# Introductie tot threading

Een thread is een set sequentiele stappen die worden uitgevoerd door een rekenkern van het systeem.

Threading is simple gezegd een software draad opdelen in meerdere delen voor de processorkern(en) zodat deze sneller opgelost kunnen worden.

Met een multicore processor wordt dit zo gedaan door multithreading. Hierdoor wordt het process opgedeeld door meerdere kernen, deze voeren dan paralell de instructies uit(een function call, een loop etc). Omdat het process wordt opgedeeld in meerdere delen en tegelijk uitgevoerd wordt is het process sneller af te handelen door de processor.

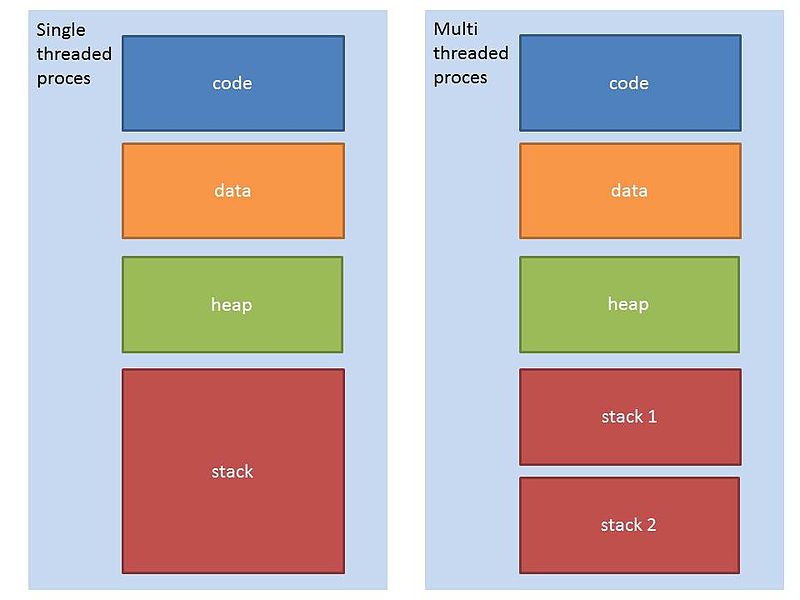
Op een CPU met een enkele processor wordt mogelijk gebruik van threading door middel van time slicing. Dit houdt in dat de processor wisselt tussen software threads. Hierdoor doet nog steeds de enkele processor al het werkt, maar worden er meerdere delen van de thread gedaan waardoor het lijkt alsof er processen in parallel lopen terwijl dit niet zo is.

Voorbeeld van single core en multithreading (vertaald van een reddit post maar zeer duidelijk)

Voor nog een simple voorbeeld, Stel je voor dat je een grote maaltijd maakt voor thanksgiving. Elk recept is een lijst instructies net zoals een thread en de persoon die het kookt is net een processor. Je zou de kalkoen alvast in de oven kunnen stoppen en dan beginnen aan een taart. Dan daarmee stoppen om aardappelen te pureren. Je werkt maar tegelijk aan één recept maar maakt wel voortgang op alle recepten omdat je steeds tussen recepten wisselt. Als je een tweede persoon zou toevoegen kun je aan twee recepten tegelijk werken, net alsof je twee processoren hebt.

Uit dit voorbeeld is duidelijk te zien hoe een tweede kern het werk kan versnellen door de recepten te verdelen door twee personen.

In figuur 1 is te zien hoe de threads verdeeld worden.

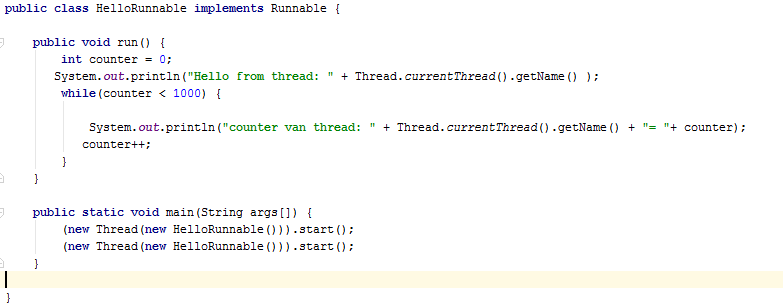


Figuur 1

Wat hieruit blijkt is dat alleen de stack verdeeld wordt en de rest niet. Dit heeft als gevolg dat er geen contextwijzingen zijn tussen de threads en dit leidt tot een snellere oplossing.

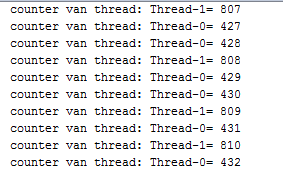
# Threads implementeren met Runnable en Thread.

