of native platform features without incurring the high development overhead of configuring and building JNI code. JNA is built and tested on Mac OS X, Microsoft Windows, FreeBSD / OpenBSD, Solaris, and Linux. It is also possible to tweak and recompile the native build configurations to make it work on other platforms. For example, it is known to work on Windows Mobile, even if it is not tested for this platform by the development team.

If you've used the Java Native Interface (JNI) [7] to make a platform-specific native library accessible to your Java programs, you know how tedious it can be. Jeff Friesen continues his series on lesser-known open source Java projects by introducing you to Java Native Access -- a project that eliminates the tedium and error associated with JNI and lets you access C libraries programmatically. In situations where Java does not provide the necessary APIs, it is sometimes necessary to use the Java Native Interface (JNI) to make platform-specific native libraries accessible to Java programs.

JNA approaches to integrate native libraries with Java programs. It shows how JNA enables Java code to call native functions without requiring glue code in another language. It is useful to know JNA, because the Java APIs with their architecture-neutral emphasis will never support platform-specific functionality. Though Java itself is architecture-neutral, JNA is perforce on platform-specific. The Java Native Access project is hosted on Java.net, where you can download the project's online Javadoc and the software itself. Although the download section identifies five JAR files, you only need to download jna.jar. The jna.jar file provides the essential JNA software and is required to run all of the examples you'll find here. This JAR file contains several packages of classes, along with JNI-friendly native libraries for the Unix, Linux, Windows, and Mac OS X platforms. Each library is responsible for dispatching native method calls to native libraries. Here are a few things you have to take care of when starting a JNA project:

1. Download jna.jar from the JNA project site and add it to your project's build path. This file is the only JNA resource you need. Remember that jna.jar must also be included in the run-time classpath.
2. Find the names of the DLLs that your Java code will access. The DLL names are required to initialize JNA's linkage mechanisms.
3. Create Java interfaces to represent the DLLs such as kernel32.dll, user32.dll and etc on your application that will access.
4. Test linkage of your Java code to the native functions.

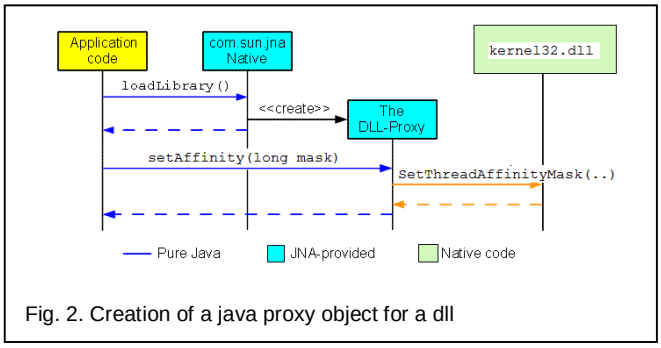


Fig. 2. Creation of a java proxy object for a dll

Many DLLs, such as those in the Windows API, host a large number of functions. But the proxy interface need only contain declarations for the methods your application actually uses.

3.2.1 A Proxy for the DLL

JNA uses the proxy pattern to hide the complexity of native code integration. It provides a factory method that Java programs use to obtain a proxy object for a DLL. The programs can then invoke the DLL's functions by calling corresponding methods of the proxy object. The code below shows an abbreviated view of a proxy interface for the Windows kernel32.dll.

Program 3. A proxy program for dll

/ / Kernel32.java

```
package javax.core;
import com.sun.jna.Library;
public interface Kernel32 extends Library {
      / / Select the CPU
     int SetThreadAffinityMask(int threadid,int mask);
     int GetCurrentThreadId();  / / Get thread Id
     int Sleep(long Milliseconds);  / / Assign waiting time
  }
```

JNA takes care of all run-time aspects, but it requires your help to create the proxy's Java class. So the first piece of code you need to create is a Java interface with method definitions that match the DLL's C functions. To play with JNA's run-time correctly, the interface must extend com.sun.jna.Library.

3.2.2 Linkage of java code to the native functions

The following KThreadCore.java uses the User32 interface shown above to create a proxy for the Windows Kernal32.DLL. When setAffinity(long mask) method is executed, which in turn invokes the DLL's SetThreadAffinityMask() function. The run-time mapping of the proxy method to the DLL function is handled transparently by JNA, the user just has to ensure that the method name matches the function name exactly.

