Multithreading

Introduction

Imagine a bank teller with a single queue of customers both waiting to deposit cash and withdraw cash. The first customer wishes to withdraw fairly huge sum of money which the bank currently does not possess in its chest. At the same time a customer at the end of the queue is awaiting his turn to deposit cash which would be accidentally more than sufficient to pay the first customer. But he never gets his turn, for the first customer never clears the queue waiting for his withdrawal to be processed, resulting in deadlock. If the bank were to implement independent queues for withdrawal and receipts in the first place this problem would not have occurred.

Imagine a stockbroker application with a lot of complex behavior that the user initiates. One of the applications is “download last stock option prices,” another is “check prices for warnings,” and a third time-consuming operation is, “analyze historical data for company XYZ.” In a single-threaded runtime environment, these actions execute one after another. The next action can happen only when the previous one is finished. If a historical analysis takes half an hour, and the user selects to perform a download and check afterward, the warning may come too late to, say, buy or sell stock as a result. We just imagined the sort of application that cries out for multithreading. Ideally, the download should happen in the background (that is, in another thread). That way, other processes could happen at the same time so that, for example, a warning could be communicated instantly. All the while, the user is interacting with other parts of the application. The analysis, too, could happen in a separate thread, so the user can work in the rest of the application while the results are being calculated.

Multithreading can help in the above 2 scenarios.

Multithreading is a specialized form of Multitasking. You are almost certainly acquainted with Multitasking, because it is supported by virtually all modern operating systems. However, there are two distinct types of Multitasking.

1. Process-based multitasking

This is the feature that allows your computer to run two or more programs concurrently. For example, a web server runs at the same time you are using a browser. In this type, a program is the smallest unit of code dispatched by the scheduler.

2. Thread-based multitasking

This is the feature that allows your program to execute two or more functions concurrently. For example, your text editor can be formatting text the same time it is printing, as long as these two actions are performed by two different threads. Here thread is the smallest unit of code dispatched by the scheduler.

Processes are heavyweight. They require separate address spaces. Inter process communication is expensive and limited. Context switching from one process to another is also costly.

Threads are lightweight. Within a process, they share the same address space and therefore context switching between one thread to another thread is less expensive. Inter thread communication also involves less overhead.

While Java makes use of the process based multitasking features provided by the operating systems, process based multitasking is not in the hands of Java. However support for thread-based multitasking is embedded into the language itself. Java is one language developed with multithreading in mind from the beginning. Multithreading allows you to write efficient programs that make maximum use of CPU, because idle time is kept to a minimum. This is especially important for interactive (user response), network environment in which Java operates. Any application you are working on that requires two or more things to be done at the same time is probably a great candidate for multithreading. It is rightly said that one has to stop thinking procedurally and start thinking objectively with the advent of OOPS. Now is the time to think concurrently.

Thread can be formally termed as "thread of control".

In Java, “thread” means two different things:

- An instance of class java.lang.Thread

- A thread of execution or thread of control

An instance of Thread is just an object. Like any other object in Java, it has variables and methods, and lives and dies on the heap. But a thread of execution is an individual process (a “lightweight” process) that has its own call stack. In Java, there is one thread per call stack - or, to think of it in reverse, one call stack per thread. Even if you don’t create any new threads in your program, threads are back there running.

The main() method that starts the whole ball rolling runs in one thread, called the main thread. If you looked at the main call stack (and you can, anytime you get a stack trace from something that happens after main begins, but not within another thread) you’d see that main() is the first method on the stack - the method at the bottom. But as soon as you create a new thread, a new stack materializes and methods called from that thread run in a call stack that’s separate from the main() call stack. That second new call stack is said to run concurrently with the main thread.

You might find it confusing that we’re talking about code running concurrently - as if in parallel - yet you know there’s only one CPU on most of the machines running Java. The JVM, which gets its turn at the CPU by whatever scheduling mechanism the underlying OS uses, operates like a mini-OS and schedules its own threads regardless of the underlying operating system. In some JVMs, the java threads are actually mapped to native OS threads.

