Multi Threading

Krishantha Dinesh
@ 2013-Oct

2000 West Park Drive
Westborough MA 01581 USA
Phone: 508 389 7300 Fax: 508 366 9901

virtusa®
Accelerating Business Outcomes

Objectives

- Not to teach Java

virtusa

Prerequisites

- Java.basicknowladge >(60/100);
- Mobilephone.soundmode=soundmode.completesilent && soundmode.completesilent != soundmode.vibrate

Objectives

- To understand the purpose of multithreading
- To describe Java's multithreading mechanism
- To explain concurrency issues caused by multithreading
- To outline synchronized access to shared resources

What is Multithreading?

- Multithreading is similar to multi-processing.
- A multi-processing Operating System can run several processes at the same time and each process has its own address/memory space
- The OS's scheduler decides when each process is executed
- Only one process is actually executing at any given time. However, the system appears to be running several programs simultaneously
- Separate processes to not have access to each other's memory space
- In a multithreaded application, there are several points of execution within the same memory space.
- Each point of execution is called a thread
- Threads share access to memory

virtusa

Why use multithreding

- In a single threaded application, one thread of execution must do everything
- If an application has several tasks to perform, those tasks will be performed when the thread can get to them.
- A single task which requires a lot of processing can make the entire application appear to be "sluggish" or unresponsive.
- In a multithreaded application, each task can be performed by a separate thread
- If one thread is executing a long process, it does not make the entire application wait for it to finish.
- If a multithreaded application is being executed on a system that has multiple processors, the OS may execute separate threads simultaneously on separate processors.

virtusa

How it is worked

- Each thread is given its own "context"
- A thread's context includes virtual registers and its own calling stack
- The "scheduler" decides which thread executes at any given time
- Since many OS now directly support multithreading, the VM may use the system's scheduler for scheduling threads
- The scheduler maintains a list of ready threads (the run queue) and a list of threads waiting for input (the wait queue)
- Each thread has a priority. The scheduler typically schedules between the highest priority threads in the run queue
- Note: the programmer cannot make assumptions about how threads are going to be scheduled. Typically, threads will be executed differently on different platforms.

