Udemy

multithreading-and-parallel-computing-in-java/learn – wonderful course – contains everything –every fork join pool – by holczer

Java Virtual Threads & Concurrency Masterclass [Hands-On] -by vinoth sel –current – this is having compl future also

modern-java-multithreading-in-java-using-virtual-threads – By pragmatic

Ref links

|  |  |
| --- | --- |
| sir git link | <https://github.com/vinsguru/java-virtual-thread-course> |
|  |  |

Terminologies

1. Platform thread = os thread

IO intensive task – where the processor regularly wait for Input/Output – like reading from disk, writing to a disk, network REST svc calls, interacting with db

Anything which is slower than main memory, where reading is slow like network calls, db calls

Thread is the one who processes the incoming request

Ex;- if our backend service is hit by users, to process those req, tomcat will have server threads those will handle that incoming requests

1. Demon threads are the helper threads- if main threads dies- then app will exit & it wont wait for helper threads to finish their task

Ex:- if any day if hero & heroin and director are not present for the shoot, then even if other persons present they will cancel the shoot

Ex:- garbage collector thread is daemon thread

Concurrency vs parallelism

Concurrency is about managing multiple tasks at the same time just to avoid thread starvation all threads will be executed by same processor using context switching

(means single cpu exec t1 for 5 sec, after that processor will give chance ot t2, again t1, then t2)

Its is like instead of thread starvation, same single processor will follow time slicing algorithm and it will excutes all threads

, while parallelism is about executing multiple tasks simultaneously using multiple processors or cores

| **Aspect** | **Concurrency** | **Parallelism** |
| --- | --- | --- |
| Execution | Interleaved, apparent simultaneity | True, simultaneous  here use all cores or processors simultaneousl |
| Hardware Requirement | Single CPU core sufficient | Multiple cores/processors needed |

|  |  |
| --- | --- |
|  | im image 1 concurrency – same single cpu is executing t1 for 1st 5sec, t2 for 3sec, like context switching  again t1 for 3 sec after that it is giving chance to t2(at any time, only 1 thread is getting executed by processor )  in image 2 – parallelism, 2 cpu are executing separate threads simoutaneously without context switching means truly parallel |

Analogies

1. A thread = a server in a hotel (not chef), chef is the one who prepares dishes he is the cpu

|  |  |
| --- | --- |
| server in a hotel | Thread |
| server in a hotel – bearer/server wont prepare dishes,  he will take order and gives it to chef  and chef is the one who prepares dishes – 1 chef can prepare only 1 dish at a time- lly 1 cpu can execute only 1 thread at a time | thread – thread also wont execute the line of code  thread will just give those lines to cpu and cpu will execute all those loc  cpu will give the chance to a thread to talk with cpu, in that meeting thread passes all the loc to cpu and he will be executing  all these threads will be competing for the cpu |
| so even if u have 100 bearers In a hotel, but if only 2 chefs are present then its waste  all the bearer/waiter will simply wait untill dishes are prepared by chef  so along with bearers to prepare the dishes chefs also should be more | lly , even if 100 threads are there to execute those lines processor must be free right  1 processor/cpu will give chance to only 1 thread at a time  so in real time along with threads processors also must be present |

|  |  |
| --- | --- |
|  | This means cpu can execute only 1 thread line of content at a time  So os scheduler will keep on switching the threads (few mins thread1, then it will give chance to thread 2 , then thread 3 …) this is called context switching  this is like people waiting for doctor appointment  1 java thread = 1 os thread = platform thread (java thread is a wrapper around os thread)  java thread introduced 25 years ago  so here all the threads will be competing for the cpu |

Many customer will come to hotel to eat food – a server /bearer is the one who will come and take his order and processes his food and he will return the food to cust

In this case the server/bearer is called thread , by default tomcat server will have 200 threads means at a time 200 requests can be handled by tomcat server

All the requests will be assigned to the thread to process

1. Executor service is having a pool of threads == it is like a TCS which consists pool of employees who are on bench

So if any work comes those emp who are in bench will take that work and he will do the project

Facts

Creating threads is costly affair , so in real time u don’t create thread, you should always submit a task to thread pool like executor service thread pool

We cant create a thread pool of 1 million threads- because these are os threads

Advantages

1. If Thread dies suddenly we don’t need to worry, another thread from pool will be assigned to work on our submitted task (like emp exits from tcs- so tcs then gives new resource to client, hence product based companies instead of hiring employees, they will take emp/contractors from tcs/service based companies)
2. We don’t need to create thread object and we don’t need to call start method

Important points

1. Why we should use multiple threads? In my ampf laptop that cpu have 22 processors- so with advent of multiple processors in single cpu, if u create only 1 thread

It means u are not utilizing all processor’s effectively, so if there are any independent tasks , run all of them in parallel by submitting those tasks to thread pool

Threads are light weight processes ,

Thread t1=new Thread(new Runnable()) ;

t1.start();; if u call start method, immediately that thread will not get processor, bec at a time 100 of threads will be ready, but that single processor may busy in executing other threads, whenever processor is free or during context switch if the thread get processor then thread will be in running state, till then its in runnable state

|  |  |  |  |
| --- | --- | --- | --- |
| Stack memory | **Every thread has its own stack area**  **During method execution**   * local variables of that method will be stored in stack area(int x=10; Employee emp=new Employee()), here employee/**any obj will be created in heap**, but that obj address is stored in variable called emp, that emp variable will be present in stack aread * method arguments, method calls(controller->service🡪repo all these calls) are stored in stack area, size=1MB  |  |  | | --- | --- | |  | **once the thread finishes its execution,**  threads stack area will be destroyed, means all the variables which have those object references will be destroyed , means there is no variable which holds those object references,  then objects will become unreachable from current stack  if nobody is storing those heap object addresses, then objects will be garbage collected  that stack memory 1MB is allocated in RAM memory | |
| Heap memory | only objects are stored in heap area (its very large ), All the threads will share same HEAP memory |

