Concurrent programming

Kristoffer Klokker

2022

Contents

1 Introduction

Concurrency is the act of having multiple execution done simultaneously which interact with each other.
This is done to utilise multiple CPU cores rather than rely on CPU speed.
Not only this but instead of having single powerfull computers, bigger networks of computers can be used.
The benefits comes at a cost of complexity, due to the all possible outcomes of different timed execution.
The risk includes race-conditions where variables are changed while used in another thread or liveness where threads are stuck forever waiting for a non realeased variable nad such.

2 Anonymous,- and lambdafunctions

For at simple class which is given in an argument, instead of creating a class and then parsing it, the class can be created in the argument field.
For instance a class which implements comparable, it can be programmed as such:

```
1  public interface StringExecute {
2      public void run(String content);
3  }
4
5  public static void doAndMeasure( StringExecutable
       runnable ) {
6      long t1 = System.currentTimeMillis();
7      runnable.run();
8      System.out.println( "Elapsed time: " + (System.
           currentTimeMillis() - t1) + "ms" );
9  }
10
11 public static void anonFunc() {
12     doAndMeasure(new StringExecute() {
13         public static void run(String content) {
14             System.out.println(content + " Wow!");
15         }
16     });
17 }
18
19 public static void lambdaFunc() {
20     doAndMeasure( (content) -> System.out.println(
```

```
        content + " Wow!"););
}

public static void lambdaFuncOpt() {
    doAndMeasure( (content) -> System.out::println;); //
        Only prints content and not + " Wow!"
}
```

Here lambda function is only possible due to the compiler knowing what type of object is created due to restrains from the function and the runnable interface only contains a single run function. Another use of lambda expression is when working with maps.

```
public static void main() {
    String text = "Hello world hope your having a good
        day!";
    Map<Charachter, Interger> occurrences = new HashMap
        <>();
    for(int i = 0; i < text.length(); i++){
        final char c = text.charAt(i);
        if(occurrences.containsKey(c))
            occurrences.put(c,occurrences.get(c)+1);
        else
            occurrences.put(c,1);
    }
    for(int i = 0; i < text.length(); i++){
        final char c = text.charAt(i);
        occurrences.merge(c,1,(currValue,value) ->
            currValue+value);
    }
```

Both for loops do the same, the second uses merge which takes a position (c) and a default value (1) and a bifunction which is a function with two inputs. The arguments will automaticly be assigned such currValue is the current hash value and value is the same as the default value 1.
Some of the most usefull built-in functional interfaces includes:

- Predicates - 1 argument returns boolean
- Functions - 1 argument returns result
- Suppliers - Like Functions but no arguments
- Consumers - 1 argument no return
- Comaprators - implements compareTo

3 Streams

Streams are like an foreach and like the name it is a stream of data. In more fine words it is a monad which is datastructure of a sequence of steps of operations.
There are different types of sterams, the *Stream* is an object stream, wheras primitives stream also exists $IntStream, LongStream, DoubleStream$ with poosibilites like $IntStream.range(1,4)$ which has a stream of 1,2,3 and $IntStream.range(1,4).sum()$
To tell the compiler if an object stream is transformed into a primitive type $.mapTo(Int, Double, ...)$ with a parser as argument.
The most common functions used by streams are:

```
40 List<String> words = Arrays.asList("watercan", "digital"
       , "citizen");
41 //Performs function with each element
42 words.stream().forEach(word -> System.out::println);
43 //Modifies the element
44 words.stream().map(word -> String::length);
45 //Only elements which fullfill the function will 'parse'
46 words.stream().filter(word -> word.startsWith("d"));
47 //Streams the sorted stream
48 words.stream().sorted((s1, s2) -> s1.compareTo(s2));
49 //Counts the number of elements
50 words.stream().count();
51 //replace element by stream of given function
52 words.stream().flatmap(word -> Stream.of(word.split("a")
       );
53 //Gather elements in stream, for analyzis or into an
       object ex list
54 words.stream().collect(Collectors.toList());
55 //Reduce takes a stream and makes a single element Ex.
       this takes the longest string
56 words.stream().reduce((s1,s2) -> s1.length() > s2.length
       () ? s1 : s2);
57 //Stops all stream even if parrallel and returns true if
        found oterwhise keeps running returns false if not
       exist
58 .anyMatch(word -> word.equals("test"));
```

The functions can then be chaineed together, so methods which returns a stream can be worked upon.
When working with file stream it is done as following:

```
59 try( Stream< String > lines = Files.lines(Paths.get("
      text.txt"))) {
60     lines.forEach(System.out::println);
61 }
62 catch ( IOException e) {
63     e.printStackTrace();
64 }
```

