Semesteroppgave: samtidig programmering

DATS1600 oppgavesett 7.

I dette oppgavesettet skal vi forbedre ytelsen til nextGeneration metoden ved hjelp av samtidig programmering (eng: concurrent programming). Spillebrettet skal 'deles opp' og evalueres separat av samtidig kjørende tråder i datamaskinen, slik at evalueringen av en ny generasjon gjøres raskere.

Funksjonell og samtidig programmering i Java presenteres i forelesning i uke 15 og 16 og en oppfrisker tutoriatime holdes i uke 18.

Merk: et alternativ til denne oppgaven er utvidelsesoppgaven fra oppgavesett 4 (GPU programmering). Hvis dere utfører denne oppgaven, så er det ikke nødvendig å utføre dette oppgavesettet.

Introduksjon til prosesser og tråder

Før dere starter med denne oppgaven, vil det være naturlig å sette seg i konseptet med prosesser og tråder, og hva som støttes av datamaskinen dere bruker.

Tradisjonelt, er ytelsen til en prosessor (dvs. en CPU) målt i antall sykluser den kan operere i per sekund (angitt i hertz, Hz). Antall sykluser per sekund refereres til prosessorens *klokkefrekvens*. I moderne systemer, er 4-5 GHz klokkefrekvens antatt som bra. Høyere frekvens er mulig (over 8 GHz kan oppnås), men da trenger man 'eksotiske' former for kjøling av CPU kjernen (husk at vibrasjoner omgjøres generelt til energi og varme), som for eksempel bruk av flytende nitrogen.

I moderne tid, bruker man andre faktorer til å måle (og forbedre) ytelsen til prosessorer. For eksempel, spiller arkitekturen til prosessoren en viktig rolle. I de siste 10 årene, har utviklere av prosessorer (f.eks. Intel og AMD) forlatt fokuset på høye frekvenser, fordi eksotiske kjølesystemer er kostbart, og gått over til lavere klokkefrekvenser og flere *kjerner*.

En CPU med flere kjerner muliggjør arbeid av oppgaver *samtidig*, fordi kjernene kan utføre instruksjoner separat. For eksempel, kjører Google Chrome hver fane som en egen prosess, noe som betyr at tolkingen av flere nettsider kan utføres samtidig.

Merk at en CPU med flere kjerner ikke direkte øker ytelsen til et system. For eksempel, er deres nåværende implementasjon av GoL rettet mot en prosess uten støtte for samtidig utførelse av instruksjoner. I et slikt program, vil man derfor få bedre ytelse med en prosessor med høy klokkefrekvens sammenliknet med en fler-kjernet prosessor med lavere klokkefrekvens.

Java programmer kjøres i Java Virtual Machine (JVM). Dvs. når du kjører et Java program på datamaskinen din, vil programmet være relatert til en prosess som representerer JVM. For programmer som har blitt kompilert direkte til maskinkode (som programmer skrevet i C), vil programmet kjøres som en egen prosess. For eksempel, når du starter et program, som Google Chrome, vil en prosess blir opprettet i systemet ditt. Denne prosessen er relatert til ulike data som instruksjoner, heap/stack, prosess-id etc. For en oversikt over de aktive prosessene som kjører i ditt Windows system, trykk på knappene ALT, CTRL, og DELETE. Merk at prosesser blir introdusert mer formelt i faget DATS2500.

Siden Java programmer er 'fanget' inne i JVM prosessen, kan man ikke opprette og håndtere nye prosesser fra Java. For å muliggjøre samtidig programmering i Java, bruker vi en type 'lettvektsprosess'

som heter *tråder* (eng: threads). En tråd er alltid relatert til en, og bare en, prosess og en prosess kan har flere tråder relatert til seg. Forskjellen mellom en tråd og en prosess er at tråder relatert til en gitt prosess deler prosessens data, som instruksjoner og heap segmentene. Slik deling av data er ikke tilgjengelig for prosesser. En tråd har imidlertid egne data som ikke kan deles med andre tråder, som tråd-id, programteller, og stabelsegmentet (stack). Diskuter hvorfor tråder ikke deler stabelsegmentet og konsekvensen av dette.

Tråder og samtidig programmering i Java

Hovedklassen for håndtering av tråder er Thread, som implementerer interfacet Runnable. Informasjon:

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

For å opprette egne tråder implementerer man Runnable interfacet. En tråd kan opprettes og kjøres med følgende kode:

```
public class HelloWorld implements Runnable {

	@Override
	public void run() {
		System.out.println("Hello world from thread " +
				Thread.currentThread().getId());
	}

	public static void main(String[] args) {

		System.out.println("Hello world from thread " +
				Thread.currentThread().getId());

		Thread thread = new Thread(new HelloWorld());
		thread.start();

	}

}
```

Denne konstruksjonen er imidlertid litt tungvinn siden vi må opprette en klasse som implementer Runnable, implementere run metoden, for deretter å opprette et objekt av denne klassen. Heldigvis, tilbyr moderne Java en enklere konstruksjon for å opprette og definere tråder. Følgende kode (i main) produserer identisk utskrift med koden over:

```
System.out.println("Hello world from thread " +
		Thread.currentThread().getId());

Thread thread = new Thread(() -> {
		System.out.println("Hello world from thread " +
		Thread.currentThread().getId());
	});
thread.start();
```

Her bruker vi en Java 'lambda'-konstruksjon (->) til å automatisk opprette en klasse som implementerer et interface med bare en metode. Et objekt av denne klassen blir deretter automatisk opprettet.