1. Thread is just like a bearer in hotel, actual thread method code / all lines will be executed by processor (actual work will be done by chef in kitchen), scheduler is the one who allocates cpu to the thread – so even if 100 bearers are there who takes orders, if only 1 chef is present in kitchen its waste – bec at any time 1 chef can prepare dishes 1 order given by bearer – lly 1 cpu can give chance only to 1 thread at a time,
2. We cant create millions of threads - 1 java platform thread = 1 os thread = so in real time that many os threads will not at all be available, for every thread os has to allocate 1-2MB stack mem, so in real time that much mem also will not be available
3. Problem of Creating Thousands of threads – 1000’s of threads doesn’t give parallalism = bec single cpu is present = threads will be idle and will just be waiting for cpu allocation all the time

So even if u create 1000 threads if single proc is there then only 1 thread will be give chance at a time (rem 999 threads will be ideal)- and cpu should give chance to all 1000 threads threads- un neccesarily cpu needs to do context switch all the time (means for 5sec it will give chance to t1 then switch and execute t2 for 3sec again switch like this no thread will complete its execution all threads will just be partially executed)

So hence create n threads based on the processor- for 4 cpu don’t create 4000 threads its waste bec here 1 cpu has to do context switch b/n 1000 threads

In real time processor is the one which executes all our instructions /each line of code, threads will just bring the lines to the processor

1. Every thread has is own stack area (1 MB)- to store local var ref and to store which method it is currently exec
2. Class and object level lock – every object in java will have a lock called intrinsic lock for syncrhonized methods, threads will get the lock of that obj and it will go inside

Synchronized means- only thread which has the lock of that object cant enter, other cant even enter without the lock

|  |  |
| --- | --- |
| public synchronized void increment(){  ++**x**; } | ex:- same like a person who enters the oyo room, he will get the lock of that room and enter, after that no person can enter the room all other has to wait, once that guy finishes the work in that room, he will gives the lock to reception,  once the lock is relased , other guy will get the lock from reception and he will enter and he will lock the room …  A thread before entering, 1st it has to get the lock of that object  once a thread acquires the lock of that obj, other threads cant even enter the method , others has to wait bec the lock is with previous thread  Once the thread finishes its execution, it will release the lock while leaving, then other thread which will get the lock that will enter that method, |

1. Always prefer synchronized blocks oversynchronized methods to reduce time delay

If u keep full method as synchronized, then all 600 lines in that method will be sync, which causes delay, hence keep only few piece of code in sync blocks

Thumb rule- we should synchronize only blocks that are neccesary not methods (so never make a full method synchronized)

Don’t keep all methods of a class as sync – if u keep independent methods also as sync, then threads has to wait for the lock

Ex:- don’t keep controller methods as sync – if u keep then untill thread 1 finishes its exection, no other user can hit the url

Thmub rule- if a method is not modifying any global var, then remove sync for that method , bec even if multiple threads even also no problem

1. Always prefer sync blocks instead of sync methods (which makes all lines sync) and instead of re-entrant lock (which mandates try catch finally)

|  |  |
| --- | --- |
| Process  An instance of computer program It includes code,resources(memory allocated by os ), & it is heavy weight, It is very expensive to create and destroy a process    entire app is considered as a process (MS word , sumatra pdf apps are called processes)  inside app we have threads | **Thread**    Thread is a part of process (A process contain one or more threads)  above pic says Threads with in a process can share the memory and space |
| processor    cpu == processor both are same  scheduler is the one which will allocates cpu to a thread  cpu is like a bike , at any time only 1 can travel  1 cpu can execute 1 thread at a time  ex:- here even though 3 threads are there, since only 1 cpu is present, at any time only 1 thread will be executed, 1st 5 mins t1 and then t2 & then t3 will be given chance  doubt- if 1 cpu is present then whats use of creating multiple threads as at any time only 1 thread will be given chance | **multi core processor**    if 2 cpu are there, each cpu can exec 1 thread at any point of time |

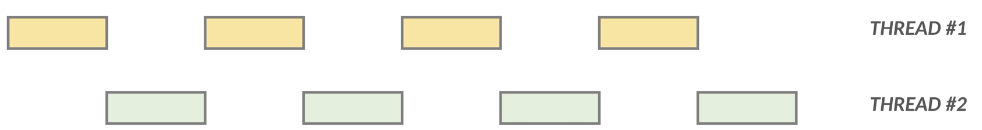
Misc concepts

Threads of same processes share same run in a shared memory space

While processes run in different memory space

Time slicinig algorithm

1. Problem of Creating Thousands of threads – means single processor handles thread 1 for 2 sec and it switch and handles thread 2 for 2sec ….



Here cpu should keep on does the context switching between all the threads

bec single core proce is present = threads will be idle and will just be waiting for cpu allocation all the time, so 1 cpu and 1000 threads- then cpu has to follow time slicing algorithm and cpu has to give opp to all 1000 threads by allocating some time to each thread

So even if u create 1000 threads if single proc is there then only 1 thread will be give chance at a time (rem 999 threads will be ideal)- main problem is if u create 1000’s of threads- then cpu must manage all threads and cpu should give chance to all 1000 threads threads- un neccesarily cpu needs to do context switch all the time

So hence create n threads based on the processor- for 4 cpu don’t create 4000 threads its waste bec here 1 cpu has to do context switch b/n 1000 threads

Whereas modern processors can have multiple cpus

|  |  |
| --- | --- |
| If u have multiple cpu then each cpu will execute some thread all the time  here no need of context switch at all |  |

Advantage of multi threading

Better cpu utilization- If 10 cpus (chefs) are there then create 10 threads , so that all the chefs/cpu will excute some thread/task at all time-work will be completed fast

If 10 cpu are there , but if u give tasks 1 by 1 means at all time 9 cpus will be idle- means u are not utilizing resources properly

Problems with multi threading

1. Data inconsistency issue- if multiple threads(associated with same process) are using same memory area- then modifying same variable then resultant data is inconsistent
2. Its very time consuming to switch between the threads (context switch) – so don’t un neccesarily create 1000’s of threads – then cpu takes more time to switch between threads

|  |  |
| --- | --- |
| this says initially if u create more threads (5 threads for 5 processors )in cpu til that time speed will be high, if u creates even more 50 threads- then it will cause delay, because now cpu has to perform context switching between all threads, now all threads will be executed partially  thumb rule- for small problems and small apps its unneccesary to create multiple threads |  |
|  |  |

Thread life cycle

we can create threads either with **Runnable interfaces** or with **Thread classes**. You may pose the question:

what approach to prefer?

