Java Programming (TCS408)

Module 3

Multithreading

Setting Priority for Threads

- Threads can run in any order. We can set priority for threads execution order.
- Hight Priority thread will execute before low priority thread.

MAX_ PRIORITY : Maximum priority of a thread is 10.

NORM_PRIORITY: Default priority is 5.

MIN_PRIORITY: Minimum priority of thread is 1.

public final void setPriority(int a)

```
public class Main extends Thread
{
   public void run()
   {
      System.out.println("Priority of thread is: "+
Thread.currentThread().getPriority());
   }
   public static void main(String args[])
   {
      // creating one thread
     Main t1=new Main();
      // print the maximum priority of this thread

        t1.setPriority(Thread.MAX_PRIORITY);
      // call the run() method

      t1.start();
   }
}
```

O/P: Priority of Thread is 10

Blocking and Resuming a Thread

sleep( ): to pause a thread for a fixed period of time.

suspend( ): to suspend a thread until resume() method is invoked.

wait( ): to pause a thread until certain condition access. To do resume we have to use notify() method.

stop( ): used to break the thread permanently.

yield(): when we need to run same priority thread.

Problems associated with threads

- Multithreading enables us to better utilize the system's resources, but we need to take special care while reading and writing data shared by multiple threads.

- Two types of problems arise when multiple threads try to read and write shared data concurrently .

1.Thread interference errors

2.Memory consistency errors

Thread interference errors (Race Conditions)

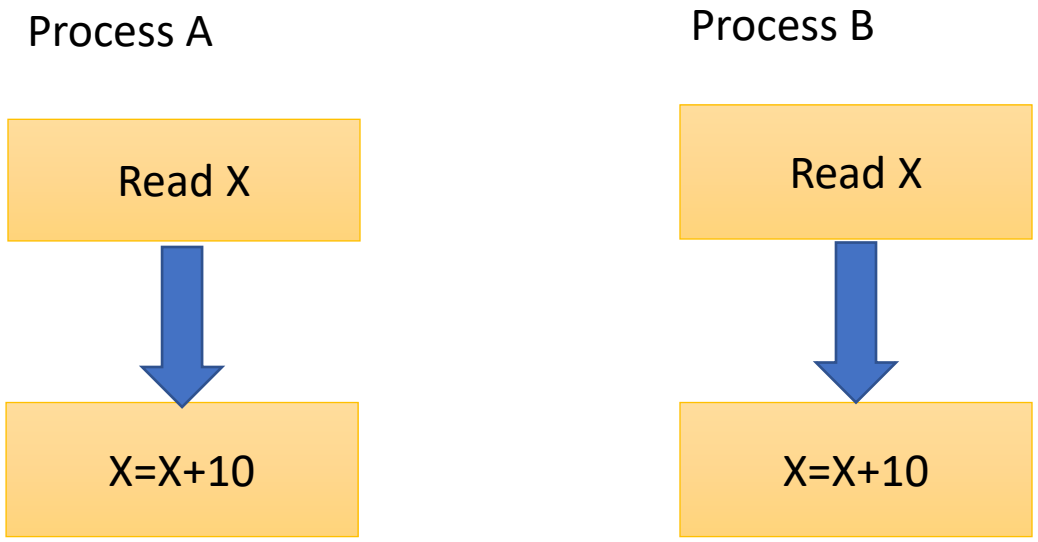
- Output depends on the sequence of events. When events do not occur in the sequence as developer want, it is called race condition.

- Race condition in Java occurs in a multi-threaded environment when more than one thread try to access a shared resource (modify, write) at the same time.

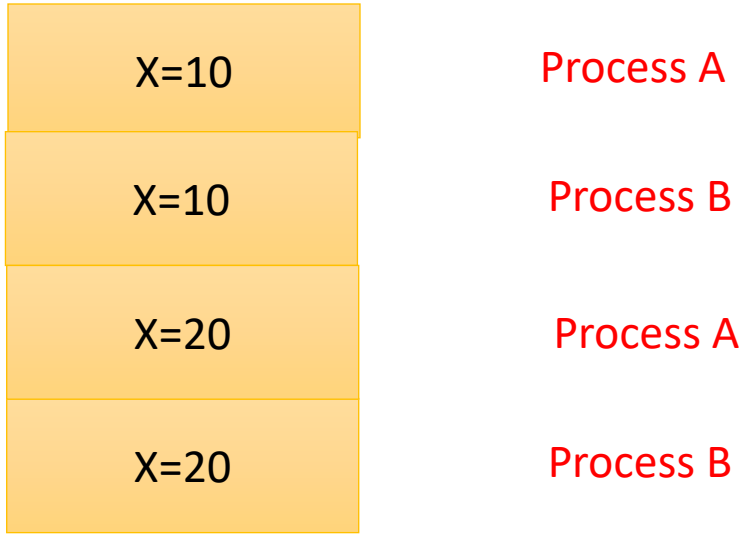
- It is safe if multiple threads are trying to read a shared resource as long as they are not trying to change it.

