|  |  |
| --- | --- |
| **Visibility issue**  Can be solved by using volatile key word |  |
| **Synchronization issue:**  Even if you use volatile it wont solve the problem in compound operations like i++; |  |
| Can be solved by using Synchronized block |  |
| Another way to solve Synchronization problem using Atomic variable |  |
|  |  |
| Out of Order Execution:  These changes can be done either by **CPU, Compiler or JVM** |  |
| Java Memory model |  |
| Field visibility issue |  |
| Happens before |  |
| **Parallelism**  **(All are independent tasks)**  **Doing lot of things at once..** |  |
| Tools to enable Parallelism |  |
| **Concurrency:**  **Dealing a lot a things at once**  **When there is shared resources** |  |
| Tools to deal with Concurrency problem |  |
| There is **one to one** mapping between java threads and OS threads |  |
| Data Locality issue when there is **context switching** | 1. Cpu has 2 cores 2. Lots of threads 3. T1 running on core and all info of T1 present in local cache of Core 1. 4. Now context switching happened 5. Also T1 info flushed to shared cache and local cache is empty.. 6. Now other thread can run on Core1 which is costly… 7. Again if T1 stated in core1 , then all the operations we have to reverse.. |
| Lock's Condition class in Java |  |
| LockCondition | **public** **class** LockCondition {  **static** Lock *lock* = **new** ReentrantLock();  **static** Condition *condition* = *lock*.newCondition();  **static** **volatile** **int** *i* = 0;  **public** **static** **void** main(String[] args) {  *t1*.start();  *t2*.start();  }  **static** Thread *t1* = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  *lock*.lock();  **try** {  Thread.*sleep*(1000);  System.***out***.println(*i*++);  *condition*.await();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **finally** {  *lock*.unlock();  }  }  }  });  **static** Thread *t2* = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  *lock*.lock();  **try** {  Thread.*sleep*(1000);  System.***out***.println(*i*++);  *condition*.signal();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **finally** {  *lock*.unlock();  }  }  }  });  } |
| Java ReentrantLock - fairness, tryLock and more |  |
| Only one thread access the shared resource at a time (this is called **explicit lock**) |  |
| Only one thread access the shared resource at a time (**this is also called implicit lock**) |  |
| Always advisable to put unlock() inside finally , as when there is exception its always unlock the shared resource. |  |
| Why it is called ReentrantLock | You can call the lock method as many times you wants inside same object.. |
|  | new ReentrantLock(true) (fair lock)  New ReentrantLock(false) (unfair lock) by default |
| ReadWriteLock vs ReentrantLock |  |
|  |  |
| What are Java interrupts?  Cooperative mechanism  Indicating stop signal to the thread |  |
| Runnable can not throw checked exception |  |
| It is better to throw interrupted exception rather return statement to know the **main method** that the current thread is interrupted. |  |
| Producer and Consumer using blocking queue  Very simple |  |
| Producer consumer using condition |  |

|  |  |
| --- | --- |
| Main method |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |