International Journal of Computer Science Trends and Technology (IJCST) – Volume 3 Issue 2, Mar-Apr 2015

RESEARCH ARTICLE OPEN ACCESS

Handling Multithreading Approach Using Java

Nikita Goel, Vijaya Laxmi, Ankur Saxena
Amity University
Sector-125, Noida
UP-201303 - India

ABSTRACT
This paper contains information on multithreading in java and application based on it. In this report we discuss the use of multithreading, its types and its methods. Java is a multithreaded programming language. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources. Multithreading is based on the idea of multitasking in applications where specific operations within a single application are subdivided into individual threads. With multithreading each of the threads can run in parallel. The operating system divides processing time not only among different applications, but also among each thread within an application. Application based on multithreading is developed using Eclipse software. Eclipse consists of a base workspace and an extensible plug-in system for customizing the environment. It is written mostly in Java.
Keywords:- Runnable, Thread, Multithreading, Thread Lifecycle, Thread priority, Eclipse

I. INTRODUCTION

Java is one of the most commonly used and mature programming languages for building enterprise applications. Over the years, Java development has evolved from small applets run on a Web browser to large enterprise distributed applications run on multiple servers. Now, Java has three different platforms, or flavors, and each addresses certain programming requirements.

Java is a multithreaded programming language which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs. [1]

A **thread** is a line of execution and it is the smallest unit of code that is dispatched by the scheduler. It is an independent part of a program which is divided in such a way that each part is independent of the other, if there occurs an exception in one thread, it doesn't affect other threads. A thread is a dispatchable unit of work. Threads are light-weight processes within a process. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Multithreading is a technique that allows a program or a process to execute many tasks simultaneously. It allows a process to run its tasks in parallel mode on a single processor system. Multithreading is performed to save CPU'S deal time or waiting time. It helps to increase the efficiency of CPU. When a program requires user input, multithreading enables creation of a separate thread for this task alone. The main thread can continue with the execution of the rest of the program. Programs not using multithreading will have to wait until the user has input a value for the continuation of the execution of the program. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application. [2]

LIFE-CYCLE OF A THREAD

A thread follows a life cycle, which starts from the creation till it is destroyed. The life cycle of a thread in java is controlled by Java Virtual Machine (JVM). The 5 states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5.Terminated

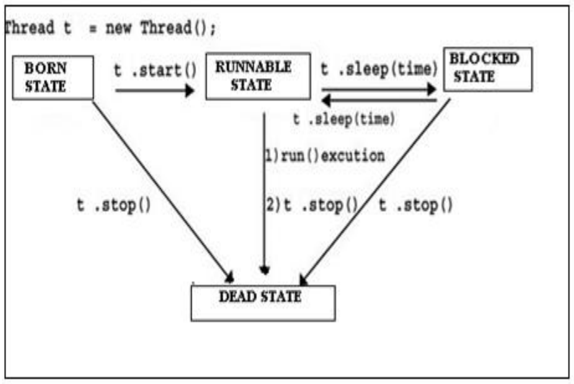


Figure 1: Life cycle of a thread

New State: This is the first state when thread object is created. Thread will be in this state until the run method is not called off. This state is also referred to as a born thread.

Runnable: After a newly born thread is started using start () method, the thread becomes runnable with the using of run () method. The thread is in runnable state but the thread scheduler has not selected it to be the running thread.

Running: The thread is in running state if the thread scheduler has selected it.

Blocked: In this state the thread is in the memory but is temporarily suspended. Blocked state is divided into Waiting and Time Waiting State.

Waiting: Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. The thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transition back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Dead: A running method ends its life when it has finished executing its run () method. It is a natural death. A thread can also be killed by using stop () method, but this throws an exception that's subclass of error, throwing an exception should be a special event and not part of normal program execution; thus the use of stop () method is discouraged.

COMMON THREAD METHODS

• **start():** This method starts the thread in a separate path of execution and then invokes the run() method on this Thread object.

• **run():** When the thread object is instantiated using start () , the thread is made runnable using the run() method.

• **suspend():** This method suspends the invoking object. The thread will become runnable again with the resume () method.