* if we extends Thread then we can’t extend any other class (usually a huge diasvantage) because in Java a given class can extends one class exclusively
* a class may implement more interfaces as well - so implementing the Runnable interface can do no harm in the software logic (now or in the future)

|  |  |
| --- | --- |
| sleep() – thread.sleep() will cause current thread to sleep for some time | runnable means – that thread is waiting for cpu allocation (bec cpu might be busy is executing other threads), when cpu gives the allocation to that thread, then that thread will start executing and then it will be in running state  blocked/ waiting  t1.join()  t2.join()  if main thread executed these lines, it means main thread will simply wait for these 2 threads to finish the execution & that waiting state is called blocked  terminated- when thread completely executed its run method |

Thread methods

|  |  |
| --- | --- |
| Join() | t1.join()  t2.join()  if main thread executed these lines, it means main thread will simply wait for these 2 threads to finish the execution & that waiting state is called blocked  Thread t=new Thread(new Runner1()); Thread t1=new Thread(new Runner2()); t.start(); t1.start();  t1.join(); //now this main thread will wait only for 2nd thread to finish its execution  System.***out***.println("main thread finished"); |

Daemon threads

Daemon threads are called helper threads (GC thread) for main thread-

note if main thread dies, then all helper threads will die automatically

Ex;- if main threads- hero, heroin, director didn’t came to shoot, then what is use of camera men(who is a helper thread ),

Thread t2=new Thread(new Runner3());  
t2.setDaemon(true);

Thread priority

|  |  |
| --- | --- |
|  | by default, the thread priority is 5 (min-1 , max-10 , if we set 11 then we will get illegal argument exception)  generally high priority thread will get the cpu 1st and they will be executed before low prio thread  so these high probut it depends on underlying os to avoid thread starvation  all threads are having same priority then FIFO order |
|  |  |

What is Synchronization and its problems

Synchronization means only 1 at a time, ex;- a person will get the room lock and will goto oyo room and while leaving he will return the lock only then others can use that room

In java also same a thread to enter into a synchronized method it should get the lock of that object , once it got the lock and untill he executes that method other threads has to wait for previous thread execution only after that he will release the lock

|  |  |
| --- | --- |
| Note;- we cant apply synchronized keyword to a variable synchronized int **x**=0;   * there is a single intrinsic lock associated with every object or class in Java * a given  thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, * and then the thread releases the intrinsic lock when it's done with them * with Locks: the acquired lock can be released any thread * RLocks can be released by the thread that acquired it exclusively | so correct way is , either we should apply to a method or use AtomicInteger/Atomic variables  private int x = 0;  public void increment() {  synchronized (this) {  x++;  }  } |

Every object will have 1 lock and every class will have 1 lock in java

|  |  |
| --- | --- |
| Class level locks | Object level lock |
| public static synchronized void increment() {  counter++;  }  public static void increment() {  synchronized(SomeClass.class) {  counter++;  }  }  This is called class level locking because we get the monitor lock (intrinsic lock) associated with the class | public synchronized void increment() {  counter++;  }  public void increment() {  synchronized(this) {  counter++;  }  }   The first method uses a **method-level lock**, automatically locking on the instance of the object (this).   The second method uses a **synchronized block**, manually locking on this, offering more control over what code is synchronized. |
| synchronized (this){  System.*out*.println("in producer method start");  wait();  System.*out*.println("in producer method end"); } | in case of synchronized blocks/ sync methods   1. When threads exits the sync block/ sync method then lock will be auto released 2. Even if an exception occurred inside method, lock will be auto released |

Synchronized keyword demo

|  |  |
| --- | --- |
| public class EmployeeController {  *List*<Employee> **employeeList**;   @WebMethod  public **synchronized** String empDetails(String *empName*) throws InterruptedException {  System.***out***.printf("\n currently executing %s 's request \n ", **Thread.*currentThread*().getName());**  Thread.*sleep*(20000);  System.***out***.printf("\n execution of %s thread if over", Thread.*currentThread*().getName());  return "Best Employee name is santhoshi and " + *empName*;  }  see the log- at a time only 1 thread is inside – once that thread exited then only other thread was able to enter  currently executing http-nio-8080-exec-1 's request  2025-07-14 13:36:31.604 INFO 6592 --- [nio-8080-exec-2] o.a.c.s.E.E.EmployeeController : Inbound Message  execution of http-nio-8080-exec-1 thread if over  currently executing http-nio-8080-exec-2 's request  execution of http-nio-8080-exec-2 thread if over2025-07-14 13:37:10.404 | see here untill 1st thread completes, 2nd will keep on waiting |
|  |  |

Note;- Don’t keep controller method as synchronized- if u keep only 1 thread can execute that controller method, untill he finished other thread has to wait

