THREADS

22UCSC401 – Object Oriented Programming

4th Semester, B Division, Academic Year: 2023 – 24

Department of Computer Science and Engineering, SDMCET, Dharwad-2

THREADS

- Java provides support for Multithreaded programming
- A multithreaded program contains two or more parts that can run **concurrently**
- Each part of such program is called as **thread**
- Each thread defines a **separate path of execution**
- Multithreading is a special form of multitasking

THREADS

- Two types of mutli-tasking:
  - Process-based
  - Thread-based
- Process-based multi-tasking allows the computer to run two or more programs concurrently
- Thread-based multi-tasking allows computer to run two or more threads concurrently
- In case of thread-based multi-tasking, **thread is the smallest unit of dispatch-able code**

THREADS

Threads	Processes
Light-weight	Heavy-weight
Less overhead	High overhead
Inter-thread communication is inexpensive	Inter-process communication is expensive
Context switching is inexpensive	Context switching is expensive
Share the same address space	Have separate address space

JAVA THREAD MODEL

- Single-threaded systems use an approach called an **event loop** with **polling**
- In such systems, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running
- In multi-threading systems, event loop and polling is **eliminated**

JAVA THREAD MODEL

- One thread can pause without stopping other parts of the program
- When a thread blocks in a Java program, only the single thread that is blocked pauses
- All other threads continue to run

THREAD STATES

- **Running** (a thread can be running)
- **Ready** (a thread can be ready to run as soon as it gets CPU time)
- **Suspend** (a thread can be suspended temporarily from its activity)
- **Resume** (only a suspended thread can be resumed)
- **Block** (a thread can be blocked when waiting for a resource)

THREAD PRIORITIES

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others
- Thread priorities are integers that specify the relative priority of one thread over another
- A thread's priority is used to decide when to switch from one running thread to the next
- This is called **context switch**

THREAD PRIORITIES

- Rules that determine when a context switch takes place:
  - A thread can **voluntarily relinquish** control
    - This is done by explicitly **yielding, sleeping, or blocking on pending I/O**
    - In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU
  - A thread can be **preempted** by a higher-priority thread
    - A lower-priority thread that does not yield the processor is simply **preempted—no matter what it is doing**—by a higher-priority thread
    - Basically, as soon as a higher-priority thread wants to run, it does
    - This is called **preemptive multitasking**

THREAD PRIORITIES

- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated
- For Windows, threads of equal priority are time-sliced automatically in round-robin fashion
- For other OSes', threads of equal priority must voluntarily yield control to their peers
- If they don't, the other threads will not run

SYNCHRONIZATION

- Multithreading introduces an **asynchronous behavior** to the programs
- So there must be a way to enforce synchronicity when we want
- This is taken care in Java by the use of "**monitors**"
- Monitor mechanism was defined by C.A.R.Hoare

SYNCHRONIZATION

- We can think of monitors 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
- Monitors can be used to protect a shared data from being manipulated by multiple threads at a time

SYNCHRONIZATION

- In Java, there is no class called "**monitor**"
- Instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called
- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

THREAD CLASS & RUNNABLE INTERFACE

- Java's multithreading system is built upon the *Thread* class, its methods, and its companion interface, *Runnable*
- To create a new thread, Java program will either extend *Thread* or implement the *Runnable* interface

METHODS OF THREAD CLASS

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

THE MAIN THREAD

- When a Java program starts up, one thread begins running immediately
- This is usually called the **main thread**
- The main thread is important for two reasons:
  - It is the thread from which other “child” threads will be spawned
  - Often, it must be the last thread to finish execution because it performs various shutdown actions

THE MAIN THREAD

- Although the main thread is created automatically when the program is started, it can be controlled through a *Thread* object
- To do this, we must first obtain a reference to it by calling the method *currentThread()*, which is a *public static* member of *Thread*

THE MAIN THREAD

- General form of *currentThread()*:
  - *static Thread currentThread()*
- This method returns a reference to the thread in which it is called
- Once we have a reference to the main thread, we can control it just like any other thread

THE MAIN THREAD

- The method *sleep()* causes the thread from which it is called to suspend its execution for the specified period of milliseconds
- General form:
  - *static void sleep(long milliseconds) throws InterruptedException*
  - *static void sleep(long milliseconds, int nanoseconds) throws InterruptedException*

THE MAIN THREAD

- General form of *setName()* method:
  - *final void setName(String threadName)*
- General form of *getName()* method:
  - *final String getName()*

CREATING A THREAD

- There are two ways in Java to create a thread:
  - Implement ***Runnable*** interface
  - Extend ***Thread*** class

CREATING A THREAD BY IMPLEMENTING RUNNABLE INTERFACE

- Create a class that implements *Runnable* interface
- To implement *Runnable* interface, a class need only to implement a single method called *run()*
  - *public void run()*
- In *run()*, we need to write the code that constitutes the new thread

CREATING A THREAD BY IMPLEMENTING RUNNABLE INTERFACE

- *run()* can call other methods, use other classes, and declare variables
- *run()* establishes an entry point for the thread
- When *run()* returns, the thread ends its execution

CREATING A THREAD BY IMPLEMENTING RUNNABLE INTERFACE

- After a class implements *Runnable* interface, it also has to instantiate an object of type *Thread* from within that class
- **Thread** has several constructors
- The one that is used here is:
  - *Thread(Runnable threadObj, String threadName)*
- *threadObj* is an instance of a class that implements *Runnable* interface
- Name of the new thread is specified by *threadName*