- Multiple threads executing inside a method is not a problem in itself, problem arises when these threads try to access the same resource(class variables, DB record in a table, writing in a file).

Race Conditions



Process A and B are sharing X at the same time. In this case, output will not be correct. For ex:
X=10, after executing A , value of X should be 10+10=20.
Process B will read X as 20 and after increment the value of X should be 20+10=30.



Process A

Process B

Process A

Process B

But, in this case both process are executing at the same time. So, if Process A is executing so it will read X as 10 but if process B starts at this time so it will also read X as 10. Now again process A execute and read X as 10 and increment it by 10. so now X=20, Now process B executes and read X as 10 and again increment it by 10. So, again X=20, which is wrong.

This is example of race condition. It happens because of thread interference or thread interleaving.

```
public class Main
{
    int x=10;
    public void increment()
   {
      x=x+10;
   }
public static void main (String[] args)
{
    Main obj=new Main();
  Thread t1=new Thread()
  {
public void run()
    {
      obj.increment();
        }};
Thread t2=new Thread()
   {

public void run()
    {
      obj.increment();


    }
  };

   t1.start();
  t2.start();

  System.out.println(obj.x);
}}
```

O/P: Inconsistent results

- **How to avoid thread interference error**

Thread interference can be avoided by making the code thread-safe through:

Synchronization

Memory Inconsistency Error

- In multithreading, there can be possibilities that the changes made by one thread might not be visible to the other threads and they all have inconsistent views of the same shared data. This is known as memory consistency error.
- Memory consistency is more of an architecture-based concept than Java-based.

- Accesses to main memory might not occur in the same order in which the CPU initiated them, especially for write operations which often go through hardware write buffers so that the CPU need not wait for them. CPUs guarantee that the order of writes to a single memory location is maintained from the perspective of all CPUs, even if CPUs perceive the write time of other CPUs differently than the actual time. This sometimes leads to memory inconsistency due to lack of visibility of the correct data.

Memory Inconsistency Error

- Memory inconsistency errors occur when different threads have inconsistent views of the same data.

- Changes done by one thread is not visible to another thread.

Balance=5000

Transaction A	Transaction B
Read Balance	
Balance=Balance-1000	Read Balance
Update balance	Balance=Balance-1000
	Update balance

In this case updates done by thread A in balance is not visible to thread B, So updated balance is 4000, which is wrong. It should be 3000. It is called memory inconsistency error. It happens in case of parallel execution. To avoid this, transaction B should not access balance when transaction A is executing.

- **How to avoid memory consistency errors**

Memory consistency errors can be avoided by establishing a **happens-before relationship**. This relationship guarantees that memory writes operation performed by one thread is visible to a read operation by any other thread on the same shared memory.

Join( )

Thread interference error

Thread interference deals with interleaving of the execution process of two threads.

Thread interference can be avoided by granting exclusive access to threads, that is only one thread at a time should access the shared memory.

Memory consistency error

Memory inconsistency is about visibility and deals with hardware memory.

Memory consistency errors can be dealt with by establishing happens-before relationship which is simply a guarantee that memory writes by one specific statement are visible to another specific statement.

join()

- The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution.

- When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period.

- However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters.