virtusa

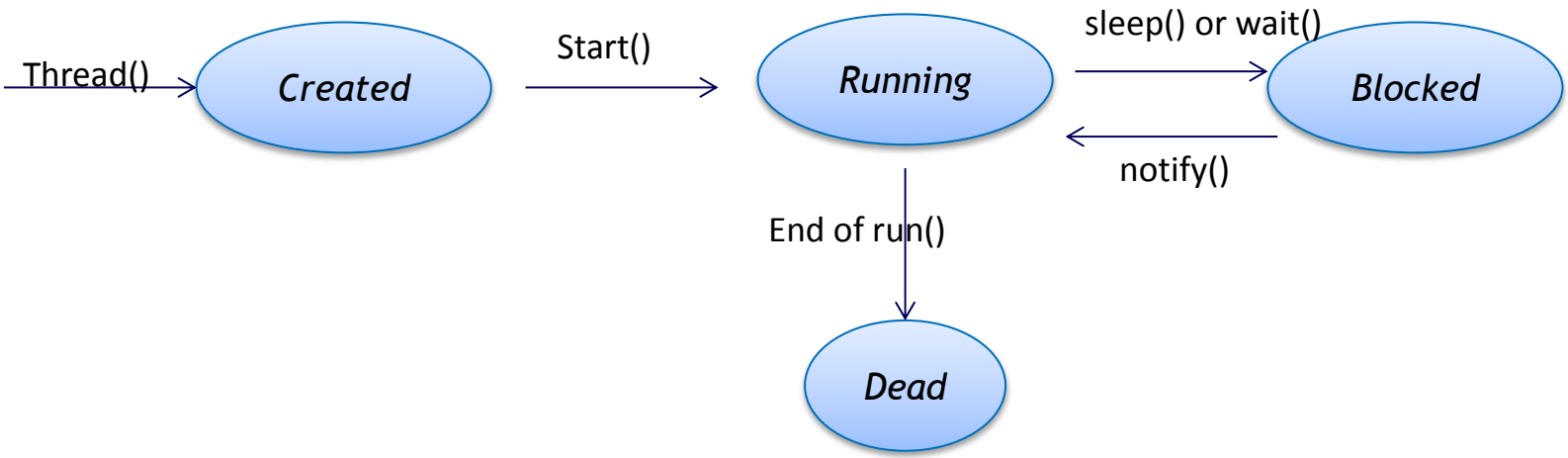
Multithreading in java

- Few programming languages directly support threading
- Although many have add-on thread support
- Add on thread support is often quite cumbersome to use
- The Java Virtual machine has its own runtime threads
- Threads are represented by a Thread class
- A thread object maintains the state of the thread
- It provides control methods such as interrupt, start, sleep, yield, wait
- When an application executes, the main method is executed by a single thread.
- If the application requires more threads, the application must create them.

virtusa

States

- Threads can be in one of four states
  - Created, Running, Blocked, and Dead
- A thread's state changes based on:
  - Control methods such as start, sleep, yield, wait, notify
- Termination of the run method



How does thread run

- The thread class has a run() method
- run() is executed when the thread's start() method is invoked
- The thread terminates if the run method terminates
- To prevent a thread from terminating, the run method must not end
- run methods often have an endless loop to prevent thread termination
- One thread starts another by calling its start method
- The sequence of events can be confusing to those more familiar with a single threaded model.

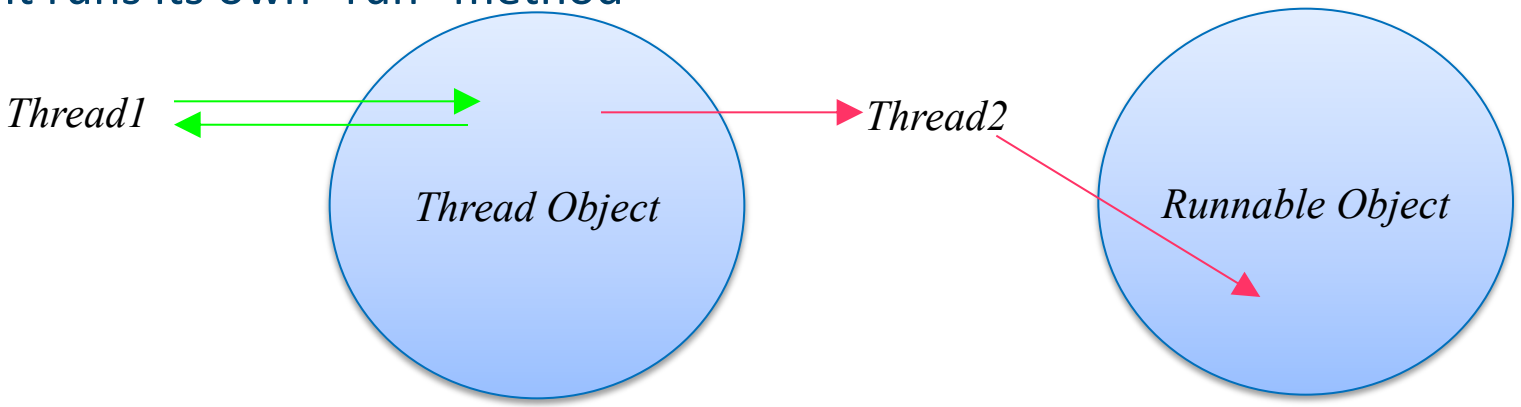
Create own threads

- The obvious way to create your own threads is to subclass the Thread class and then override the run() method
- This is the easiest way to do it also It is not the recommended way to do it.
- Because threads are usually associated with a task, the object which provides the run method is usually a subclass of some other class
- If it inherits from another class, it cannot inherit from Thread.
- The solution is provided by an interface called Runnable.
- Runnable defines one method - public void run()
- One of the Thread classes constructor takes a reference to a Runnable object
- When the thread is started, it invokes the run method in the runnable object instead of its own run method.