By using the try and putting it into the parenthesis it will cause the stream to close when the stream is finished.
When a terminal operation (returns result rather than intermideate which returns stream) is done one the stream, it will close the stream. To counteract this a function which returns the stream may be used as such:

```
65 Supplier<Stream<String>> streamSupplier =  () -> Stream.
      of("d2", "a2").filter(s -> s.startsWith("a"));
66 //Returns true due to there is elements in the stream
67 streamSupplier.get().anyMatch(s -> true);  //
68 //Returns false due to elements existing in stream
69 streamSupplier.get().noneMatch(s -> true);
```

3.1 Parrallel streams

When working with parrallel stream the stream is outsourced to threads using a common ForJoinPool.
This makes stream elements being handled side by side.
Some function needs modifications such as reduce:

```
70 words.stream().parallelStream().reduce(0,(sum,s) -> sum
      + s.length(), (sum1, sum2) -> sum1+sum2);
```

Reduce is now broken up onto three arguments,

- indetifier - initial value
- accumulator - takes current result and element and operatoes
- Combiner - takes two partial accumulators and combines them

Like sot the *sort*() function will either wait for all parralle streams or use parralell sorting in large amounts.
Be aware that parrallle streams will take the main thread into the work, therefore when accessing the hardrive or such, all paths should be gathered beforehand.

4 Bugs and solutions of parallelizing

When working with parrallel threads it is important to encapsulate as much as possible. Threads may access data outside it thread, but this can lead to some of the following bugs.

4.1 Race conditions

This is a bug which accour when two threads are accessing the same global variable.
This is a bug based upon atomicity is not kept and an operation is not done before another access data from the operation.
The most simple form is two threads which are incrementing a variable.
Say that the first threads read the variable, at the same the second thread reads the variable.
This will conclude in both threads having the same value for the variable and when incrementing the variable will only increment once.
To counteract this lock can be used. This is where code snippets are ran syncronised and threads are not able to execute twice.

4.2 Unmutable objects

Most often objects are a problem of race conditions.
To counteract this an object is made unmutable.
This means the object:

- is not including setters
- is not including references to itself
- has only final methods and variables
- is private such no method is overwritten

4.3 Low level thread locking

The low level method for locking threads used in libraries uses an object and it ability to wait() and notify().

```
71 synchronized(monitor) {
72     try {
73         monitor.wait();
```

```
74      } catch(InterruptedException e) e.printStackTrace();
75  }
76  .
77  .
78  .
79  synchronized(monitor) { monitor.notify();}
```

This code will in one thread use the try catch which will wait for the object called monitor. Then in another thread the monitor is notified.
The problem is if the monitor is notified before the other thread is waiting the thread will never escape the wait.
So another boolean is needed such after a notify it is changed and the try is in an if to the boolean.

5 Threading in Java

Threading is done thorugh the *Thread* object which takes a runable object as an argument.
The thread is then executed by *start*() and *join*() is used as a wait till a thread is done.
If join is not used the main thread will stop its execution before child threads are done.

```
80  Thread t1 = new Thread( () -> {for(int i = 0; i < 1000;
       {i++; System.out.println(i);})});
81  t1.start();
82  try{
83      t1.join();
84  }
85  catch(InterruptedException e) {e.printStackTrace();}
```

The try is due to threads may be interrupted, which otherwise would case the program to halt, due to the thread never joining.

5.1 Syncronize

In java the syncronize keyword is used to lock objects or methods, such race conditions are prevented.
An exampel of the use of syncronization can be seen below.

```
86  private static class Counter {
```

```
87      private int c = 0;
88  }
89
90  public static void main() {
91      Thread t1 = new Thread( () -> {for(int i = 0; i <
            1000; i++) {
92          synchronized( counter ) {
93              counter.c++;
94          }
95      });
96      t1.start();
97      Thread t2 = new Thread( () -> {for(int i = 0; i <
            1000; i++) {
98          synchronized( counter ) {
99              counter.c++;
100         }
101     });
102     t2.start();
103     try{
104         t1.join();
105         t2.join();
106     }
107     catch(InterruptedException e) {e.printStackTrace();}
```

The synchronized will here lock the counter object before access from which the code is executed. This will ensure that c is equal 2000.
Another way is to making a whole method syncronised by:

```
108 private synchronized void increment() {
109      i++;
110 }
```

5.2 Spin lock

Spin locks are an alternative to syncronizing.
The way it works, is a global atomic boolean variable is set, and used as being true if the object is locked.
Then instead of writing syncronized it can be written as followed:

```
111 Thread t = new Thread( () -> {for(int i = 0; i < 1000; i
        ++) {
112     while(!locked.compareAndSet(false,true)){}
113     counter.c++;
```

```
114     locked.set(true);
115     }
116 });
117 }
```

Here compareAndSet is an atomic function which compares to the second argument an returns true if equal. If not then it reutrns false and set the value equal to the first argument.