Always prefer synchronized blocks oversynchronized methods

Advantages of sync –it will solve dirty read problem

1. Problem of dirty reads- when multiple threads are trying to update same variable at a time

int x=10; if 2 threads are modifying the same variable, 1st /another thread may read old value before updating

ex:- if each thread responsibilty is to update the current value by 1, then suppose if both threads read current value as 10 & if they update to 11 then its wrong

problem is before the 1st thread updates the variable, if 2nd threads reads the old value ..then its wrong (means 2nd thread should read only afte 1st thread updation)

|  |  |
| --- | --- |
|  |  |
| public class DirtyReadProblem {  int **x** = 0;   public void increment() {  for (int i = 0; i < 10000; i++) {  ++**x**;  }  }   public synchronized static void main(String[] *args*) throws InterruptedException {  DirtyReadProblem syncDemo = new DirtyReadProblem();  Thread t1 = new Thread(() -> {  syncDemo.increment();  });  Thread t2 = new Thread(() -> {  syncDemo.increment();  });  t1.start();  t2.start();  t1.join();  t2.join();  System.***out***.printf("Final global variable val is %d", syncDemo.**x**);  } }  if u execute this, everytime we will get different output,  bec both the threads are executing same method at a time , t1 maynot get the copy of latest value, and t1 and t2 may update same value, so solution is  public synchronized void increment() {  for (int i = 0; i < 10000; i++) {  ++**x**;  } }  with this, now only 1 thread will enter and exit at a time | here 2 threads t1, t2 are trying to modify same variable x ,  hence there is a problem of dirty reads  to avoid this problem use syncronized keyword  Problem:- so in this case, since 2 threads are modifying same variable,  solution- just use AtomicInteger instead of int or make the method sync or use synchronised block |

|  |  |
| --- | --- |
| Exception inside sync method | exception inside a sync block |
| Yes, even if a thread enters a synchronized method and throws an exception, it releases the lock on that object when it exits the synchronized block — even if the exit was caused by an exception. | Yes — **even if a thread throws an exception inside a synchronized block**, it will **automatically release the lock** it was holding on the synchronized object. |

|  |
| --- |
| **so synchronization means – only 1 thread at a time and sync means lock ex:-** a person locks his oyo room **every object will have only single object lock and 1 class level lock** |
| ex:- if a method is kept synchronized, only 1 thread can execute a method at a time, that thread will get the lock of that object and it will go inside (he wont release the lock untill he finishes the execution )  ex:- a person will enter the oyo and lock the door, as long as a person has the lock, others cant enter the room  How?  ex:- the oyo person of the house will get the lock of that house and he will go inside and he holds the lock  and since he holds the lock, other threads cant get the lock and hence other threads cant go inside, and the threads has to wait untill previous thread exits that method   * As long as the thread owns a lock, no other person can enter the room/method |

Problems of synchronization

**Thmub rule- if a method is not modifying any global var, then remove sync, bec even if multiple threads even also no problem**

1. **Keeping Too many independent synchronized methods in same class causing more delay –** so don’t keep indep methods as synchronized

Ex- if the class contain 2 independent synchronized methods m1(), m2(), if thread t1 is executing m1() method , even t2 cant execute m2() method bec the obj lock is with t1 and now eventhough m2() is independent method t2 has to wait untill t1 finishes his m1() method

, in this case only 1 thread can execute all the methods of that class bec all methods are synchronized

* If all methods are synchronized then app will be slower (same like string buffer)- so in this if single thread entered a method , then all other threads have to wait even to execute other methods

→ if we have 2 independent synchronized methods then the threads have to wait for each other to release the lock

So in string buffer, eventhough most methods are independent since they kept as sync, if 1 thread executing 1 sync mehtod , all the other threads has to wait for prev thread completion to release the intrinsic lock of that obj

|  |  |
| --- | --- |
|  |  |

Don’t keep all methods of a class as sync –

Ex:- don’t keep controller methods as sync

Solution- if there are independent methods if u feel those needs to be synchronized, then don’t keep those methods in single class

**So when to keep a method synchronized**

If any method is updating a global variable/if changing the state of a class, then keep that method synchronized , bec if 2 threads operate on the same method parallely then data inconsistency problem may occur, hence to avoid It only in that case we should keep as synchronized

Wait() notify() inter thread notification

These 2 methods are present in object class ,

|  |  |
| --- | --- |
| synchronized (this){  System.*out*.println("in producer method start");  wait();  System.*out*.println("in producer method end"); } | these 2 methods wait(), notify() will release the lock of current object  wait() will pause the current thread execution & it will go into waiting state by releasing the lock immediately for other high priority thread so that other thread will take the lock of current object  and the current parent thread will go into waiting state forever and it will keep on waiting untill someone notifies it  notify() will notify the parent thread saying my execution completed and he can take back the lock  – but it will not release the lock immediately, it will release only when it exits the sync block  note:- if child thread doesn’t notify that parent thread (who released the lock) then parent thread will keep on waiting forever by thinking child thread didn’t completed , |

|  |  |
| --- | --- |
| note 2- even if child notifies parent ,but if child didn’t completed its execution, parent cant get the lock & parent cant resume,  parent resumes only when child thread exits sync block  here since I didn’t call notify(), parent thread will be in waiting mode forever  and program stucks  ouptut of code  produce method is being executed byThread-0  consume method is being executed byThread-1  in producer method s-1  in consumer method s-1  in consumer method s-2  in consumer method s-3  in consumer method s-4  //consumer over but since it didn’t notified producer , producer thread will be In waiting state for ever  public static void main(String[] *args*) throws InterruptedException {  WatiNotifydemo notifydemo = new WatiNotifydemo();  Thread t1 = new Thread(() -> {  try {  notifydemo.produce();  } catch (InterruptedException *e*) {  throw new RuntimeException(*e*);  }   });  Thread t2 = new Thread(() -> {  try {  notifydemo.consume();  } catch (InterruptedException *e*) {  throw new RuntimeException(*e*);  }   }); t1.start(); t2.start();  } } | in  package org.ampf;  public class WatiNotifydemo {   public void produce() throws InterruptedException {  System.*out*.println("produce method is being executed by" + Thread.*currentThread*().getName());  synchronized (this) {  System.*err*.println("in producer method s-1");  wait();  System.*err*.println("in producer method s-2");  System.*err*.println("in producer method s-3");  System.*err*.println("in producer method s-4");  }  }   public void consume() throws InterruptedException {  System.*out*.println("consume method is being executed by" + Thread.*currentThread*().getName());  Thread.*sleep*(1000);  synchronized (this) {  System.*out*.println("in consumer method s-1");  System.*out*.println("in consumer method s-2"); *// notify();* System.*out*.println("in consumer method s-3");  Thread.*sleep*(8000);  System.*out*.println("in consumer method s-4");   }  } |