Program 4. Linkage of java code to the native function

```
/ / KThreadCore.java
package javax.core;
import com.sun.jna.Native;
import javax.core.Kernel32;
public class KThreadCore extends Thread {
          Kernel32 kernel321=(Kernel32)
     Native.loadLibrary("kernel32", Kernel32.class);
     int mask;
public KThreadCore(){      }
public KThreadCore(String thread_Name){
          super(thread_Name);
}
public int setAffinity(int tid,int mask) {
int mask1= Runtime.getRuntime().availableProcessors();
if (mask>mask1)
 throw new IllegalArgumentException("The CPU mask should
starts from 1..N");
this.mask=mask;
return kernel321.SetThreadAffinityMask(tid,mask);
}
public int getAffinity(){
          return mask;
}
public int getCurrentThreadId(){
          return kernel321.GetCurrentThreadId();
}
public int context(int duration){
           return kernel321.Sleep(duration);
}
}
```

In Program 4, KThreadCore class extends with Thread class consequently KThreadCore class will inherit all the methods of Thread class, which will be used in Program 3. The kernel321 object is a loaded with class library that includes the declared function of Kernel32.java (Program 3). The KThreadCore() constructor will create threads without name of the thread and KThreadCore(String thread_Name) will create threads with specified of the threads using the constructor argument. The setAffinity(long tid,long mask) is alias method for SetThreadAffinityMask(long threadid,long mask) on proxy DLL. The setAffinity(long tid,long mask) method has an exception mechanism to check whether the specified CPU is available or not. If CPU is present then it will call the proxy dll SetThreadAffinityMask(long threadid,long mask) to assign the affinity for a particular thread. If CPU is not present then it will convey the exception with "The system has + mask1 + CPU only and the specified CPU + mask + is not available in the system". The getAffinity() method will return the affinity of the thread. The getCurrentThreadId() call the proxy DLL's GetCurrentThreadId(), which will assign thread id to every threads. The context(int duration) call the proxy DLL's Sleep(long Milliseconds) method, which will assign the thread to wait for specified milliseconds to avail other threads to execute on CPU.

3.2.3 Application code

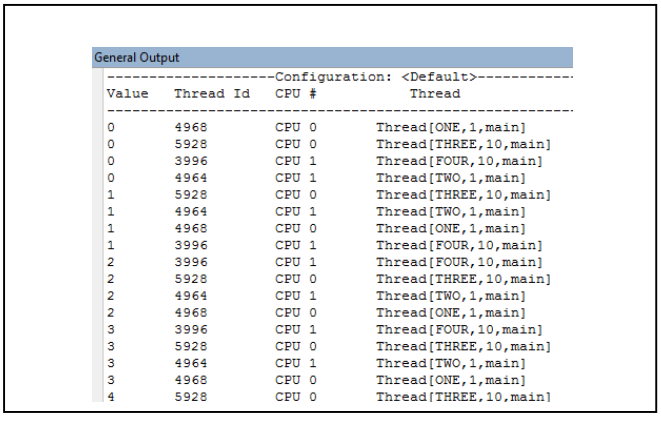
Until now, we created the proxy DLL (Program 3) and linkage of java code to the native function (Program 4). This section shows you to test the thread with an affinity on CPUs.

Program 5. Testing the affinity thread

```
/ / Test.java
import javax.core.KThreadCore;
class Test extends KThreadCore{
String name;
static int wait,affinity;
int pri;
Test(String name,int wait,int pri,int affinity) throws Exception{
          super(name);
           this.name=name;
           this.wait=wait;
           this.pri=pri;
           this.affinity=affinity;
           setPriority(pri);
           setAffinity(getCurrentThreadId(),affinity);
           start();
 }
public synchronized void run(){
try{
for(int i=0;i<5;i++){
 System.out.println(i+"\t\t"+getCurrentThreadId()
 +"\t\tCPU " +getAffinity()+"\t\t" +currentThread());
 context(wait);
 }
 }catch(Exception e){System.out.println("Error :"+e);}
}
public static void main(String aa[]){
long startTime,stopTime,elapsedTime;
try{
startTime = System.currentTimeMillis();
System.out.println("Value\tThread Id\tCPU #\t\t\tThread");
System.out.println("-----------------------------------------------------------");
```

```
//(thread name, waiting time, priority, affinity)
Test a1=new Test("ONE",5,1,0);
Test a2=new Test("TWO",5,1,1);
Test a3=new Test("THREE",5,10,0);
Test a4=new Test("FOUR",5,10,1);
// joins waiting thread through Thread class method
a1.join();  a2.join();  a3.join();  a4.join();
stopTime = System.currentTimeMillis();
elapsedTime = (stopTime - startTime) / wait;
System.out.println("Used exection time in ms: " +elapsedTime);
 }catch(Exception e){System.out.println("Error : " + e);}
 }
}
```