5.3 Latches

When working with many threads, instead of using each threads join method, a CountDownLatch can be used.
The latch is initialized with the amount of threads which will be used: CountDownLatch latch = new CountDownLatch(10);
Then every thread should end with: latch.countDown();
Because then when joining the following can be done:

```
118     try{
119         latch.await();
120     }
121     catch(InterruptedException e) {e.printStackTrace();}
```

Latches may also be used, to wait for all threads to be initialized before starting, in case of benchmark or such.

5.4 Future task

A future task object, is a threading object, which takes a task.
The object can be called with *.get*() which will return the result of the task.
When the task is done, the object stays in the finished state.
This can be used in case of a cache, where earlier results are in a hash table in form of future tasks.

5.5 CompletableFuture

Much like a future task, but has advantages, in some scenarios.
The CompletableFuture are able to subit task itself to a threadPool and can handle themselves after done.

```
122 CompletableFuture<Void>[] futures = files.foreach( file
    -> CompletableFuture
123     .supplyAsync( () -> task())
124         .thenAccept(fileInfo - > task())
125     .collect(Collectors.toList()
126         .toArray(new CompleteableFuture[0]);
127 CompletableFuture.allOf(futures).join();
```

Here the task is created with CompleteableFuture.suppeAsync, handled when done by thenAccept, and all threads are shutdown after by the allOf().

5.6 Semaphores

Semaphores are objects, which is used to manage the amount of threads in a pool.
It does this using permits, which are given to a thread to start a task.
When the task is done the permit is given back to the semaphore.
These are very usefull in cases of API's which has a maximum amount of connections.

5.7 Barriers

Barriers are a lot like lateches.
These work to ensure every thread is at the same point.
This may be used in simulations, where data from another thread is needed, and it is needed to know that the data is already computed or at the same point in time.

5.8 Iterating over safe objects

Iterating over a thread safe object may lead to problems. The most common method of to simply try iterate and in case of a change in the collection a *ConcurrentModificationException* will be thrown.
There is other solutions including locking the obejct while iterating or creating a clone of the object and iterator throught it.
Remember to be carefull of hidden iterations, this can be things like *toString*() which iterates through some object and converts it into chars for a string.

5.9 Thread safe classes

Threads safe classes are a good way to make code thread safe. These classes includes ConcurrentHashMap and CopyOnWriteArrayList.
These classes does not eliminate concurrency errors, self made check then write errors can still accour, but the classes include function for thos purposes.
THe most optimal methods are still if possible using the collect method of a stream, this is thread safe and faster.

5.10 Using queues

Queues provide a method for using the producer consumer pattern.
This pattern uses most often block queues to store task, created by producer which consumers can perform when possible.
The pattern also helps making code more readable, by dividing it into creating task and performing tasks.
Queues also handles throttling due to it being able to handle a maximum amount of tasks which is being asked for at once.
The queue acts as a middleman for storing the tasks and ensures ownership of objects are changed.
In case of the consumer also being a producer a deque (pronounced 'deck') is used, which is a queue with fast accessing both head and tail, to insert and remove tasks.
This can be implemented as:

```
128 int maxThreads = Runtime.getRuntime().
       availableProcessors() -1;
129 CountdownLatch latch = new CountDownLatch(maxThreads);
130 final BlockingDeque<Optional<Path>> tasks = new
       LinkedBlockingDeque<>();
131 IntStream.range(0,maxThreads).forEach( i -> {
132     new Thread(() -> {
133         try {
134             Optional<Path> task;
135             do {
136                 task = tasks.take();
137                 task.ifPresent( object -> function(
                       object));
138             } while(task.isPresent());
139             tasks.add(task);
140         } catch(InterruptedException e) {}
141         latch.countDown();
```

```
142     }).start();
143 }
```

This code gets the number of avaliable threads minus the current thread, the creates a queue and latch.
Then threads are created according to the amount of avaliable threads, which takes task from the queue until an empty Optional.
It adds it again so other threads also can stop.
The reason the use a pool of threads is due to the time makign threads and too many threads needs switching between by the CPU if there are more threads than cores.