With notify

|  |  |
| --- | --- |
| public void produce() throws InterruptedException {  System.*out*.println("produce method is being executed by" + Thread.*currentThread*().getName());  synchronized (this) {  System.*err*.println("in producer method s-1");  wait();  System.*err*.println("in producer method s-2");  System.*err*.println("in producer method s-3");  System.*err*.println("in producer method s-4");  }  }  public void consume() throws InterruptedException {  System.*out*.println("consume method is being executed by" + Thread.*currentThread*().getName());  Thread.*sleep*(1000);  synchronized (this) {  System.*out*.println("in consumer method s-1");  System.*out*.println("in consumer method s-2");  notify();  System.*out*.println("in consumer method s-3");  Thread.*sleep*(8000);  System.*out*.println("in consumer method s-4");   } } | output  produce method is being executed byThread-0  consume method is being executed byThread-1  in producer method s-1  in consumer method s-1  in consumer method s-2  in consumer method s-3  in consumer method s-4  in producer method s-2  in producer method s-3  in producer method s-4  //observe that notify wont release the lock immediately, instead it will just notify  lock will be released only when that child thread came out of sync block |

these can be used only in synchronized blocks /sync methods/ sync area only

 **Avoid if you can use higher-level tools** like:

* BlockingQueue (e.g., ArrayBlockingQueue)
* CountDownLatch
* Semaphore
* CyclicBarrier
* ReentrantLock with Condition.await()/signal()

 They’re safer, easier to debug, and more readable than raw wait()/notify() logic

Wait vs sleep()

* wait (and notify) must happen in a synchronized  block on the monitor object whereas sleep does not
* sleep operation does not release the locks it holds while on the other hand wait releases the lock on the object that wait() is called on
* you call wait on the **Object**while on the other hand you call sleepon the **Thread**itself
* wait can be interrupter (this is why we need the *InterruptedException*) while on the other hand sleep can not

So as you can see there are some differences between wait and sleep operations!

Producer consumer solution

|  |  |
| --- | --- |
| public class ProducerConsumerUsingWait {  int **x** = 0;   public void produce() throws InterruptedException {  System.***out***.println("produce method is being executed by --> " + Thread.*currentThread*().getName());  for (int i = 0; i < 5; i++) {  synchronized (this) { *// System.err.println("executing producer start ");* ++**x**;  System.***err***.println("produced new value");  wait(); *// System.err.println("executing producer start");* }   }  }   public void consume() throws InterruptedException {  System.***out***.println("consume method is being executed by --> " + Thread.*currentThread*().getName());  Thread.*sleep*(2000);  for (int i = 0; i < 5; i++) {   synchronized (this) {  System.***out***.println("consumed value as --> "+**x**);  notify();  }  Thread.*sleep*(1000);   }  } | public static void main(String[] *args*) throws InterruptedException {  ProducerConsumerUsingWait notifydemo = new ProducerConsumerUsingWait();  Thread t1 = new Thread(() -> {  try {  notifydemo.produce();  } catch (InterruptedException *e*) {  throw new RuntimeException(*e*);  }   });  Thread t2 = new Thread(() -> {  try {  notifydemo.consume();  } catch (InterruptedException *e*) {  throw new RuntimeException(*e*);  }   });  t1.start();  t2.start();  } } |

Re- entrant locks

Ok so a thread **cannot acquire a lock owned by another thread**. But a given thread **can acquire a lock that it already owns**. Allowing a thread to acquire the same lock more than once is called *re-entrant synchronization.*And this is exactly what is happening in Python when using RLocks- the same thread may acquire the lock more than once.

**For example**: let's consider recursive method calls. If a given thread calls a recursive and synchronized method several times then it is fine (note that in this case the same thread "enters" the synchronized block several times). There will be no deadlock because of re-entrant synchronization.

Major issue with re-entrant lock:-

|  |  |
| --- | --- |
| synchronized (this){  System.*out*.println("in producer method start");  wait();  System.*out*.println("in producer method end"); } | in case of synchronized blocks/ sync methods   1. When threads exits the block/ method then lock will be auto released 2. Even if an exception occurred inside method, lock will be auto released |

|  |  |
| --- | --- |
| public class ReentrantLock implements *Lock*  public ReentrantLock(boolean *fair*){  }  // if fairness==true means then longest waiting thread will get the lock | *Lock* **lock** =new ReentrantLock(true);  public synchronized void increment() {  **lock**.tryLock();  try {  for (int i = 0; i < 10000; i++) {  ++**x**;  }  } finally {  **lock**.unlock();  } } |

But incase of re-entrant locks untill this line **lock**.unlock();is executed lock will not be released automatically , hence we have to manually write this line in finally block

Sync blocks vs sync methods vs Re-entrant locks which one to use

|  |  |  |
| --- | --- | --- |
| Sync methods | sync blocks | re entrant lock |
| main issue is if we make full method sync, if method is so big of 600 lines if has to wait so use sync block | this is best among all 3,  so keep only those few lines of code which is changing the global variable | main issue here is if the thread didn’t execute this line **lock**.unlock();lock will not be released automatically , hence we have to manually write this line in finally block |

we can make a lock fair: prevent thread starvation  
     Synchronized blocks are unfair by default, bec which any thread can get into the sync block- but in re-entrant lock we have option like longest waiting thread will get the chance to go into sync block