virtusa

Runnable

- In the example below, when the Thread object is instantiated, it is passed a reference to a "Runnable" object
- The Runnable object must implement a method called "run"
- When the thread object receives a start message, it checks to see if it has a reference to a Runnable object:
- If it does, it runs the "run" method of that object
- If not, it runs its own "run" method



example

```
package com.virtusa.training.java.thread;

public class ThreadSample implements Runnable {

    private Thread thisThread;

    public void start() {

        if (thisThread == null) {
            thisThread = new Thread();
            thisThread.start();
        }
    }

    @Override
    public void run() {
        // this run on its own thread
    }
}
```

Termination

- In Java 1.1, the Thread class had a stop() method
- One thread could terminate another by invoking its stop() method.
- However, using stop() could lead to deadlocks
- The stop() method is now deprecated. DO NOT use the stop method to terminate a thread
- The correct way to stop a thread is to have the run method terminate
- Add a boolean variable which indicates whether the thread should continue or not
- Provide a set method for that variable which can be invoked by another thread

virtusa

Stop - example

```java
package com.virtusa.training.java.thread;

public class TerminationSample implements Runnable {

    private Thread thisThread;
    private boolean isToStop;

    public void setToStop(boolean isToStop) {
        this.isToStop = isToStop;
    }

    public void start() {

        if (thisThread == null) {
            thisThread = new Thread();
            thisThread.start();
        }

    }

    @Override
    public void run() {
        while (true) {
            if (isToStop)
                break;
        }

    }
```

Create multiple Threads

- The previous example illustrates a Runnable class which creates its own thread when the start method is invoked.
- If one wished to create multiple threads, one could simple create multiple instances of the Runnable class and send each object a start message
- Each instance would create its own thread object
- Is the a maximum number of threads which can be created?
- There is no defined maximum in Java.
- If the VM is delegating threads to the OS, then this is platform dependent.
- A good rule of thumb for maximum thread count is to allow 2Mb of ram for each thread
- Although threads share the same memory space, this can be a reasonable estimate of how many threads your machine can handle.

virtusa

Priorities

- Every thread is assigned a priority (between 1 and 10) The default is 5
- The higher the number, the higher the priority Can be set with setPriority
- The standard mode of operation is that the scheduler executes threads with higher priorities first.
- This simple scheduling algorithm can cause problems. Specifically, one high priority thread can become a "CPU hog".
- A thread using vast amounts of CPU can share CPU time with other threads by invoking the yield() method on itself.
- The more CPU a thread receives, the lower its priority becomes
- The more a thread waits for the CPU, the higher its priority becomes
- Because of thread aging, the effect of setting a thread's priority is dependent on the platform

virtusa

Yield() and sleep()

- Sometimes a thread can determine that it has nothing to do
- Sometimes the system can determine this. ie. waiting for I/O
- When a thread has nothing to do, it should not use CPU
- This is called a busy-wait.
- Threads in busy-wait are busy using up the CPU doing nothing.
- Often, threads in busy-wait are continually checking a flag to see if there is anything to do.
- It is worthwhile to run a CPU monitor program on your desktop
- You can see that a thread is in busy-wait when the CPU monitor goes up (usually to 100%), but the application doesn't seem to be doing anything.

virtusa

Cont..

- Threads in busy-wait should be moved from the Run queue to the Wait queue so that they do not hog the CPU
- Use yield() or sleep(time)
- Yield simply tells the scheduler to schedule another thread
- Sleep guarantees that this thread will remain in the wait queue for the specified number of milliseconds.

Concurrent access

- Those familiar with databases will understand that concurrent access to data can lead to data integrity problems
- Specifically, if two sources attempt to update the same data at the same time, the result of the data can be undefined.
- The outcome is determined by how the scheduler schedules the two sources.
- Since the schedulers activities cannot be predicted, the outcome cannot be predicted
- Databases deal with this mechanism through "locking"
- If a source is going to update a table or record, it can lock the table or record until such time that the data has been successfully updated.
- While locked, all access is blocked except to the source which holds the lock.
- Java has the equivalent mechanism. It is called synchronization