Als je een programma wilt laten werken met threads, met je de klasse de interface Runnable laten implementeren.



Nadat je de klasse hebt aangemaakt moet je daarin de methode run() aanpassen naar de uitvoer van wat je wilt dat er gebeurt. run() MOET je implementeren, dit is de methode die door Thread.start wordt aangeroepen.

Daarna maak je een nieuwe instantie van de door java geleverde klasse Thread en geef je als parameter een nieuwe instantie van de klasse die runnable implementeert. Daardoor worden er twee threads aangemaakt en die



In het voorbeeld is te zien dat de tellers steeds apart optellen, dit gebeurt allebei op een aparte thread ze zijn dus allebei “tegelijk” bezig met het runnen van hun deel van het programma.

Er is een andere manier omdat te doen die niet bij onze user story aan bod komt, mochten jullie dit willen zal ik dit ook nog uitwerken. Runnable implementeren is meest gebruikte manier.

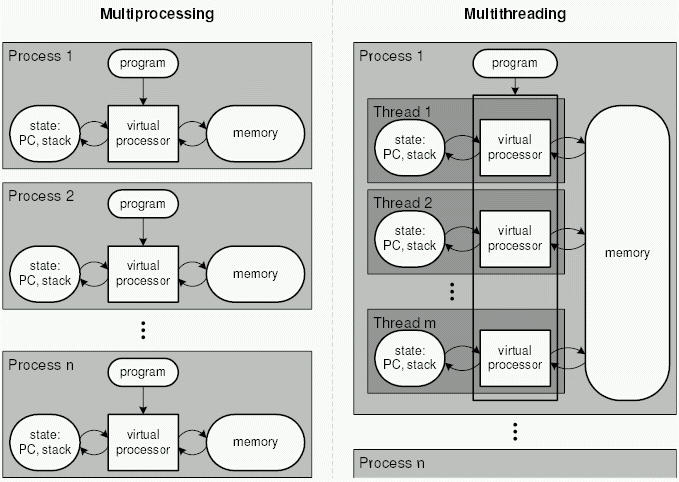
# Verschil tussen multi-threading en multi-processing

Het verschil tussen multithreading en multiprocessing is dat processes per proces een ander plekje in het geheugen innemen, waardoor er niet zo gemakkelijk onderlinge communicatie mogelijk is zoals bij multithreading het geval is. In het voordeel van multi-processing heb je dat er meerdere processen kunnen gemaakt worden over meerdere CPU’s, terwijl je bij multithreading vast zit op één CPU (maar wel meerdere cores op diezelfde CPU).

Verder zou je kunnen beargumenteren dat je met meerdere processen wat immuner bent tegen fouten en crashes: als je bijvoorbeeld twintig processen hebt en in ééntje ontstaat een fatale fout, dan crasht alleen die ene waardoor je nog negentien processen overhoudt. Als je dit nou met multithreading vergelijkt: alle threads zitten op één proces, dus als er een fatale fout gebeurd bij een thread, dan crasht als gevolg dat ene proces.

Het nadeel hiervan is wel dat meerdere threads sneller zijn dan dan meerdere processen, omdat threads veel minder geheugen innemen en je hoogstwaarschijnlijk meer benodigdheden hebt voor processen. In het geval van Java maakt elk nieuwe proces een nieuwe Java Virtual Machine aan, wat ten koste gaat van bijvoorbeeld de opstarttijd, terwijl een thread dat niet doet.

*Onderstaande afbeelding laat in een snel opzicht het verschil zien tussen multiprocessing en multithreading.*



# Threading en de rol van de JVM, een besturingssysteem en de processor met zijn cores hierin spelen.

In het geval van multithreading wordt er eenzelfde Java Virtual Machine gebruikt, waardoor objecten en attributen met de threads gedeeld kan worden, terwijl de Java Virtual Machine maar één proces is. Als er binnenin die JVM gevraagd wordt om een andere thread te maken, dan wordt er één thread binnen die JVM gemaakt. Het besturingssysteem regelt dat deze threads op verschillende cores zitten waardoor de snelheid veel hoger ligt dan wanneer het allemaal op één core ligt.