- **join():** When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

```
public final void join() throws InterruptedException
```

- **join(long mls):** When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds) is over.

```
public final synchronized void join(long mls) throws InterruptedException,
```

join(long mls, int nanos): When the join() method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

public final synchronized void join(**long** mls, **int** nanos) **throws** InterruptedException, where mls is in milliseconds.

```
public class Main
{
      int x=10;
      public void increment()
     {
        x=x+10;
     }
public static void main (String[] args)
{
     Main obj=new Main();
   Thread t1=new Thread()
   {
public void run()
     {
        obj.increment();
            }};
Thread t2=new Thread()
    {

public void run()
     {
        obj.increment();


     }
  };

    t1.start(); t1.join();
  t2.start(); t2.join();

  System.out.println(obj.x);
}}
```

```
public class Main extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e)
{System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[])throws Exception{
Main t1=new Main();
Main t2=new Main();
Main t3=new Main();
 t1.start();
  t1.join();
 t2.start();
 t3.start();
 }
}
```

Thread Synchronization

The synchronization is mainly used to:

1.To prevent thread interference.

2.To prevent consistency problem.

There are two types of thread synchronization:

1.Mutual Exclusive

1. Synchronized method.
2. Synchronized block.
3. Static synchronization.

2.Cooperation (Inter-thread communication in java)

Mutual Exclusive

- Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1.By Using Synchronized Method

2.By Using Synchronized Block

3.By Using Static Synchronization

```
class Table{
void printTable(int n){//method not synchronized
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }

 }
}

class Main{
public static void main(String args[])
{
Table obj = new Table();//only one object
Thread t1=new Thread()
{
public void run(){
obj.printTable(5);
}
};
Thread t2=new Thread()
{
public void run()
{
obj.printTable(100);
}
};


t1.start();
t2.start();
}
}
```

O/P:
inconsistent

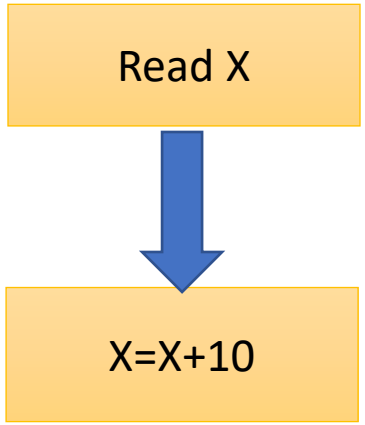
Synchronized Method

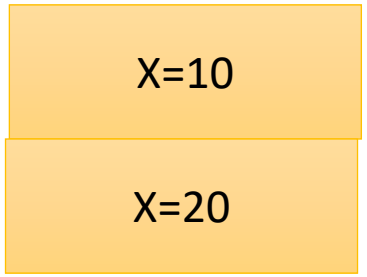
```
class Table{
 synchronized void printTable(int n){//method not synchronized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class Main{
public static void main(String args[])
{
Table obj = new Table();//only one objeclass MyThread1
extends Thread{
Thread t1=new Thread()
{
public void run(){
obj.printTable(5);
}
};
Thread t2=new Thread()
{
public void run()
{
obj.printTable(100);
}
};


t1.start();
t2.start();
}
}
```

O/P: 5
10
15
20
25
100
200
300
400
500

Solution

Read X and X=X+10, should run in batch(one after another). It is called critical section.



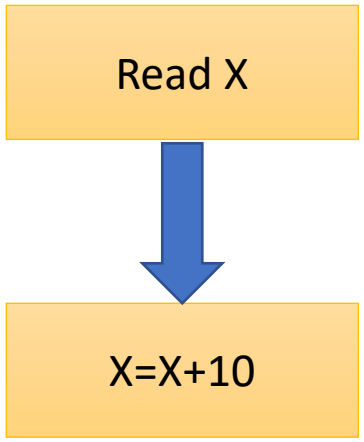


Process A

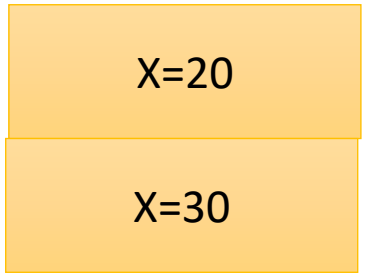
Process A

X=10 is locked in this critical section. No another process can read or modify X. Now next statement of this critical section will execute and x will become 20. Now all statements of critical section has been executed, so, lock on A has been released.

Process B



Now, Process B will read X=20 and it will lock X for its critical section, Now second statement of this critical section will execute and X=20+10=30. After this lock on X will be released.



Process B

Process B

- Thread.join() waits for the thread to completely finish, whereas a synchronized block can be used to prevent two threads from executing the same piece of code at the same time.