Synchronization

- To obtain the lock, you must synchronize with the object
- The simplest way to use synchronization is by declaring one or more methods to be synchronized
- When a synchronized method is invoked, the calling thread attempts to obtain the lock on the object.
- if it cannot obtain the lock, the thread goes to sleep until the lock becomes available
- Once the lock is obtained, no other thread can obtain the lock until it is released. ie, the synchronized method terminates
- When a thread is within a synchronized method, it knows that no other synchronized method can be invoked by any other thread
- Therefore, it is within synchronized methods that critical data is updated

virtusa

Provide thread safe

- If an object contains data which may be updated from multiple thread sources, the object should be implemented in a thread-safe manner
- All access to critical data should only be provided through synchronized methods (or synchronized blocks).
- In this way, we are guaranteed that the data will be updated by only one thread at a time.

```
package com.virtusa.training.java.thread;

public class ThreadSafe {

    private double balance;

    public synchronized void withdraw(float amount)
    {
        if ((amount>0.0) && (amount<=balance))
            balance = balance - amount;
    }

    public synchronized void deposit(float anAmount)
    {
        if (anAmount>0.0)
            balance = balance + anAmount;
    }

}
```

Consequences

- Many threads may be waiting to gain access to one of the object's synchronized methods
- The object remains locked as long as a thread is within a synchronized method.
- Ideally, the method should be kept as short as possible.
- Another solution is to provide synchronization on a block of code instead of the entire method
- In this case, the object's lock is only held for the time that the thread is within the block.
- The intent is that we only lock the region of code which requires access to the critical data. Any other code within the method can occur without the lock.
- In high load situations where multiple threads are attempting to access critical data, this is by far a much better implementation

virtusa

solution

```
package com.virtusa.training.java.thread;

public class ThreadSafe {

    private double balance;

    public void withdraw(float amount) {

        synchronized (this) {
            if (amount <= balance)
                balance = balance - amount;
        }
    }

    public void deposit(float anAmount) {

        synchronized (this) {
            balance = balance + anAmount;
        }

    }
}
```

Thread sleep example

```java
package com.virtusa.training.java.thread.sleep.D6;

public class ThreadClass implements Runnable {

    @Override
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
package com.virtusa.training.java.thread.sleep.D6;

public class ThreadRunner {

    public static void main(String[] args) {
        ThreadClass fromThread = new ThreadClass();

        new Thread(fromThread, "Nala").start();

        new Thread(fromThread, "Damayanthi").start();

    }

}
```

Yield example

```java
package com.virtusa.training.java.thread.yield.D5;

public class ThreadClass implements Runnable {

    @Override
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
            Thread.yield();
        }
    }
}
```

```java
package com.virtusa.training.java.thread.yield.D5;

public class ThreadRunner {

    public static void main(String[] args) {
        ThreadClass fromThread = new ThreadClass();

        new Thread(fromThread, "Nala").start();

        new Thread(fromThread, "Damayanthi").start();

    }

}
```

Termination example

```java
package com.virtusa.training.thread.terminate;

public class ThreadTermination implements Runnable {

    private boolean isStop;

    @Override
    public void run() {

        while (!isStop) {
            System.out.println("i am a live");

        }
        System.out.println("ooops i am dead");

    }

    public void setStop(boolean isStop) {
        this.isStop = isStop;
    }

}
```

```java
package com.virtusa.training.thread.terminate;

public class ThreadRunner {

    public static void main(String[] args) {
        ThreadTermination fromThread = new ThreadTermination();
        Thread thread = new Thread(fromThread, "Nala");
        thread.start();
        for (int i = 0; i < 1000000; i++) {

            if (i == 50000) {
                fromThread.setStop(true);
            }
        }

    }

}
```

Yield tester

```java
package com.virtusa.training.thread.yield;

public class YieldSampleClass implements Runnable {

    private static int Ganga,yamuna;
    private String name;
    @Override
    public void run() {

        while (true) {
            name=(Thread.currentThread().getName());

            if (name.equals("Ganga"))
            Ganga++;
            else
                yamuna++;

            System.out.println("G: "+Ganga + "="+"Y: "+yamuna);
            Thread.yield();

        }

    }

}
```

```java
package com.virtusa.training.thread.yield;

public class ThreadRunner {

    public static void main(String[] args) {
        YieldSampleClass fromThread = new YieldSampleClass();


        Thread thread = new Thread(fromThread, "Ganga");
        Thread thread2 = new Thread(fromThread, "Yamuna");

        thread.setPriority(1);
        thread2.setPriority(10);

        thread.start();
        thread2.start();
    }
}
```