→ we can check whether the given lock is held or not with reentrant locks

→ we can get the list of threads waiting for the given lock with reentrant locks

→ synchronized blocks are nicer: we do not need the try-catch-finally block

Keywords – volatile

Volatile means rapidly changing which is un predictable (weather is volatile ), so if u keep a volatile keyword on a variable this means we are declaring that variable as rapidly changing variable and we are asking cpu to fetch the value directly from RAM instead of from thread cache

|  |  |
| --- | --- |
|  | Every read of a volatile variable will be read from the RAM directly so from the main memory (and not from cache) → usually variables are cached for performance reasons  → caches are faster. Do not use volatile keyword if not necessary ( + it prevents instruction reordering which is a performance boost technique )  use case- if it is a rapidly changing variable via while loop then declare that variable as volatile |

Virtual threads

Wht is the problem with platform threads

|  |  |
| --- | --- |
|  | problem here is when order service is calling 3 other services in microservices communication, the main threads/os threads/platform threads will have to wait for those responses   * Most values os threads were in waiting state – this is d problem * Ex:- in reality clients will not keep their most valuable employees idle |

Executor service

Never create a thread pool with 100’s of threads, create only how many processors u have

Int number =Runtime.getRuntime().availableProcessors()

A class called

|  |  |
| --- | --- |
| Int numberOfProc =Runtime.getRuntime().availableProcessors()  *ExecutorService* es = Executors.*newFixedThreadPool*(numberOfProc); | //worst as some would have blocked for IO  And we cant increase number of threads based on load |
| Executors.newCachedThreadPool() | Ok—threads are created on demand/workload – but this keeps on increases if more tasks came  Best is Threads will be shutted down automatically if they are not used |
| Executors.*newWorkStealingPool*(3); | Here among 3 threads if any thread is ideal, that thread will steal that work from other threads queue |
| *ScheduledExecutorService* scheduledes = Executors.*newScheduledThreadPool*(2); |  |
| ThreadPoolExecutor | Executes each submitted task using a pooled worker thread |
|  |  |
|  |  |
|  |  |

Key interfaces

Executor, ExecutorService, ScheduledExecutorService, CompletionService,

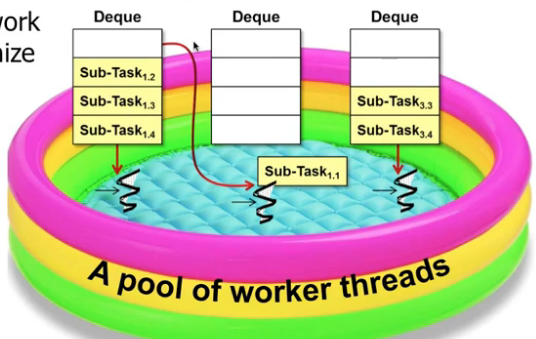
Tasks

These tasks represents the tasks whose instances are executed by another thread

|  |  |  |
| --- | --- | --- |
| Runnable | Callable | Future (I) & impl class is FutureTask(c)  Future (represents result of an async task, like result of callable) |
| Void run() | Object call() | boolean cancel(boolean *mayInterruptIfRunning*);  boolean isCancelled();  boolean isDone();  *V* get() – this is a blocking call & returns the result after some thread executed that task  *V* get(long *timeout*, *TimeUnit unit*)  Callable<String> task = () -> searcher.search(target);  Future<String> future = executor.submit(task); |

dfg

Stealing pool



Here if u see 2nd thread has completed all subtasks, as this is ideal, this will steal another sub tasks which belong to other thread

Here deque- is called double ended queue, means from both sides we can pick the tasks,

Note:- while using this steal pool, its our responsibility to split or fork a main task into multiple subtasks, else other idle threads cant pick those subtasks,

If there is only single task, that task will be owned by a thread, so other threads cant come and pick partially, hence better to split as sub tasks

Note:-

1. so always use thread pools , instead of dynamically creating or spawning a thread per client incurs excessive processing overhead

Fork join pool

Here forking means – splitting into smaller tasks (until u cant split further anymore)

This is based on divide and conquer, it says

If

Problem is small solve it directly

Else

Split into independent parts

Run all sub tasks parallelly (Fork new sub tasks to solve each part)

Join all those sub-tasks

1. split into individual smaller sub tasks (if u split a big task into smaller then only other ideal threads can take this smaller sub tasks,

else only that dedicated thread need to run this big task meantime if other threads are ideal they cant take as it is not a sub task)

1. all these sub tasks will be run in parallel by n threads of same core or diff core processor
2. merge the sub tasks results using join() 🡪 waiting for them to complete or

This is similar to executor service with below diff

Similarities

Here also we will submit the tasks to pool like in executor service

Difference

1. Here work stealing will be there
2. If 20 threads are there for 10 tasks, then each thread will take 1 task, now remaining 10 will be ideal (in fork join pool these will not be ideal if we define those tasks as even more sub tasks), whereas in fork join pool, we should fork means we should divide into even small parallel tasks (not IO operations like db , REST service calls as we can’t split these tasks), now when we split each task into even small tasks

Then remaining ideal threads will tasks these small tasks

When to use this

1. ForkJoinPool may also be appropriate for use with event-style tasks that are never joined.
2. We should not use this IO operations (like db calls, network REST calls)

Completable future

Here a task can be a function/ supplier,Runnable, consumer

|  |  |
| --- | --- |
| Thread pool – if all tasks to be run in parallel then submit all tasks to pool | completable future – if there are multiple dependent tasks to be run in async one after one then prefer this completable future |
| if there are 3 independent tasks, and if u want all of then to run in parallel by 3 sep threads  then create 3 runnable/callable tasks and submit to the pool , then all those 3 tasks will run in parallel | if we have 3 tasks , A,B,C where A,B should run in parallel, but c should run only after A and B completion, then use completable future ex;- cf.thenCombine(task1, BiFunction)  ex- if C wants to run only after A, then we can write that c –task in A itself  ex:-2 if u want to run like 1st A task if it completes then B, and after that C tasks – then use completable future cf.thenRun().thenRun().thenRun() |

