MULTITHREADED PROGRAMMING-II

Need for Synchronization

- **multiple threads run concurrently and often share common resources** (variables, objects, files, memory).
- **Synchronization is needed to control access to these shared resources** so that threads don't interfere with each other.

Why synchronization is necessary

1. To prevent race conditions

- A *race condition* occurs when two or more threads access and modify shared data at the same time.
- The final result depends on the order of execution, which is unpredictable.

2. To maintain data consistency

- Without synchronization, shared data may enter an inconsistent or invalid state.
- Synchronization ensures that only **one thread updates critical data at a time**.

Need for Synchronization

3. To ensure mutual exclusion

- Critical sections (code that accesses shared resources) must be executed by **only one thread at a time**.
- Synchronization provides *mutual exclusion*.

4. To avoid unpredictable program behavior

- Unsynchronized threads can produce:
  - Incorrect output
  - Partial updates
  - Random errors that are hard to debug

5. To coordinate thread execution

- Sometimes threads must wait for others to complete certain tasks.
- Synchronization mechanisms help in proper coordination (ordering of execution).

Without Synchronization

```
class MyThread extends Thread {
    static int count = 0;

    void increment() {
        count = count + 1;
        System.out.println(Thread.currentThread().getName() + "count is " + count);
    }

    public void run() {
        increment();
    }    }

public class SyncEx {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
   t1.start();
    t2.start();   }   }
```

```
[Running] cd "d:\OOSDL\" && javac SyncEx.java && java SyncEx
Thread-0 count is 1
Thread-1 count is 2
```

```
[Running] cd "d:\OOSDL\" && javac SyncEx.java && java SyncEx
Thread-1 count is 2
Thread-0 count is 2
```

With Synchronization

```
class MyThread extends Thread {
    static int count = 0;

     static synchronized void increment() {
        count = count + 1;
        System.out.println(Thread.currentThread().getName() + " count is " + count);
    }

    public void run() {
        increment();
    }
}

public class SyncEx {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

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

```
   [Running] cd "d:\OOSDL\" && javac SyncEx.java && java SyncEx
   Thread-0 count is 1
   Thread-1 count is 2
```

JOIN--

join() is a **method of the Thread class**. It makes **one thread wait until another thread finishes execution**.

```
class JoinExample extends Thread {
    public void run() {
     for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getNa
me() + " : " + i);
          try {
             Thread.sleep(500);
          } catch (InterruptedException e) {
             System.out.println(e);
          }
       }
   }   }
```

```
public class exampleJoin{
    public static void main(String[] args)     throws
InterruptedException {

        JoinExample t1 = new JoinExample();
        JoinExample t2 = new JoinExample();

        t1.setName("Thread-1");
        t2.setName("Thread-2");

        t1.start();

        t1.join();   // main thread waits until t1 finishes

        t2.start();  // starts only after t1 completes
    }
}
```

Producer Consumer

Shared Buffer Class

```
class Buffer {
   int item;
   boolean available = false;

   synchronized void produce(int value) {
      while (available) {
         try {
            wait();   // wait if item already produced
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      item = value;
      available = true;
      System.out.println("Produced: " + item);
      notify();   // notify consumer
   }
```

```
synchronized void consume() {
      while (!available) {
         try {
            wait();   // wait if no item to consume
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println("Consumed: " + item);
      available = false;
      notify();   // notify producer
   }
}
```

Producer and Consumer Thread

```
class Producer extends Thread {
    Buffer buffer;

    Producer(Buffer b) {
        buffer = b;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            buffer.produce(i);
        }
    }
}
```

```
class Consumer extends Thread {
    Buffer buffer;

    Consumer(Buffer b) {
        buffer = b;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            buffer.consume();
        }
    }
}
```

Main Class

```
public class ProducerConsumer {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();

        Producer p = new Producer(buffer);
        Consumer c = new Consumer(buffer);

        p.start();
        c.start();
    }
}
```

synchronized Keyword (Core Mechanism)

- Java uses the **synchronized** keyword to allow **only one thread at a time** to access a critical section.

a) Synchronized Method

```
synchronized void update() {
    // critical section
}
```

- Locks the **object's monitor**
- Other threads must wait until the lock is released

b) Synchronized Block

```
synchronized(lockObject) {
    // critical section
}
```

- Finer control than synchronized methods
- Improves performance by locking only required code

Inter-thread communication is a mechanism that allows threads to **communicate and coordinate with each other** to avoid race conditions and busy waiting.

1. **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( )**.

final void wait( ) throws InterruptedException

2. **notify( )** wakes up the first thread that called wait( ) on the same object**.**

final void notify( )

3. **notifyAll( )** wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

final void notifyAll( )

Deadlock

 A **deadlock** occurs when **two or more threads are blocked forever**, each waiting for a resource held by another thread.

 Classic Deadlock Scenario

- Thread A holds **Resource 1** and waits for **Resource 2**
  Thread B holds **Resource 2** and waits for **Resource 1**

 Necessary Conditions for Deadlock

- All **four must hold simultaneously**:
- **Mutual Exclusion**
  Resource can be held by only one thread at a time.
- **Hold and Wait**
  Thread holds one resource while waiting for another.
- **No Preemption**
  Resource cannot be forcibly taken away.
- **Circular Wait**
  Circular chain of threads waiting for resources.

```java
public class dloktest {
    static final Object lock1 = new Object();
    static final Object lock2 = new Object();

    public static void main(String[] args) {

        Thread t1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding lock1");
                try { Thread.sleep(100); } catch (Exception e) {}

                synchronized (lock2) {
                    System.out.println("Thread 1: Holding lock2");

                }                   }                });

        Thread t2 = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding lock2");
                try { Thread.sleep(100); } catch (Exception e) {}

                synchronized (lock1) {
                    System.out.println("Thread 2: Holding lock1");

                }                   }                });
         [Running] cd "d:\OOSDL\" && javac dloktest.java && java dloktest
        t1.start();  Thread 2: Holding lock2
        t2.start();  Thread 1: Holding lock1
                    }    }
```