• **sleep()**: This method suspends execution of the thread for the specified number of milliseconds. It can throw an InteruptedException.

• **resume()**: This method restarts the suspended thread from the point where it was halted. The resume() method is called by some thread outside the suspended one.

• **stop()**: The stop() method is used to stop the execution of a thread before its run() method terminates. However, the use of stop () is discouraged because it does not always stop a thread.

CREATION OF A THREAD IN JAVA

Threads are objects in the Java language. They can be created by using two different mechanisms:

1. Create a class that extends the standard Thread class.

2. Create a class that implements the standard Runnable interface.

That is, a thread can be defined by extending the java.lang.Thread class or by implementing the java.lang.Runnable interface.

➢ EXTENDING THE THREAD CLASS:

In this method a thread can be created by defining a class that is a derived from the Thread class which is built into Java. An object of this derived class will be a thread.

The run () method should be overridden and should contain the code that will be executed by the new thread. This method must be public with a void return type and should not take any arguments.

The steps for creating a thread by extending Thread class are:

1. Create a class by extending the Thread class and override the run () method:

```
class MyThread extends Thread {

public void run () {

// thread body of execution

}

}
```

2. Create a thread object:

```
MyThread thr1 = new MyThread();
```

3. Start Execution of created thread:

```
thr1.start();
```

➢ IMPLEMENTING RUNNABLE INTERFACE:

Creation of a thread object using Runnable interface is the easiest way of creating a Thread object. The Runnable interface contains only run () method, which should be included in classes implementing them.

The steps for creating a thread by implementing runnable interface are:

1. Create a class that implements the interface Runnable and override run () method:

```
class MyThread implements Runnable {

public void run() {

// thread body of execution

}

}
```

2. Creating Object:

```
MyThread myObject = new MyThread();
```

3. Creating Thread Object:

```
Thread thr1 = new Thread(myObject);
```

4. Start Execution:

```
thr1.start();
```

THREAD PRIORITIES

In Java, all the thread instances that the developer creates have the same priority, which the processor will schedule without any specific order.

It is important for different threads to have different priorities. Important threads should always have higher priority than less important ones, while threads that need to run quietly may only need the lowest priority.

For example, the garbage collector thread, this thread just needs the lowest priority to execute, which means it will not be executed before all other threads are scheduled to run. Thread priority can be controlled using the java API'S, the Thread.setPriority() method serves this purpose.

Three constants values defined in Thread class are:

- **MIN_PRIORITY =1**
- **NORM_PRIORITY=5**
- **MAX_PRIORITY=10**

The priority range of the thread should be between the minimum and the maximum number.

II . RELATED WORK

A thread of execution by definition is a "fork" of a computer program into two or more concurrently running tasks [3]. The implementation of threads and processes differs from one operating system to another operating system. In most cases, a thread is contained inside a process. Multiple threads exist within the same process and share resources such as memory, but different processes do not share this data.

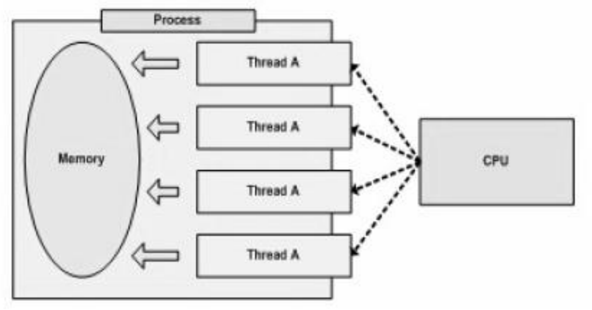


Figure 2: Four threads running on a single core processor sharing the same resources

The threaded programming model provides an abstraction of concurrent execution, and when applied to a single process it enables parallel execution on a multi-core system. This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines. This is because the threads of the program naturally lend themselves to truly concurrent execution [3].