Lambda-uttrykk kommer fra funksjonelle språk, som er derivert fra Lambdakalkyle. På den andre siden, imperative språk, som Java, er derivert fra Turing maskiner. Lambda-uttrykk kan sees på som

matematiske funksjoner (med noen begrensinger som vi ikke trenger å forstå). Ut ifra koden angitt over, diskuter hvorfor Lambda-uttrykk i Java ikke er identisk med 'ekte' lambda-uttrykk (dvs. matematiske funksjoner). Tips: har vi sideeffekter?

En fordel med funksjonelle språk er at funksjoner kan håndteres som 'first class citizens'. Det betyr at funksjoner i funksjonelle språk kan returnere funksjoner og ta funksjoner som parametere. For eksempel, er objekter i Java first class citizens, siden man kan returnere objekter (i hvert fall referanser til dem) og ta objekter som parametere. Ved hjelp av elegant bruk av objekt-orientert programmering, tilbyr Java en emulering av funksjoner som first class citizens, der vi håndterer funksjoner via objekter av interface med bare en metode (og den metoden er funksjonen vi er ute etter). Funksjoner i Java kan imidlertid ha sideeffekter og tillater foranderlige (mutable) objekter, noe som ikke er tillatt i funksjonell programmering. Vi skal nå se på en negativ konsekvens av denne fleksibiliteten og hvordan man håndterer dette.

Problemer med foranderlige verdier

Følgende kode oppretter flere tråder som øker verdien til en global foranderlig variabel:

```
import java.util.*;
public class HelloWorld {

	static int a = 0;
	static List<Thread> workers = new ArrayList<Thread>();

	public static void f() {
		a++;
	}

	// opprett Java trådobjekter
	public static void createWorkers() {
		for(int i = 0; i < 10; i++) {
			workers.add(new Thread(() -> {f();}));
		}
	}

	// kjør trådobjektene
	public static void runWorkers() throws InterruptedException {
		for(Thread t : workers) {
			t.start();
		}

		// vent på at alle trådene har kjørt ferdig før vi returnerer
		for(Thread t : workers) {
			t.join();
		}
	}

	public static void main(String[] args) throws InterruptedException {

		createWorkers();
		runWorkers();

		// når en tråd har kjørt en gang, blir den ubrukelig.
		// vi kan derfor slette tråd-objektene
		workers.clear();

		System.out.println(a);
```

```
            createWorkers();
            runWorkers();
            System.out.println(a);

        }

}
```

Den forventede utskriften til programmet er 10 \n 20. Ved å kjøre programmet flere ganger, vil dere imidlertid se at vi kan få forskjellig utskrift, som for eksempel 9 \n 19. Dette er et eksempel på et samtidighetsproblem (eng: concurrent problem), som kan oppstå når delte variabler, som variabelen **a**, manipuleres av samtidig kjørende tråder.

Vi kan se for oss følgende scenario i utførelsen av **a++** kall for to samtidige tråder:

- a er lik 5 før kallene.
- TRÅD 1 henter verdien til a (som er 5), gjør 5+1 (som er 6), og angir at verdien til a skal være 6.
- SAMTIDIG, henter TRÅD 2 verdien til a (som er 5), gjør 5+1 (som er 6), og angir at verdien til a skal være 6.
- Verdien til a er dermed 6, selv om vi har gjort to a++ kall.

For å løse dette problemet, tilbyr Java 'kritiske seksjoner' (eng: critical code sections). Poenget med en kritisk seksjon er å la en, og bare en, tråd få lov til å utføre instruksjoner i seksjonen om gangen. Det vil si, hvis to (eller flere) tråder ønsker å utføre instruksjoner i en kritisk seksjon, vil bare en tråd få lov til å fortsette sin instruksjonssekvens. Den andre tråden må da vente på at seksjonen blir ledig. Hvis vi antar at a++ instruksjonen er angitt i en kritisk seksjon, kan vi nå tenke oss følgende scenario:

- a er lik 5 før kallene.
- TRÅD 1 forespør tilgang til kritisk seksjon. JVM tillater dette. TRÅD 1 henter verdien til a (som er 5), gjør 5+1 (som er 6), og angir at verdien til a skal være 6.
- SAMTIDIG, forespør TRÅD 2 tilgang til kritisk seksjon. JVM tillater ikke dette (TRÅD 1 er allerede i den kritiske seksjonen). TRÅD 2 må da vente på at TRÅD 1 blir ferdig med å utføre instruksjonene i den kritiske seksjonen. Når TRÅD 1 er ferdig, henter TRÅD 2 verdien til a (som er 6), gjør 6+1 (som er 7), og angir at verdien til a skal være 7.
- Verdien til a er dermed 7 etter to 'samtidige' a++ kall.