Note: Don’t make the mistake of designing your program to be dependent on a particular implementation of the JVM. Different JVMs can run threads in profoundly different ways. For example, one JVM might be sure that all threads get their turn, with a fairly even amount of time allocated for each thread in a nice, happy, round-robin fashion. But in other JVMs, a thread might start running and then just hog the whole show, never stepping out so others can have a turn.

You can define and instantiate a thread in one of two ways:

- Extend the java.lang.Thread class
- Implement the Runnable interface

Thread Class

Code Listing 4.1

```
class threadapp extends Thread
{
   String name_of_thread;

   threadapp(String s)
   {
      name_of_thread = s;
   }

   public void run()
   {
      try
      {
         for(int i = 0; i < 3; i++)
         {
            System.out.println(name_of_thread + ": " + i);
            Thread.sleep(1500);
         }
      }
      catch(InterruptedException ie) {}
   }
}

class Demo
```

```
{
   public static void main(String[]args)
   {
      threadapp  arr[] = new threadapp[3];

      for(int i = 0; i < 3; i++)
      {
         arr[i] = new threadapp("Thread " + i);
         arr[i].start();
      }
   }
}
```

- Any class that is Threadable should extend either Thread Class or implement its companion interface Runnable. It is just a matter of personal taste and does in no way alter the effectiveness of the program. In fact Thread class itself implements Runnable. Either way one should override/define the run() method. The code in this method is what gets executed concurrently. In fact the run() method can be thought of as the main() method of the newly formed thread. A newly formed thread begins execution with the run() method in the same way a program begins execution with the main() method. Whether you implement multithreading or not there always is one thread running in your program, the main thread.

- Then we instantiate the first object of this class in the main-thread. In the constructor, towards the end of initialization process we attempt to invoke the start() method of the Thread Class. The start() method performs system specific details for the proper functioning of thread, invokes the run() method and most importantly "returns", even before the execution of run() method is completed / started. Note that super() creates a Thread object while start() creates a "thread of control".

- Now we have two thread of controls performing their tasks independently. One the main thread and the other, the thread that is executing the run() method for the first object. While main thread is busy instantiating, invoking/executing the run () method for the second object, the thread of the first object is busy executing its own run() method.

- Now we have three thread of controls. Main thread, first object thread, second object thread and shortly four threads, the third object thread included. Threads are single minded in their purpose always executing the next statement in the sequence. Each thread executes its code independently of the other threads in the program. If the threads choose to cooperate with each other, there are a number of mechanisms, but it is optional.

- Each thread is so separate, that local variables in the methods that the thread is executing are separate for different threads. These local variables are completely private, there is no way for one thread to access the local variables of another thread.

- On the other hand objects and their instance variables can be shared between various threads. The big exception to this analogy is static variables. They are automatically shared between all the threads.

- Meanwhile the three child threads are in a queue waiting for their turn to execute their run() methods. The first thread executes the run() method first, but is made to sleep (1500) milliseconds. This idle time of CPU is not wasted and the control is passed to the second thread in the queue and then it is also made to sleep (1500) milliseconds. This idle time of CPU is now utilized by the next thread in the queue, i.e. the third thread and the process repeats itself.

Runnable interface

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

Code Listing 4.2

```
class threadapp implements Runnable
{
   String name_of_thread;

   threadapp(String s)
   {
      name_of_thread = s;
   }

   public void run()
   {
      try
      {
         for(int i = 0; i < 3; i++)
         {
            System.out.println(name_of_thread + ": " + i);
            Thread.sleep(1500);
         }
      }
      catch(InterruptedException ie) {}
   }
}

class Demo
{
   public static void main(String[]args)
   {
      threadapp  arr[] = new threadapp[3];

      for(int i = 0; i < 3; i++)
      {
         arr[i] = new threadapp("Thread " + i);
         Thread t1 = new Thread(arr[i]);
         t1.start();
      }
   }
}
```

Thread States

A thread can be only in one of the following states:

- New - This is the state the thread is in after the Thread instance has been instantiated, but the start() method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered not alive.