But in real time, if c wants to be executed after A, B , we will pause C thread for A,B completion using join()

Finally if another thread is doing that task, main thread will not be blocked and it will be executed parallelly by another thread

Method of completable future:- this completable future will internally use fork join pool, if u don’t want to submit to predefined pool if u have already created pool,

u can give that pool info public static CompletableFuture<Void> runAsync(*Runnable runnable*,*Executor executor*) , so that ur task runs in ur pool

means if we do completableFuture.supplyAsync(runnable) – now this runnable task will be submitted to fork join thread pool

Difference between future and completable future

1. We cant complete a future 2) we cant combine multiple futures 3) we cant handle exceptions occurred in future

Here

|  |
| --- |
| public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {  public boolean complete(T value) {} // when we manually call complete method it will be completed, .get() / .join() methods will be giving us results back  public *T* join() // both get() and join() methods are same blocking calls to get the response but join() will never throw checked expections as part of method signature  public *T* get() // this is a blocking call which will wait untill this completes  public *T* get(long *timeout*, *TimeUnit unit*) // in real time use this get(5,timeUnit.SECONDS) // in real time block max for 1-2 seconds else throw an exception , don’t block for ever, prefer this over join()  public static CompletableFuture<Void> runAsync(*Runnable runnable*) //this will take a value and simply runs asynchronously like submitting to pool use this when we don’t want any return value  public static CompletableFuture<Void> runAsync(*Runnable runnable*,*Executor executor*) // either u can run in predefined pool or our existing pool  public static <*U*> CompletableFuture<*U*> supplyAsync(*Supplier*<*U*> *supplier*) // if we want to run some task in background but if we want to return anything from that task  public CompletableFuture<Void> thenRun(*Runnable action*) //use this method when one task completed , then run this task,  public <*U*> CompletableFuture<*U*> thenApply(*Function*<? super *T*,? extends *U*> *fn*)  public CompletableFuture<Void> thenAccept(*Consumer*<? super *T*> *action*)  // means this thenCombine will take a completable future  public <*U*,*V*> CompletableFuture<*V*> thenCombine( *CompletionStage*<? extends *U*> *other*, *BiFunction*<? super *T*,? super *U*,? extends *V*> *fn*)  public boolean complete(*T value*) // complete method is to indicate , that completable futre method is complete  public boolean completeExceptionally(Throwable *ex*)  public static <*U*> CompletableFuture<*U*> completedFuture(*U value*) // instead of manually completing, this is completed by default when u call this method  public boolean isDone()  public boolean isCancelled()  }  here both join(), get() are blocking calls  cf.get()// here if this line is exec by main thread, then main thread will be blocked |

|  |  |
| --- | --- |
| Get(5,SEC) | This method immediately returns the result of the operation if it’s already completed or blocks your thread, waiting for its result to be available.  instead of waiting indefinitely. it’s almost always a good idea to use its *T* get(long *timeout*, *TimeUnit unit*), accepting a timeout defining the maximum time your thread has to wait for the Future’s result, |

Real life examples

Make 2 rest calls simultaneously using supplyAsync(), and then combine both results and save it in database

“When the result of the long computation is available, please send its result to another long computation, and when that’s done, combine its result with the result from another query.

Completion Stage

A model for a task: that performs an action an may return a value when another completion stage completes

that may trigger other tasks

So a completion stage is an element/step of a chain

* it can be triggered by previous completion stage and
* it can trigger another completion stage
* it can be executed in a given executor

Methods

Runasync vs supplyAsync()

public static CompletableFuture<Void> runAsync(*Runnable runnable*)//use run async when we want to don’t take anything & when we don’t want to return anything from async task

public static <*U*> CompletableFuture<*U*> supplyAsync(*Supplier*<*U*> *supplier*)// when we want some return type from the task

|  |  |
| --- | --- |
| *Runnable* runnable = () -> {  ObjectMapper objectMapper = new ObjectMapper();  try {  File file = new File("sampleJsonDataEmployees.json");  *List*<Employee> employees =  objectMapper.readValue(file, new *TypeReference*<*List*<Employee>>() {  });  employees.stream().forEach(System.***out***::println);  } catch (IOException *e*) {  throw new RuntimeException(*e*);  }  };  CompletableFuture<Void> voidCompletableFuture = CompletableFuture.*runAsync*(runnable);  voidCompletableFuture.get();  System.***out***.println("compl future done"); | *Supplier*<*List*<Employee>> supplier=()->{  ObjectMapper mapper=new ObjectMapper();  File file = Paths.*get*("src/main/resources/employeesData.json").toFile();  try {  return mapper.readValue(file, new *TypeReference*<*List*<Employee>>() {});  } catch (IOException *e*) {  throw new RuntimeException(*e*);  } }; CompletableFuture<*List*<Employee>> listCompletableFuture = CompletableFuture.*supplyAsync*(supplier); *List*<Employee> empList = listCompletableFuture.join(); |

thenRun() thenAccept() thenApply()

public CompletableFuture<Void> thenRun(*Runnable action*)

public <*U*> CompletableFuture<*U*> thenApply(*Function*<? super *T*,? extends *U*> *fn*)

public CompletableFuture<Void> thenAccept(*Consumer*<? super *T*> *action*)