An example that illustrates the above idea is shown in

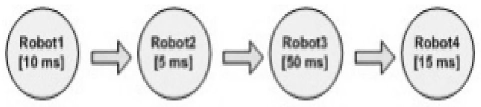


Figure 3: Multi mobile robot simulation sequence

In the non threaded approach when dealing with simulating multi robot systems, the robots are put in an array, and then sequentially the CPU resources are passed to each of them, in their turn, to calculate their actions, and when each of them finishes, then the available resources are passed to the next robot, and so on, until all robots in the array do their jobs. Then this cycle starts again, from the first robot to decide its next step, and so on. The performance of this approach is acceptable when all robots are of the same type, or if they need similar time to complete their decision making process. In the case of a scenario like the one shown in Figure 3, in which we are simulating four robots at the same time (each of them needing a different time to accomplish its control strategy calculations), we encounter the following situation: the Robot3 will need 50 milliseconds to finish the calculations in order to take a decision, and this will affect the simulated Robot4, since it cannot start its own calculations until Robot3 finishes. This situation causes a lag that the Robot4 is not responsible of. This is one shortcoming of not using threads in the programming. Threading will eliminate this problem even when we only have single core processor, because if we use multithreading and assign a separate thread to each simulated robot, then the slow robot (Robot3) will not slow down all the simulation and other robots till it finishes its calculations. Instead, each robot will only affect itself, and the robot with easy calculations will be simulated normally as it should be, while the complex robot will stay in its place until finishing its calculations without affecting the other robots on their simulation.

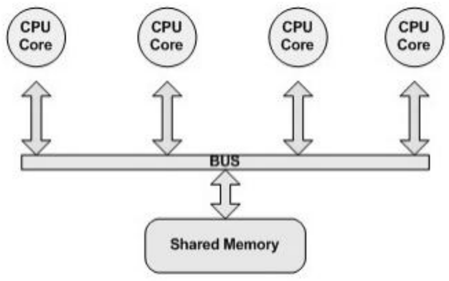


Figure 4: : A Multi-Core Processor architecture

On single processor systems, the threads are executed sequentially and the processor switches between the threads quickly enough that both processes appear to occur simultaneously. In multi-core processors threads can run more or less independently of each other without requiring thread switches to get at the resources of the processor.

The approach of making a separate thread for each mobile robot is most suitable when we simulate heterogeneous robots, because each of them needs a different amount of calculations. The operating system will switch between threads on a time basis. This way, it assures a fair distribution of processing resources among the threads, assuming we give all the robot threads the same priority. We can think of the robot threads as agents (Figure 5), and the set of robots is a multi-agent system, this is because each robot thread meets the requirement for being considered as an agent, since it is autonomous (or partly autonomous depending on the user needs), each has its own local view to the world, and decentralization is met, since there is no controlling agent, each agent acts separately, reading from its environment through its sensors, and then sends its actions to the actuators as commands according to the duty (task) that the agent has to perform [4].

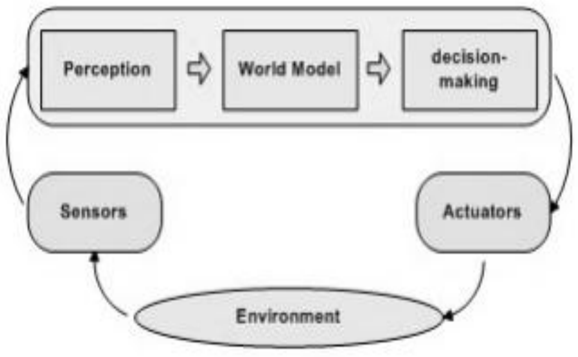


Figure 5: A mobile robot as an agent

In the non threaded approach when dealing with simulating multi robot systems, the robots are put in an array, and then sequentially the CPU resources are passed to each of them, in their turn, to calculate their actions, and when each of them finishes, then the available resources are passed to the next robot, and so on, until all robots in the array do their jobs [3]. Then this cycle starts again, from the first robot to decide its next step, and so on [5].

The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For an instance, when there are two subsystems within a program that can execute concurrently, make them individual threads .With the help of multithreading very efficient programs can be created. If a number of threads are created, the performance of your program is actually degraded [6].

III. IMPLEMENTATION

Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment. [5]

An application has been developed with a class name called message to contain method dips, this method accepts a string argument and displays the same ,it should wait for 1000 miliseconds and print "end of message". After creating a class sender which creates threads within its constructor and start then the constructor accepts a message object and a string as parameter. Finally we create a class syntest which creates objects of sender class, with different strings.