In Program 5, Test class extends with KThreadCore class consequently all Thread class methods can be accessed. In the main function an object is created for Test(String name,long wait,int pri,long affinity) constructor. This constructor has the argument for name of the thread, waiting time, priority, and affinity for a thread. Once an object is created, a constructor is automatically called. In the constructor super(name) method, the thread is created by passing the "name" to the KThreadCore(String thread_Name) constructor. The setPriority(pri) is actually from Thread class, which will set the priority for a thread and setAffinity ((getCurrentThreadId(),affinity) is our method from KThreadCore class, which will set the affinity for a thread, here getCurrentThreadId() method get current thread id for a thread to set affinity on CPU. Once start() method starts the thread, it will automatically call the public void run() method. The run() method contain a loop to print the value from 0 up to 5. Test class created with four objects a1, a2, a3 and a4 respectively as ONE, TWO, THREE and FOUR threads. The ONE and TWO threads have 1 (MIN_PRIORITY) priorities and the affinities of threads are 0 and 1 respectively; The THREE and FOUR threads have 10 (MAX_PRIORITY) priorities and the affinities of threads are 0 and 1 respectively. Hence, ONE and THREE threads executed on processor CPU 0 with respect to priority; similarly, TWO and FOUR threads are expected to execute on processor CPU 1 with respect to priority. This program is tested with with Inter Core™ 2 Duo CPU T8100 @ 2.10 GHz each (see Figure 3).



General Output

```
--------------------Configuration: <Default>-----------
Value   Thread Id    CPU #          Thread
-------------------------------------------------------
0       4968         CPU 0       Thread[ONE,1,main]
0       5928         CPU 0       Thread[THREE,10,main]
0       3996         CPU 1       Thread[FOUR,10,main]
0       4964         CPU 1       Thread[TWO,1,main]
1       5928         CPU 0       Thread[THREE,10,main]
1       4964         CPU 1       Thread[TWO,1,main]
1       4968         CPU 0       Thread[ONE,1,main]
1       3996         CPU 1       Thread[FOUR,10,main]
2       3996         CPU 1       Thread[FOUR,10,main]
2       5928         CPU 0       Thread[THREE,10,main]
2       4964         CPU 1       Thread[TWO,1,main]
2       4968         CPU 0       Thread[ONE,1,main]
3       3996         CPU 1       Thread[FOUR,10,main]
3       5928         CPU 0       Thread[THREE,10,main]
3       4964         CPU 1       Thread[TWO,1,main]
3       4968         CPU 0       Thread[ONE,1,main]
4       5928         CPU 0       Thread[THREE,10,main]
```

4 Results and Discussion

4.1 Thread Migration

Thread migration is when threads in computer core are able to move from core to another core. Just Peculiar Algorithm (JPA) enables to set the affinity to the thread on multi-core systems. This research finding will enables threads migration in ready, waiting and running states of threads. The following fragmented code added in Program 2 shows, how a thread can migrate between cores.

```
for(int i=0;i<5;i++)
if(i==2 and getID()=14)
 setAffinity(1);
…..
2     14     CPU 0  Thread[Thread-3,10,main]
2     10     CPU 0  Thread[Thread-1,10,main]
2     8      CPU 0  Thread[Thread-0,1,main]
3     14     CPU 1  Thread[Thread-3,10,main]
…..
```

Here, when a threadID 14 reaches the loop iteration 2, then it migrate the thread from core 0 to core 1. This is done for a single thread. But when more number of threads needed to be migrate, then like other platform thread should have a queue. For example, Linux thread migration mechanism[1], normally used for relatively long-term load-balancing across cores. To our knowledge, Linux thread migration mechanism is the current state of the art for core-switching. When a task wants to migrate, it puts itself on a per-core migration queue, wakes up and switches control to a per core migration thread, which does the actual work of moving the thread to the run queue of the target core.