# Een thread correct starten en stoppen en de beperkingen van de API bij het stoppen.

## Starten

Om een thread correct te starten moet je gebruik maken van de start methode. Deze methode zit in de Thread class of in de Runnable interface, die je dus moet uitbreiden of implementeren zoals hieronder gebeurd:

|  |
| --- |
| public KlasseNaam extends Thread |

|  |
| --- |
| public KlasseNaam implements Runnable |

De aanbevolen manier is door de Runnable interface te implementeren, al kan dit er wel voor zorgen dat de code wat uitgebreider word: als je een Thread extend kan je namelijk gewoon start op een instantie van die klasse aanroepen, maar als je Runnable implementeert moet je een nieuwe instantie maken van de Thread klasse, met als argument jouw eigen klasse, en daarop moet je de start methode aanroepen. Waarschijnlijk is het wat makkelijker om te tonen in code:

|  |
| --- |
| new KlasseNaam().start() // KlasseNaam extends Thread |

vs.

|  |
| --- |
| new Thread(new KlasseNaam()).start() // KlasseNaam implements Runnable |

De run methode wordt daarna automatisch aangeroepen, en de thread is dus klaar met taken uitvoeren wanneer hij bij het eind is van de run methode.

## Stoppen

KlasseNaam.currentThread().interrupt(), en dan checken of KlasseNaam.currentThread().isInterrupted() of een while loop met een volatile variable (zoals running) zodat de functie eindigt

Het stoppen van de threads is niet zo eenvoudig, met verschillende manieren om te stoppen en verschillende beperkingen per methode. Je ‘stopt’ de thread namelijk door aan het eind van de run functie te komen, en gebaseerd op het doel van jouw applicatie kan het verschillen hoe je dat doet. Hoe je een thread kan stoppen is als volgt:

* Jouw run methode is aan het eind van de functie beland;
* Jouw run methode retourneert iets, en gebaseerd op jouw code kan er een situatie ontstaan waar je niets wil retourneren maar alsnog moet om de thread te stoppen;
* Je roept de Thread-methode ‘interrupt’ aan. Let wel op dat dit niet per se de thread stopt, want de ontwikkelaar kan zelf aangeven wat de thread moet doen. Je kan dan checken of jouw thread ge-interrupt is door de Thread-methode ‘isInterrupted’.
* Je creëert een variabele in jouw thread die volatile is, en checkt met bijvoorbeeld een while loop of die variabele true, en als dat zo is dan word een taak uitgevoerd. Volatile variabelen zijn variabelen die niet geoptimaliseerd worden door de Java compiler, en kan je gebruiken om aan te geven dat de waarde van de variabele aangepast gaat worden door meerdere threads. Dit is in mijn ogen de meest gebruikte manier om een thread te stoppen;

Je beëindigt het proces waar de thread in draait, alleen wil je dit natuurlijk niet in elke situatie doen.

# Kan het keyword synchronized correct inzetten om te voorkomen dat threads tegelijk een gedeelde resource benaderen.

Stel dat je Java-programma twee threads creëert, met beide als doel om van nul naar honderd te tellen en elke stap te melden aan de console. Zonder synchronized zou dat betekenen dat er in de console heel vaak niet van correct van nul naar honderd geteld wordt. Je zou dan bijvoorbeeld 1, 1, 2, 3, 2, 4, 3 zien.

Met behulp van synchronisatie kan maar één thread per keer de gesynchroniseerde methode aanroepen, waarna threads die die methode ook willen aanroepen moeten wachten tot de andere thread klaar is met die methode.

Het stukje tekst hierboven kan je als volgt uitwerken:

|  |
| --- |
| public class Example {  public static void main(String[] args) {  Counter counterA = new Counter();  Thread[] threads = new Thread[20];  // Maak de threads aan  for (int i = 0; i < threads.length; i++) {  threads[i] = new Thread(new CounterThread(counterA));  }  // Start de threads  for (int i = 0; i < threads.length; i++) {  threads[i].start();  }  } } |

|  |
| --- |
| public class CounterThread implements Runnable {  protected Counter counter = null;   public CounterThread(Counter counter) {  this.counter = counter;  }   public void run() {  for (int i = 0; i < 100; i++) {  this.counter.add(i);  try {  Thread.sleep(20);  } catch (InterruptedException e) {  System.out.println(e);  }  }  } } |

|  |
| --- |
| public class Counter {  long count = 0;   public synchronized void add(long value) {  this.count += value;  System.out.println("Toegevoegde hoeveelheid: " + value + " - Totale hoeveelheid: " + this.count);  } } |

Wanneer je de functie add in de klasse Counter niet gesynchroniseerd maakt, dan zou je uiteindelijk een verkeerde totale hoeveelheid hebben omdat alle threads tegelijkertijd proberen om de variabele count aan te passen. De correcte hoeveelheid is 99000. Zonder synchronized is de kans heel erg groot dat je lager dan dit getal uitkomt, want soms wordt de functie add door meerdere threads aangeroepen.