- Runnable - This is the state a thread is in when it’s eligible to run, but the scheduler has not selected it to be the running thread. A thread first enters the runnable state when the start() method is invoked, but a thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state. When the thread is in the runnable state, it is considered alive.

- Running - This is where the action is. This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process. A thread can transition out of a running state for several reasons, including because “the thread scheduler felt like it.” We’ll look at those other reasons shortly. There are several ways to get to the runnable state, but only one way to get to the running state: the scheduler chooses a thread from the runnable pool.

- Waiting/blocked/sleeping - OK, so this is really three states combined into one, but they all have one thing in common: the thread is still alive, but is currently not eligible to run. In other words, it is not runnable, but it might return to a runnable state later if a particular event occurs. A thread may be blocked waiting for a resource (like I/O or an object’s lock), in which case the event that sends it back to runnable is the availability of the resource - for example, if data comes in through the input stream the thread code is reading from, or if the object’s lock suddenly becomes available. A thread may be sleeping because the thread’s run code tells it to sleep for some period of time, in which case the event that sends it back to runnable is that it wakes up because its sleep time has expired. Or the thread may be waiting, because the thread’s run code causes it to wait, in which case the event that sends it back to runnable is that another thread sends a notification that it may no longer be necessary for the thread to wait. Note also that a thread in a blocked state is still considered to be alive.

- Dead - A thread is considered dead when its run() method completes. It may still be a viable Thread object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life. If you invoke start() on a dead Thread instance, you’ll get a runtime (not compiler) exception.

Thread Priorities and Sleep, Yield, and Join

To understand yield(), you must understand the concept of thread priorities. Threads always run with some priority, represented usually as a number between 1 and 10. The scheduler in most JVMs uses preemptive, priority-based scheduling. This does not mean that all JVMs use time slicing. The JVM specification does not require a VM to implement a time-slicing scheduler, where each thread is allocated a fair amount of time and then sent back to runnable to give another thread a chance. Although many JVMs do use time slicing, another may use a scheduler that lets one thread stay running until the thread completes its run() method.

The setPriority() method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed. If not explicitly set, a thread’s priority will be the same priority as the thread that created this thread (in other words, the thread executing the code that creates the new thread).

In most JVMs, however, the scheduler does use thread priorities in one important way: If a thread enters the runnable state, and it has a higher priority than any of the threads in the pool and higher than the currently running thread, the lower-priority running thread usually will be bumped back to runnable and the highest-priority thread will be chosen to run. In other words, at any given time the currently running thread usually will not have a priority that is lower than any of the threads in the pool. The running thread will be of equal or greater priority than the highest priority threads in the pool. This is as close to a guarantee about scheduling as you’ll get from the JVM specification, so you must never rely on thread priorities to guarantee correct behavior of your program.

Note Don't rely on thread priorities when designing your multithreaded application. Because thread-scheduling priority behavior is not guaranteed, use thread priorities as a way to improve the efficiency of your program, but just be sure your program doesn't depend on that behavior for correctness.

What is also not guaranteed is the behavior when threads in the pool are of equal priority, or when the currently running thread has the same priority as threads in the pool. All priorities being equal, a JVM implementation of the scheduler is free to do just about anything it likes. That means a scheduler might do one of the following (among other things):

- Pick a thread to run, and keep it there until it blocks or completes its run() method.
- Time slice the threads in the pool to give everyone an equal opportunity to run.

Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep. A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running. The sleep() method is a static method that sleeps the currently executing thread. One thread cannot tell another thread to sleep.

The yield() method may cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out it will be different thread selected to run. A thread might yield and then immediately reenter the running state.

The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will preempt the running low-priority thread and put the high-priority thread in.