|  |  |
| --- | --- |
| thenAccept  public CompletableFuture<Void> thenAccept(*Consumer*<? super *T*> *action*)  *Supplier*<*List*<Employee>> supplier =()->{  File file=new File("src/main/resources/employeesData.json");  try {  *List*<Employee> employees =  new ObjectMapper().readValue(file, new *TypeReference*<*List*<Employee>>() {  });  return employees;  } catch (IOException *e*) {  throw new RuntimeException(*e*);  } }; *Consumer*<*List*<Employee>> consumer=(*employees* -> System.***out***.println(*employees*)); CompletableFuture<Void> voidCompletableFuture = CompletableFuture<Void> voidCompletableFuture = CompletableFuture.*supplyAsync*(supplier).thenAccept(consumer); voidCompletableFuture.join(); | *thenRun*  */\*\*  \* first execute task A, if that completes then run task B  \*/* public void thenRunDemo(){  *Runnable* r1=()->{  *IntStream*.*range*(1,10).forEach(*e*->{  try {  System.***err***.println(String.*format*("Preparing dish --> %d by %s",*e*,Thread.*currentThread*().getName()));  *TimeUnit*.***SECONDS***.sleep(2);  } catch (InterruptedException *ex*) {  throw new RuntimeException(*ex*);  }   });  };  *Runnable* r2=()->{  *IntStream*.*range*(1,5).forEach(*e*->{  try {  System.***out***.println(String.*format*("packing dish --> %d by %s",*e*,Thread.*currentThread*().getName()));  *TimeUnit*.***SECONDS***.sleep(2);  } catch (InterruptedException *ex*) {  throw new RuntimeException(*ex*);  }   });  };  CompletableFuture<Void> voidCompletableFuture = CompletableFuture.*runAsync*(r1).thenRun(r2);  voidCompletableFuture.join(); }  result  Preparing dish --> 7 by ForkJoinPool.commonPool-worker-1  Preparing dish --> 8 by ForkJoinPool.commonPool-worker-1  Preparing dish --> 9 by ForkJoinPool.commonPool-worker-1  packing dish --> 1 by ForkJoinPool.commonPool-worker-1 |
| supplyAsync().thenAccept()  *//we want mail ids of who is new joiner and who didnt complete training*  *//we want mail ids of who is new joiner and who didnt complete training* public void supply\_thenApply\_thenAcceptDemoV1() throws IOException {  *Supplier*<*List*<Employee>> listSupplier = () -> {  try {  System.***out***.println(String.*format*("Completely fetched all emp by %s",Thread.*currentThread*().getName()));  File file = new File("src/main/resources/employeesData.json");  *List*<Employee> employees = new ObjectMapper().readValue(file, new *TypeReference*<*List*<Employee>>() {  });  return employees;  } catch (IOException *e*) {  throw new RuntimeException(*e*);  }  };  CompletableFuture<Void> voidCompletableFuture = CompletableFuture  .*supplyAsync*(listSupplier)  .thenApplyAsync(*list* -> {  System.***out***.println(String.*format*("applyAsync() method running by %s", Thread.*currentThread*().getName()));  return *list*.stream().filter(*emp* -> {  return "TRUE".equalsIgnoreCase(*emp*.getNewJoiner()) && "TRUE".equalsIgnoreCase(*emp*.getLearningPending()) ? true : false;  }).collect(Collectors.*toList*());  })  .thenApplyAsync(*list* -> {  System.***out***.println(String.*format*("applyAsync() method running by %s", Thread.*currentThread*().getName()));  return *list*.stream().map(*emp* -> *emp*.getEmail()).collect(Collectors.*toList*());  })  .thenAccept(*list* -> {  System.***out***.println(String.*format*("thenAccept() method running by %s", Thread.*currentThread*().getName()));  System.***out***.println(*list*);  });  voidCompletableFuture.join();  System.***out***.println("main thread waited for all tasks completion, now its resumed");  }  *simple easily understandable*  public void supply\_thenApply\_thenAcceptDemo() throws IOException {   *Supplier*<*List*<Employee>> supplier=()->{  File file=new File("src/main/resources/employeesData.json");  try {  *List*<Employee> employees = new ObjectMapper().readValue(file, new *TypeReference*<*List*<Employee>>() {});  System.***out***.println(String.*format*("Completely fetched all emp by %s",Thread.*currentThread*().getName()));  return employees;  } catch (IOException *e*) {  throw new RuntimeException(*e*);  }  };  *Function*<*List*<Employee>,*List*<Employee>> filterFunction=(*list*)->{  *List*<Employee> existingEmployees = *list*.stream().filter(*e* -> {  return "FALSE".equalsIgnoreCase(*e*.getNewJoiner()) ? true : false;  }).collect(Collectors.*toList*());  System.***out***.println(String.*format*("Completely filtered all emp by %s",Thread.*currentThread*().getName()));  return existingEmployees;  };   *List*<Employee> allEmpls = CompletableFuture.*supplyAsync*(supplier).thenApply(filterFunction).join();  System.***out***.println(allEmpls);  } | |

Whats difference between thenApply() and thenApplyAsync() methods

public <*U*> CompletableFuture<*U*> thenApplyAsync(*Function*<? super *T*,? extends *U*> *fn*)

public <*U*> CompletableFuture<*U*> thenApply(*Function*<? super *T*,? extends *U*> *fn*)

but my opinion is if u want to run task-B after task-A then single thread itself can handle the same na? 1st thread will execute A ,

once it completes then same thread will take another task- B – so in this case, why we should bother which thread executes which task, what we want is diff sequence na?

Synchronous vs Asynchronous Execution

* **thenApply**: The callback executes on the same thread that completed the previous CompletableFuture, which could be the thread running an async task or even the main thread if the future was already completed.
* **thenApplyAsync**: Guarantees the callback runs asynchronously, in a different thread either from the ForkJoinPool.commonPool() or a custom executor (if specified).

thenCombine()

Misc concepts

Creating a thread in java 21 way

Count down latch

If there are 5 threads, and if u want main thread to wait for those 5 threads completion then we can achieve this In 2 ways

1. Thread.join() – worst and not used in prod
2. Count down latch – internally it will maintain a counter, when any thread completes its exec we should dec the counter, main thread will be blocked at that line it self

Untill count is zero , once count is zero, then main thread will continue its execution