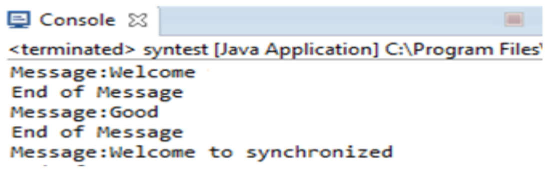
```
class Message

{

void dips (String m)

{

System.out.println ("Message:" +m);

try

{

Thread.sleep (500);

} catch (InterruptedException e) {}

System.out.println ("End of Message");

}

}
```

The class Message contains a Method, which accepts a string as argument and display the same. It waits for 5 seconds and prints the last print statement.

```
class Sender implements Runnable

{

Message m;

Thread t;

String s;

public Sender (Message g, String i)

{

m=g;

s=i;

t=new Thread (this);

t.start ();

}

public void run ()

{

synchronized (m)

{

m.disp(s);

}
}

}
```

The sender class is responsible for creating threads. Its constructor takes a message object and a string as argument.

```
class syntest
{
public static void main (String a [])
{
Message e=new Message ();
Sender s1=new Sender (e,"Welcome");
Sender s2=new Sender (e,"Welcome to synchronized");
Sender s3=new Sender (e,"Good");
try
{
s1.t.join ();
s2.t.join ();
s3.t.join ();
} catch (InterruptedException o) {}
}
}
```



```
Console
<terminated> syntest [Java Application] C:\Program Files\
Message:Welcome
End of Message
Message:Good
End of Message
Message:Welcome to synchronized
```

Figure 6: Output of application

The main method of syntest class creates objects s Sender class which it turns creates threads. The respective messages are displayed, not necessarily in order. At the end of this display, all the child threads are destroyed and the program terminates.

IV. FUTURE WORK AND REFERENCES

Multithreaded software applications are programs that run multiple tasks (threads) at the same time to increase performance for heavy workload scenarios, are already positioned to take advantage of multi-core processors. The implementation of threads and processes differs from one operating system to another. The use of multithreading programming is the key to take advantage of the increasing number of processing cores in central processing units in each new generation of processors.

V. CONCLUSION

This study is based on multithreading in java. This study is based on multithreading in java in which various aspects of multithreading has been covered which comprises of multithreading and its states and its methods . A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources. Multithreading is based on the idea of multitasking in applications where specific operations within a single application are subdivided into individual threads. With multithreading each of the threads can run in parallel. The operating system divides processing time not only among different applications, but also among each thread within an application.

Multithreading has many advantages for application operations where it's okay to interleave user actions, making a GUI multithreaded can be very useful. It can even help to improve end user productivity by allowing a number of application operations to proceed in parallel. However, the power that multithreaded programming provides should be used with care. If you use shared data across threads, then it's quite easy to get into some very difficult debugging scenarios.

Application based on multithreading is developed using ECLIPSE software. Eclipse is an integrated development environment (IDE) to develop applications using java.

REFERENCES

[1] K. P. &. D. Sharma, "Multithreading In Java," *International Journal of Research (IJR)*, Vols. Vol-1, no. Issue-10, November 2014.

[2] F. F. E. a. I. P. Mondada, "Mobile robot miniaturisation: A tool for investigation in control algorithms." Proceedings of the 3rd International Symposium on Experimental Robotics," *Kyoto, Japan*, 1993.

[3] B. Lewis, "Threads Primer: A Guide to Multithreaded Programming," *Prentice Hall*, 1995.

[4] I. Fadi1, O. Olumide2 and I. Dumitrache, "On multi robot simulation: multi-threading approach for implementation," *GESJ: Computer Science and Telecommunications*, 2011.

[5] Gerkey, B. P., Vaughan, R. T. and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," *Coimbra, Portugal*, pp. 317-323, 203.

[6] H. Schildt, Java:The complete reference, 2007.

[7] G. Pandey and D. Dani, "Android Mobile Application Build on Eclipse," *International Journal of Scientific and Research Publications,*, vol. 4, no. 2, February 2014.