When one thread calls the join() method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the join() method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Code Listing 4.3

```
class threadapp extends Thread
{
   public int tick = 1;
   String name_of_thread;

   threadapp(String s)
   {
      name_of_thread = s;
   }

   public void run()
   {
      while(tick < 5000000) {
         tick++;
         if((tick % 1000000) == 0)
            System.out.println(name_of_thread + ": " + tick);
      }
   }
}

class Priority_Join
{
   public static void main(String[]args)
   {
      threadapp  arr[] = new threadapp[3];

      for(int i = 0; i < 3; i++)
      {
         arr[i] = new threadapp("Thread "+i);

         if(i == 0)
            arr[i].setPriority(Thread.MAX_PRIORITY);
         else if(i == 1)
            arr[i].setPriority(Thread.MIN_PRIORITY);

         arr[i].start();

         try {
            // arr[i].join();
            }
            catch(Exception e)
            {
                System.out.println("Join exception is " +
                    e.getMessage());
            }
        }
    }
}
```

Note Test the above code with join uncommented. Also test with priorities commented.

Synchronization

Can you imagine the havoc that can occur when two different threads have access to a single instance of a class, and both threads invoke methods on that object, and those methods modify the state of the object? In other words, what might happen if two different threads call, say, a setter method on a single object? A scenario like that might corrupt an object's state (by changing its instance variable values in an inconsistent way).

How does synchronization work? With locks. Every object in Java has a built-in lock that only comes into play when the object has synchronized method code. Since there is only one lock per object, if one thread has picked up the lock, no other thread can enter the synchronized code (which means any synchronized method of that object) until the lock has been released. Typically, releasing a lock means the thread holding the lock (in other words, the thread currently in the synchronized method) exits the synchronized method. At that point, the lock is free until some other thread enters a synchronized method on that object. Important points related to locking and synchronization:

- Only methods can be synchronized, not variables.
- Each object has just one lock.
- Not all methods in a class must be synchronized. A class can have both synchronized and nonsynchronized methods.
- If two methods are synchronized in a class, only one thread can be accessing one of the two methods. In other words, once a thread acquires the lock on an object, no other thread can enter any of the synchronized methods in that class (for that object).
- If a class has both synchronized and nonsynchronized methods, multiple threads can still access the nonsynchronized methods of the class! If you have methods that don't access the data you're trying to protect, then you don't need to mark them as synchronized. Synchronization is a performance hit, so you don't want to use it without a good reason.
- If a thread goes to sleep, it takes its locks with it.
- A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires a lock and then attempts to call a synchronized method on that same object, no problem. The JVM knows that this thread already has the lock for this object, so the thread is free to call other synchronized methods on the same object, using the lock the thread already has.
- You can synchronize a block of code rather than a method. Because synchronization does hurt concurrency, you don't want to synchronize any more code than is necessary to protect your data. So if the scope of a method is more than needed, you can reduce the scope of the synchronized part to something less than a full method - to just a block.

Code Listing 4.4

```
class myThread extends Thread
{
   double myamount;
   int op_type;
   static bankapp bank = new bankapp();
   myThread(double amt, int j)
   {
      myamount = amt;
      op_type = j;
   }

   public void run()
   {
      if(op_type == 1)
        bank.deposit(myamount);
      else if(op_type == 2)
         bank.withdraw(myamount);
   }
}

class bankapp
{
   double amount = 5000;
   double getBalance()
   {
      return amount;
   }

   void setBalance(double a)
   {
      amount = a;
   }

   /* synchronized */
   void deposit(double a)
   {
      String str = Thread.currentThread().getName();
      System.out.println(str + " In Deposit");
      try
      {
         double amount=getBalance();
         System.out.println(str + " Got the initial balance as " +
            amount);
         amount += a;
         Thread.sleep(2000);
         System.out.println(str + " Deposit Amount is " + a);
         setBalance(amount);
         System.out.println(str + " Balance " + getBalance());
         show();
      }
      catch(Exception e) {}
   }

   /* synchronized */
   void show()
```

```
    {
       String str = Thread.currentThread().getName();
       System.out.println(str+" In Show  ");
       double amount = getBalance();
       System.out.println(str+" total amount is: " + getBalance());
    }

    /* synchronized */
    void withdraw(double a)
    {
       String str=Thread.currentThread().getName();
       System.out.println(str+" In Withdraw");
       try
       {
          double amount = getBalance();
          System.out.println(str + " Got the balance as " + amount);
          System.out.println(str + " Withdraw amount is "+ a);
          double diff = amount - a;
          if(diff > amount)
             System.out.println("Your balance is less");
          else
             amount -= a;
          System.out.println(str + " Current balance is "+ amount);
          setBalance(amount);
          show();
       }
       catch(Exception e)
       {
          e.printStackTrace();
       }
    }

    public static void main(String[] args)
    {
       myThread u1 = new myThread(1000, 1);
       myThread u2 = new myThread(2000, 2);
       u1.start();
       u2.start();
    }
}
```

