JAVA SYNCHRONIZATION & INTER-THREAD COMMUNICATION

Gurmeet Singh
PG Deptt. Of Computer Sci. & I.T.
Hans Raj Mahila Maha Vidyalaya, Jalandhar
gurmeethmv@gmail.com

JAVA SYNCHRONIZATION

Concurrent access to shared data may result in data inconsistency.

To maintain consistency of data, mechanisms are required so that multiple threads execute in an orderly fashion.

So, whenever two or more threads execute on same data, we need synchronization.

Suppose ThreadA and ThreadB tries to update the shared variable data. The final value of data depends upon…

- Which thread was the last one to update??
- The competition among both threads to execute upon data is race condition.
- The result is generally unpredictable.

A number of synchronization problems exists in Operating systems like

1. *Bounded Buffer Problem*
2. *Producer Consumer Problem*
3. *Dining Philosophers Problem.*

HOW TO SYNCHRONIZE CODE ??

Java Provide Method Synchronization and Block Synchronization.

```
synchronize returntype method_name(parameter list)
{
---
}

synchronize(object)
{
---
}
```

CODE WITHOUT SYNCHRONIZATION

```
class Account
{
int balance=1000;
int getbalance()
{
return balance;
}
void deposit()
{
int newbalance=balance+500;
try {Thread.sleep(2000);}catch(InterruptedException e) { }
balance=newbalance;
} }
```

```
class Add extends Thread
{
Account obj;
Add(Account t)
{
obj=t;
}
public void run()
{
obj.deposit();
}
}
```

```
class Test
{
public static void main(String[] args)
{
Account obj=new Account();
Add t1=new Add(obj);
Add t2=new Add(obj);
t1.start(); t2.start();
try { t1.join(); t2.join(); }
catch(InterruptedException e) { }
System.out.println(obj.getbalance());
}
}
```

CHANGE DEPOSIT METHOD AS SYNCHRONIZED

```
synchronized void deposit()
{
int newbalance=balance+500;
try { Thread.sleep(2000);
}catch(InterruptedException e) { }
balance=newbalance;
}
}
```

Execute and check the output in both cases….

INTER-THREAD COMMUNICATION IN JAVA

Sometimes one thread may be dependent on the activities of another thread. In such cases, one thread needs to talk to another thread which is called *Inter-thread communication*. E.g

- Reading from a file or over a network?
- Waiting for a thread to return a result.
- Producer Consumer Problem

WAIT() FOR NOTIFICATION…

- As defined in Object class, every object has a wait(), notify(), and notifyAll() method.
  - These should never be overridden because these are declared as *final*
- These functions can only be called from inside `synchronized` blocks/synchronized methods

WAIT() CONTINUES…

- Whenever a thread enters a *wait* state by executing wait() function, it does nothing until it is *notified* by another thread.
- It also gives up it's lock on the object when wait is called.

```
public synchronized method1() {
  wait();
  … // do something
}
```

NOTIFY() ANOTHER THREAD

- To awaken a thread, a different thread which has a lock on the same object must call the notify() method.
- When notify() is called, the block which had the lock on the object continues
  - **Then** a thread is awakened from its wait() and can grab the lock and continue processing.

NOTIFY() AND NOTIFYALL()

- You never specify which thread should be awoken in notify(). *If there are more than one thread waiting on the same condition, you have no control that this particular thread be awaken*.
- notify() method awakens only one thread.
- notifyAll() method awakens **all** threads waiting for the same condition.

NOTIFYALL() CONTINUES…

- There are two versions - notify() and notifyAll().
- Notify is safe only under 2 conditions:
  - When only 1 thread is waiting, and thus guaranteed to be awakened.
  - When multiple threads are waiting on the same condition, and it doesn't matter which one awakens.
- In general, use notifyAll()

PRODUCER-CONSUMER PROBLEM

Consider a situation where one thread (Producer) produces something and places it on shared place. Other thread (Consumer) consume the same data produced by the producer. Obviously both the producer and consumer should synchronize their execution in such a way so that..

- *Data produced by producer should not be lost*
- *Consumer should not use duplicate data.*

CODE OF PRODUCER-CONSUMER

```
class shareddata
{
int number=-1;
boolean dataavailable=false;
synchronized void setdata(int n)
{
while(dataavailable)
try {       wait(); } catch(InterruptedException e) { }
number=n;
dataavailable=true;
notify();
}
synchronized int getdata()
{
while(!dataavailable)
try {       wait(); } catch(InterruptedException e) { }
dataavailable=false;
notify();
return number;
```

```
class producer extends Thread
{
shareddata s;
producer(shareddata t)
{
s=t;
}
public void run()
{
for(int i=1;i<=10;i++)
{
try {
Thread.sleep(1000); } catch(InterruptedException e) { }
s.setdata(i);
System.out.println("producer: "+i);
}
}
}
```

```
class consumer extends Thread
{
shareddata s;
consumer(shareddata t)
{
s=t;
}
public void run()
{
for(int i=1;i<=10;i++)
{
try {
Thread.sleep(1000); } catch(InterruptedException e) { }
System.out.println("consumer: "+s.getdata());
}
}
}
```

```
class producerconsumer
{
public static void main(String[] args)
{
shareddata s=new shareddata();
producer t1=new producer(s);
consumer t2=new consumer(s);
t1.start(); t2.start();
}
}
```

TICK TOCK EXAMPLE..

This problem simulates the ticking of a clock by displaying the word *"Tick"* and *"Tock"* on the console in a consistent manner i.e. repeated pattern of one tick followed by one tock.

```
class shared
{
String data="Tick";

synchronized void ticklogic()
{
try{
if(data.equals("Tock"))
                wait();
System.out.println("Tick");
data="Tock";
notify();
}
catch(InterruptedException e) {}
}

synchronized void tocklogic()
{
try{
if(data.equals("Tick"))
                wait();
System.out.println("Tock");
data="Tick";
notify();
}
catch(InterruptedException e) {}
} }
```

```
class Tick extends Thread
{
shared tickdata;
Tick()
{}
Tick(shared  temp)
{
tickdata=temp;
}
public void run()
{
while(true)
        tickdata.ticklogic();
}
}
```

```
class Tock extends Thread
{
shared tockdata;
Tock(){}
Tock(shared  temp)
{
tockdata=temp;
}
public void run()
{
while(true)
        tockdata.tocklogic();
}
}
```

```
class TickTock
{
public static void main(String[] args)
{
shared ss=new shared();
Tick t1=new Tick(ss);
Tock t2=new Tock(ss);
t1.start();
t2.start();
}
}
```

Thanx