4.2 The Performance issue

The performance of Just Peculiar Algorithm (JPA) (described in secretion 3.1) and Java Native Access (JNA) (described in section 3.2) evaluated on different multi-core environment by selecting different affinity on CPU and adding few more threads with different iterations.

TABLE 1
EXECUTION TIME IN MILLISECONDS

Methods		Core 2 Duo				Core i5				Core i7			
		Iterations											
		5	25	50	100	5	25	50	100	5	25	50	100
Four threads	JPA	5	25	96	261	5	26	51	102	5	29	62	107
		5	25	97	265	5	27	52	101	5	29	59	102
		5	25	102	260	5	26	51	102	5	31	61	103
		5	26	101	262	5	26	51	102	5	28	62	105
	JNA	26	44	94	248	29	54	78	126	31	55	79	130
		25	47	90	242	32	53	76	127	32	56	88	132
		27	46	92	246	31	50	77	130	33	57	91	133
		28	53	95	267	28	51	74	128	34	64	81	129
Seven threads	JPA	5	79	222	504	3	25	49	100	5	48	51	101
		7	82	225	510	4	28	50	104	5	50	99	109
		5	78	229	507	5	30	53	106	4	51	102	103
		5	81	218	503	5	29	55	103	5	52	52	105
	JNA	26	83	207	476	43	106	184	343	43	106	181	343
		27	73	210	471	46	109	187	340	46	109	184	340
		37	81	212	494	43	107	190	343	43	107	187	343
		36	95	213	493	46	108	185	340	46	108	190	340

We analyze about the execution on different computer core with the time in milliseconds, which is described in the above Table 1. Just Peculiar Algorithm (JPA) is just a techniques to set the affinity through Java Thread library, hence the execution speed is somewhat little than Java Native Access (JNA). Because, the Java Native Access (JNA) is approaching windows DLL to perform the affinity schedule on the processor, hence it acquired more time than Just Peculiar Algorithm (JPA).

5 CONCLUSION

Working with multiple threads on symmetric multiprocessor is very natural to improve the performance based on number of CPUs. Thread affinity benefit a thread to run on a specific subset of processors, enable us to schedule threads on a particular CPU. Setting an affinity to CPU is not new to research. Since affinity thread is already sustained by other multiprogramming languages on different platforms. Java does not have method to set affinity for a thread. While Java exercised already with affinity threads in UNIX platforms through Java Native Interface. There is lack in windows platform for Java affinity thread. This paper exemplified the method to set an affinity for threads to execute on particular CPU through Java Java Peculiar Algorithm (JPA) and Native Access (JNA) in windows platforms. The performance metric also deliberated.

REFERENCES

1. ACM SIGOPS Operating Systems Review special issue on The Interaction Among the OS, the Compiler, and Multicore Processors, April, 2009, Volume 43, Number 2.
2. API reference, http://docs.oracle.com/javase/7/docs/api/index.html
3. Brian goetz, tim peierls, joshua bloch, joseph bowbeer, david holmes, doug lea, java concurrency in practice, addison wesley professional, may 09, 2006, isbn-10: 0-321-34960-1
4. Get started with jna, https://jna.dev.java.net
5. Herbert schildt, java™ 2: the complete reference, fifth edition, mcgraw-hill, 2002, isbn:0-07-222420-7
6. Hesham el-rewini, mostafa abd-el-barr, 2005, advanced computer architecture and parallel, a john wiley & sons, inc publication.
7. Java Native Interface, http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html
8. Microsoft developer network (msdn): setprocessaffinitymask http://msdn.microsoft.com/library/default.asp?url=/library/enus/dllproc/base/setprocessaffinitymask.asp
9. Mike loukides, gian-paolo d. Musumeci, system performance tuning, second edition, o'reilly, 2002, isbn : 0-596-00284-x
10. parallel programming? Well, it's all about cpu affinity or how to set processor affinity in wpf http://khason.net/blog/parallel-programming-well-it%e2%80%99s-all-about-cpu-affinity-or-how-to-set-processor-affinity-in-wpf/
11. Take charge of processor affinity, http://www.ibm.com/developerworks/linux/library/l-affinity/index.html
12. The win32 programming tutorials for fun (and funny too), http://www.installsetupconfig.com/win32programming/