**Deadlock in Java**

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Diagram

Description automatically generated

A **simple real-time example of deadlock** is that suppose there are two friends John and Jerry that are drawing a diagram. During drawing, John needs an eraser, so he will use (lock) the eraser.

Meanwhile, Jerry needs ruler, so he will use (lock) the ruler. Currently, eraser and ruler are not occupied by any of the friends. Now John needs a ruler to continue his drawing. But Jerry will not give it because he is using it currently.

Later on, Jerry needs an eraser but John will not give it as his drawing is not completed because of waiting for the ruler. Thus, none of the friends will release stationary objects and both will wait infinitely for each other to release stationary. This situation is called deadlock in Java.

* Deadlock in a program may occur due to the following conditions. They are as follows:

**1. Mutual Exclusion:** There is at least one resource that must be held in a non-sharable mode and hence, can be used only by one thread. If another thread requests for it, it should wait until the resource is available.

**2. Hold and Wait:** This condition occurs when one thread holds a resource and waits for another resource that is held by another thread.

**3. No Preemption:** The resource will be released only after the execution of thread is completed.

**4. Circular Wait:** This condition occurs when each thread is waiting for a resource held by the preceding one and the last thread is waiting for a resource held by first thread.

This is called circular wait deadlock. This is because every thread is waiting for a resource held by next one and the last thread is waiting for a resource held by first.

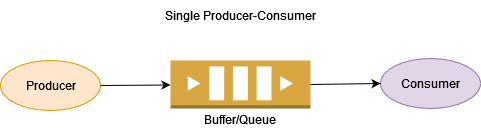
* Deadlocks cannot be completely resolved. But we can **avoid them by following basic rules** mentioned below:

1. **Avoid Nested Locks**: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks**: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join**: A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use **join** with a maximum time that a thread will take.

**Producer-Consumer Problem**

Producer and Consumer are two separate processes. Both processes share a common buffer or queue. The producer continuously produces certain data and pushes it onto the buffer, whereas the consumer consumes those data from the buffer.

Let's review a diagram showing this simple scenario:

[](https://www.baeldung.com/wp-content/uploads/2022/02/Producer-Consumer1.png)

**Problems occur in Producer-Consumer:**

* The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
* Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
* Accessing memory buffer should not be allowed to producer and consumer at the same time.

**Solution**

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

|  |
| --- |
| / Java program to implement solution of producer consumer problem.    **import** java.util.LinkedList;    **public** **class** Threadexample  {  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          // Object of a class that has both produce() and consume() methods  **final** PC pc = **new** PC();            // Create producer thread          Thread t1 = **new** Thread(**new** Runnable())  {              @Override  **public** **void** run()              {  **try** {                       pc.produce();                   }  **catch** (InterruptedException e) { e.printStackTrace(); }              }           } );            // Create consumer thread          Thread t2 = **new** Thread(**new** Runnable()  {              @Override  **public** **void** run()              {  **try** {                       pc.consume();                   }  **catch** (InterruptedException e) { e.printStackTrace();  }              }          } );            // Start both threads          t1.start();          t2.start();            // t1 finishes before t2          t1.join();          t2.join();      }        // This class has a list, producer (adds items to list) and consumer (removes items).  **public** **static** **class** PC  {            // Create a list shared by producer and consumer . Size of list is 2.          LinkedList<Integer> list = **new** LinkedList<>();  **int** capacity = 2;            // Function called by producer thread  **public** **void** produce() **throws** InterruptedException          {  **int** value = 0;  **while** (**true**)  {  **synchronized** (**this**)                  {                      // producer thread waits while list is full  **while** (list.size() == capacity)                          wait();                        System.out.println("Producer produced-"+ value);                        // to insert the jobs in the list                      list.add(value++);                        // notifies the consumer thread that now it can start consuming                      notify();                        // makes the working of program easier to  understand                      Thread.sleep(1000);                  }              }          }            // Function called by consumer thread  **public** **void** consume() **throws** InterruptedException          {  **while** (**true**)  {  **synchronized** (**this**)                  {                      // consumer thread waits while list is empty  **while** (list.size() == 0)                          wait();                        // to retrieve the first job in the list  **int** val = list.removeFirst();                        System.out.println("Consumer consumed-" + val);                        // Wake up producer thread                      notify();                        // and sleep                      Thread.sleep(1000);                  }              }          }      }  } |
|  |
|  |

**Output:**

Producer produced-0

Producer produced-1

Consumer consumed-0

Consumer consumed-1

Producer produced-2