Interthread Communication

In the preceding example, we unconditionally blocked other threads from asynchronous access to certain methods (by making use of synchronized). It is powerful, but there is more subtle level of control you can achieve through interprocess communication.

The problem discussed here represents the classic "queuing" problem, wherein one thread produces the data (num in our case) while the other thread consumes the data (calculates root of it in our case).

Java provides us with an elegant mechanism to achieve interprocess communication via the wait(), notify() and notifyAll() methods. These methods are implemented as final methods in Object class, so all classes have them. All the three methods can only be called from within a synchronized method.

wait() method lets a thread say, “there’s nothing for me to do here, so put me in your waiting pool and notify me when something happens that I care about.” Basically, a wait() call means “wait me in your pool,” or “add me to your waiting list.

notify() method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.

notifyAll() method works in the same way as notify(), only it sends the signal to all of the threads waiting on the object. The highest priority thread will run first.

Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.

Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will never be released.

Code Listing 4.5

```
class Calculate
{
   double num;
   boolean check = false;

   synchronized void number(double num)
   {
      try {
         if(check == true)
            wait();
      }
      catch(InterruptedException ie) {}
      this.num = num;
      System.out.println("Number is " + num);
      check = true;
      notify();
   }

   synchronized void root()
   {
      try {
         if(check == false)
            wait();
      }
      catch(InterruptedException ie) {}
      System.out.println("Root is " + Math.sqrt(num));
      check = false;
      notify();
   }
}

class Produce implements Runnable
{
   Thread t;
   Calculate calc;

   Produce(Calculate calc)
   {
      this.calc = calc;
      t = new Thread(this);
      t.start();
   }
   public void run()
   {
      try {
         for(int i = 0; i < 6; i++)
         {
            calc.number((double)i);
            Thread.sleep(300);
         }
      }
      catch(InterruptedException ie) {}
   }
}

class Consume implements Runnable
{
   Thread t;
   Calculate calc;

   Consume(Calculate calc)
   {
      this.calc = calc;
      t = new Thread(this);
      t.start();
   }

   public void run()
   {
      try {
         for(int i = 0; i < 6; i++)
         {
            calc.root();
            Thread.sleep(50);
         }
      }
      catch(InterruptedException ie) {}
   }
}

class Demo
{
   public static void main(String[]args)
   {
      Calculate calc = new Calculate();

      Produce p = new Produce(calc);
      Consume c = new Consume(calc);
   }
}
```

- Upon starting of both the threads, root() thread goes to sleep since the value of check is false. It is forced to wait until data is produced.

- The number() thread completes its task and assigns the value of true to check and notifies the root() thread so that the root() thread can wake up and acquire the mutex. Since the value of check is true, the number() method goes to sleep in the subsequent invocation, until root() consumes the data and notifies it.

- Meanwhile root() method completes its task and assigns a value of false to check and notifies the number() thread so that the number() thread can wake up and acquire the mutex. Since the value of check is false, the root() thread goes to sleep in the subsequent invocation, until the number() method produces data and notifies it.

Exercise

1. What are the different ways of building a multithreaded application and which is the preferred way and why?

2. Explain the different thread states?

3. What is join and thread priorities? Justify with an example.

4. Explain the practical significance of synchronization?

5. How is interthread communication addressed?