5.11 Executor framework

The executor framework is a framework to work with pools of threads.
It implements a number of threads and a task queue. By defualt the newWorkStrealingPool() should be used due to it optimizing itself in many scenarios.
Other implementations may be newFixedThreadPool(n) and newCachedThreadPool() which creates new threads and cache them for a small time after.

```
144 private static final Executor exec = Executors.
       newWorkStealingPool();
145 exec.submit(() -> task());
```

This code creates a form of executor with a fixed number of threads, from which execute on runnable called task.
The framework also support ScheduledThreadPoolExecutor for timed tasks which should be used over the timer class.
To shutdown the executor the following can be done:

```
146 try{
147     executor.shutdown();
148     executor.awaitTermination(1,TimeUnit.DAYS);
149 } catch (InterruptedException e) { e.printStackTrace();}
```

5.12 ExecutorCompletionService

This is a object which helps with working with futures.
The point of the object is to, gather all futures submitted to threads from

which it can return finised futures, such there is no order of which they need to finish.

```
150 ExecutorService executor = Executor.newWorkStrealingPool
        ();
151 ExecutorCompletionService<Map <String, Integer>>
        completionService = new ExecutorCompletionService<>(
        executor);
152 for..
153     completionService.submit( () -> task());
154 for everyTask
155     Map<String, Integer> file = completionService.take()
            .get();
```

5.13 Thread optimization

To use threads efficiently are states used. These stated includes BLOCKED, WAITING or TIMED_WAITING.
By using these states threads, can be interrupted and used for another task meanwhile waiting or being blocked.
A thread can also be interrupted by another thread. Though the thread will continue until a point which makes sense, which may be when it is done.

6 Safe shared objects

Shared objects between threads can introduce many dangers, mainly that thread can be sure to read the same value it just wrote.
Stale data is term for none up to date values from shared objects which can lead to non expected outcomes.
A solution for this is syncronizing the shared object. By this only one thread can access the shared object at a time, therefore ensuring no stale data.
Another solution is all variables being local to each threat

In Java there is a safety gurantee to variables, that once updated they are fully dated. Unless the type of variable is long or double, due to the 64 bit required data, it is updated by two times of 32 bit each, therefore half updated values may accour.
Syncronization is done by locking the given object, such no other thread can access it.

Volatile variables are a weak syncronised variable, due to not ensuring atomicity. It is not as safe, but can be used for veryfying states rather correctness. Therefore volatile variables should only be used when its value does not depend on current value or only one threads updates aswell as the variable not participating in invariants.

Ad-hoc thread confinement is a way of describing how an object may only be in certain safe ways. This is ofcourse not a stable way of implementing objects.
A more stable alternative is stack confinement, where reference an variables are all local to a thread.
ThreadLocal solves helps to ensure no leaks could accour in stack confiment, by making an object with variables which can not be accessed outside a thread.
Another way is an immutable object, which is a final object which only contains final fields, can not be modified and construction has not been escaped. It is not enough to just make the object reference final, due to the object itself being able to change its variables.
The immutable object may also be good in case of outside methods, this ensures that the methods may not change anything or save a reference.
A object may be treated as ummutable such as the date object, but it requires clear documentation.

6.1 Constructing objects

When constructing objects it is important, that a reference to the object not escape before initialization.
This can result in half initialized objects being treated as done.
To prevent this a static initializer may be used, which creates the object, parses it to a method which makes it public and returns the reference.

6.2 Thread safe classes

First the invariant variables are identified and variables for object states. Such they can get policy according to use.
State-depend operations are preconditions on operations, which also acts as a policy for the class.
It will always be easier to encapsulate methods for objects such syncronised always has the correct lock.
Remember to always use thread safe classes, to make life easier but still re-

member to handle them safely.
When access multiple variables wich relates send them as tuples, such they always are syncronized.
A thread safe class may include public variables, which has to be handled safe but in some cases it makes sense to make them public.
Some class may have to be extended with a syncronised method, such as a list extension with a method for adding of absent.
A custom lock may also be but in a method Ex. public boolean m() { synchronzed(list) {...}}, use the list variable as a custom lock.

7 Cancellation and Shutdown

A thread may be shutdown, due to user input or another thread already finding the answer.
The most simple solution is a volatile boolean, which is set to true when a thread should be cancelled.
Then in the thread check for the boolean is done if true then return.