CREATING A THREAD BY IMPLEMENTING RUNNABLE INTERFACE

- After the new thread is created, we have to start its execution by calling *start()* method, which is declared within *Thread*
- *start()* executes a call to *run()*
- General form of *start()*:
  - *void start()*

CREATING A THREAD BY INHERITING THREAD CLASS

- Create a new class and make it to extend *Thread*
- Then create an instance of that class
- The extending class must override the *run()* method, which acts as the entry point for the new thread
- The extended class must also call *start()* to begin execution of new thread

CREATING A THREAD BY INHERITING THREAD CLASS

- Child thread was created by instantiating an object of *NewThread* which is extended from *Thread*
- Call to *super()* in the program invokes the following form of *Thread* constructor:
  - *public Thread(String threadName)*
- *threadName* specifies name of the thread

CHOOSING AN APPROACH

- Which approach to use for creating threads in Java??

USING ISALIVE() AND JOIN()

- How can one thread know when another thread has ended?
  - `final boolean isAlive()`
- The *`isAlive()`* method returns true if the thread upon which it is called is still running
- It returns *`false`* otherwise

USING ISALIVE() AND JOIN()

- If we want to wait for a thread to finish its execution, use *join()* method:
  - *final void join() throws InterruptedException*
- Waits until the thread on which it is called terminates

THREAD PRIORITIES

- To set a thread's priority, use the *setPriority()* method, which is a member of *Thread*
  - *final void setPriority(int level)*
- The value of *level* must be within the range *MIN_PRIORITY* and *MAX_PRIORITY*
- *MIN_PRIORITY=1 & MAX_PRIORITY=10*
- *NORM_PRIORITY=5* (default priority)

THREAD PRIORITIES

- To obtain the current priority, use the *getPriority()* method of *Thread*
  - *final int getPriority()*

SYNCHRONIZATION

- A Monitor is an object that is used as mutually exclusive lock, or **mutex**
- Only one thread can own monitor at a time
- When a thread acquires a lock, it is said to have entered the monitor
- All the other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor

SYNCHRONIZATION

- Concept of synchronization is implemented in Java by two ways:
  - Using synchronized methods
  - Using synchronized statement
- Both involve the use of *synchronized* keyword

USING SYNCHRONIZED METHODS

- In Java, all objects have their own implicit monitor associated with them
- To enter an object's monitor, just call a method that has been modified with the *synchronized* keyword
- When a thread is inside a synchronized method, all other threads that try to call on the same instance have to wait
- To exit the monitor, the owner of the monitor simply returns the synchronized method

```
class Display {
  void output(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("]");
  }
}
```

```
class MyThreads implements Runnable {
   private String msg;
   Thread t;
   Display d;
   MyThreads(Display d, String msg) {
      this.d = d;
      this.msg = msg;
      t = new Thread(this);
      t.start();
   }
   public void run() {
      d.output(msg);
   }
}
```

```
public class SynchronizationDemo {
   public static void main(String[] args) {
      Display d = new Display();
      MyThreads mt1 = new MyThreads(d, "SDM");
      MyThreads mt2 = new MyThreads(d, "CET");
      MyThreads mt3 = new MyThreads(d, "CSE");
      try {
         mt1.t.join();
         mt2.t.join();
         mt3.t.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

USING SYNCHRONIZED STATEMENT

- General form of synchronized statement:

```
synchronized(object) {
  // statements to be synchronized
}
```

INTER-THREAD COMMUNICATION

- To implement ITC, Java uses *wait()*, *notify()* & *notifyAll()* methods and these are implemented as *final* methods in *Object*
- All these methods can only be called from within a *synchronized* context

INTER-THREAD COMMUNICATION

- *wait()* tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls *notify()*.
- *notify()* wakes up a thread that called *wait()* on the same object.
- *notifyAll()* wakes up all the threads that called *wait()* on the same object. One of the threads will be granted access.

INTER-THREAD COMMUNICATION

- These three methods are declared within *Object*:
- *final void wait() throws InterruptedException*
- *final void notify()*
- *final void notifyAll()*

PRODUCER-CONSUMER PROBLEM

- Contains 4 classes:
  - Q, the queue that needs to be synchronized
  - Producer, the threaded object that is producing queue entries
  - Consumer, the threaded object that is consuming queue entries
  - PC, the tiny class that creates a single Q, Producer and Consumer

```
class Q {
  int n;
  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }
  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}
```

```
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }
  public void run() {
    int i = 0;
    while (true) {
      q.put(i++);
    }
  }
}
```

```
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }
  public void run() {
    while (true) {
      q.get();
    }
  }
}
```

```
class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Press Control-C to stop.");
  }
}
```

PRODUCER-CONSUMER PROBLEM

- Correct Solution

```
class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    while (!valueSet)
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
  }
```

```
  synchronized void put(int n) {
    while (valueSet)
    try {
      wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
  }
}
```

```
class Producer implements Runnable {
   Q q;
   Producer(Q q) {
      this.q = q;
      new Thread(this, "Producer").start();
   }
   public void run() {
      int i = 0;
      while (true) {
         q.put(i++);
      }
   }
}
```

```
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }
  public void run() {
    while (true) {
      q.get();
    }
  }
}
```

```
class PCCorrectDemo {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Press Control-C to stop.");
  }
}
```

THANK YOU