7.1 Interruption

Interruption is a great way to cancel a thread, but it got downsides.
When a thread is interrupted a flag in the thread is raised and when it reaches a cancellation point it will stop.
To ensure a thread is cancelled correctly a new thread class as an extension of Thread is created such the cancel() can be overwritten.
The most simple cancel method is for it to just call interrupt(), which raises the flag, and when a cancel point is reached an interruptedException is thrown in run().
In case of insuring code is stopped due to no cancel point is using Thread.currentThread().isInterru
which returns the interrupted status. It should be noted that if true the flag is removed so, raising it again is often needed.

Threads may have different policies towards what an interruption should result in.
Most often either it propagates the interrupt, by the method throwing an InterruptedException.
Or restore the interupt, this is done by in the catch to call interrupt() again.

Some tasks may not be cancellable, these should keep trying in a loop and

catch possible interrupts which then set a global boolean to true. Such when the work is done in a finally if the global boolean is true is run Thread.currentThread.interrupt(), such the interrupt still shows.

Some code may rely on a socket or a like where the cancel may also have to close a socket to shutdown a thread.
In case of a task with a limited time, a Future can be used. Future have the method .get(time,unit) which starts athread and ends it in the given time.

7.2 Shutdown of thread-based service

There are cases where a shutdown of multiple threads have to be initiated.
ExecutorService got two methods for clossing all threads shutdown() and shutdownNow() which returns infinished tasks.
For a more manual approach a poison pill can be used. This way on the queue a task representing the poison pill is placed. Once a consumer meets the pill it stops.
By this method the queue will always be emptied before shutdown. The biggest problem is the amount of consumer is needed to be known, such an equal amount or more is placed in the queue.
The problem with shutdownNow() is some tasks may get lost if they where halfway done. An extension of AbstractExecutorService can here be made such a more custom approach may be taken to the given task.
If a thread is running poor code, a handler for runtime exceptions may be needed. Unlike normal program flow a runtime exception will crash a thread but the main program will continue.
For this the UncaughtExceptionHandler inmterface can be used, for handling the exception in a thread.

A shutdown hook may be needed in some cases, to controle the order which threads are shutdown, so fx. a logger thread is closed last.

8 Liveness hazards

Liveness hazards are different program breaking problems which can occur due to bad programming.

8.1 Deadlocks

Deadlocks are the case when two threads are holding a lock and waiting for eachothers lock.
The most easy way to ensure this does not happend is to keep the order of locks the same.
This is done manually or if locks are user input the order can come from the smallest hash code.
To make life easy methods should always use open calls, which a method calls outside locks. Calls inside locks may leads to unexpected deadlocks.
A deadlock can also be on a resource. This may be a lock upon a connection, socket or CPU, which also may lead to a deadlock. In cases where the locking can not be assured the tryLock which waits for a lock in a given time. This is not ideal but may be a solution.
In case of a deadlock the thread dump will include information about which locks and from where the deadlock accoured.

8.2 Starvation

A starved thread is a thread which does not have access ressource most often the CPU.
This most often occour when threads are forced into thread prioties.
Normally the OS can prioties threads such CPU starvation is not an issue.
The only cause for forcing a priority is for big background tasks.

8.3 Livelock

Livelocks are a form of infinite loops.
The loop may always lock some object or just result in starvation.
The loop is often a problem aroused of non recoverable errors which is treated as recoverable.
The result of this is the error is repeatebly tried, to just result in a new error task.

9 Performance and Scalability

The first priority is always getting it to work. From this point performance may be optimized, but only if needed due to optimization often leads to bugs.
When talking about performance there are different forms, mainly througput and speed.

When optimization it is therefore needed to know what performance should be optimized and in which scenarios, a lot of light work or heavy load.
To begin optimization a library can be used to measure times stamps such, the slow areas can be detected rather than guessed.

9.1 Amount of threads

The amount of threads often has to be well though through.
The easy solution is for simple application match the number of cores the number of threads.
But in some cases more may lead to faster work, due to if a thread is locked out and waiting another thread can be context switched in.
Too many threads will also slow down due to too much switching when not needed.
vmstat can be used to see amount of switches and kernel usage, which should be under 10%

9.2 Locks

Locks should be as minimal as possible, to some extend such four locks should not be in a row but rather together.
If a thread seems to alone using a single object, a lock may be put on anyways, due to the compiler being able to remove the synchronization.
A way to optimize is reducing the scope of a lock. An example is the the concurrentMap which divides the table list into section such smaller sections are locked.
Concurrent version of ovjects should always be used, such as atomic variables and concurrent maps and such.