En kritisk seksjon angis i Java ved **synchronized** nøkkelordet. Gjør om signaturen til funksjonen f til følgende:

```
public static synchronized void f()
```

Instruksjonene i f er nå definert i en kritisk seksjon og JVM vil garantere at bare en tråd kan utføre dens instruksjoner om gangen. Hvordan denne garantien kan tilbys er ikke en del av pensum av dette faget. For interesserte studenter, les om mutex locks, semaphores, og monitors.

Husk at målet med samtidig programmering er å gjøre operasjoner *samtidig*; ikke å tvinge tråder til å vente på hverandre. Kritiske seksjoner burde derfor utelukkende inneholde instruksjoner relatert til delt data (som variabelen a), som ellers hadde ført til samtidighetsproblemer.

GoL ytelsesforbedring med samtidig programmering

Last inn et stort mønster i programmet deres. Eksempler:

http://www.conwaylife.com/wiki/Category:Patterns_with_1000_or_more_cells

http://www.conwaylife.com/wiki/Tlog%28t%29_growth

Opprett en metode som skriver ut tiden det tar å utføre de to oppgavene utført i nextGeneration (telling av naboer og oppdatering av brett). Foreslått signatur:

```
public void nextGenerationPrintPerformance()
```

Eksempel på tidtaking i Java:

```
long start = System.currentTimeMillis();
countNeighbours();
long elapsed = System.currentTimeMillis() - start;
System.out.println("Counting time (ms): " + elapsed);
```

Slik tidtaking skal brukes i deres nye nextGeneration metode for å verifisere om dere faktisk øker ytelsen til programmet.

Opprett to nye metoder:

```
public void nextGenerationConcurrentPrintPerformance() {}
```

```
public void nextGenerationConcurrent() {}
```

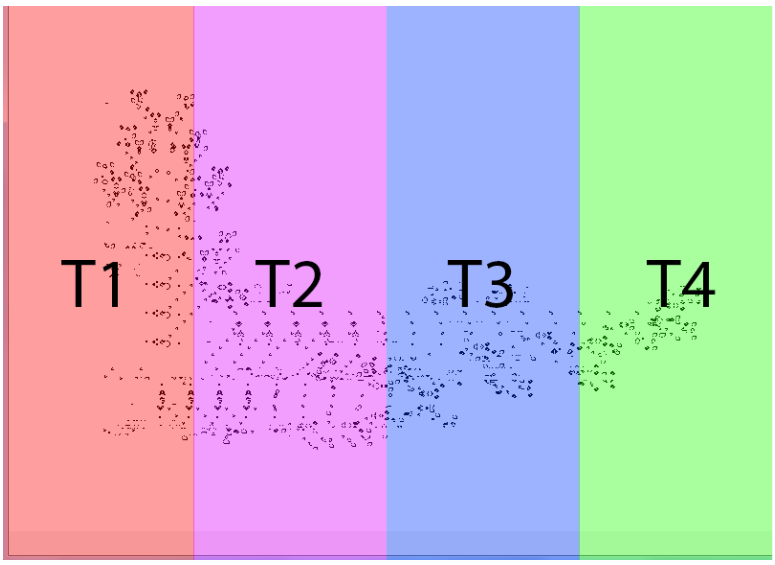
der den første metoden skal brukes for testing og verifisering, mens den siste metoden, som ikke skal skrive noe til konsollen, skal brukes etter at dere har utført denne oppgaven (printing av data til konsollen kan drastisk senke ytelsen til programmet).

De nye metodene skal utføre nextGeneration metoden litt annerledes enn vår originale nextGeneration metode. Brettet skal nå deles opp i n deler der n er antall tråder. Det er naturlig at antall tråder er lik antall kjerner til prosessoren til datamaskinen som kjører programmet. Dette antallet kan hentes via følgende instruksjon:

```
int numWorkers = Runtime.getRuntime().availableProcessors();
```

Denne variabelen kan eventuelt multipliseres med 2 hvis vi antar at CPU'en støtter 'hyper-threading'.

En tråd vil nå få tildelt et angitt område av spillebrettet. Hvert området blir dermed oppdatert av en, og bare en, tråd. Foreslått oppdeling av brettet for fire tråder:



For denne oppdelingen kan det være behjelpelig å huske indekseringsfunksjonen for lav-nivå tabeller (fra DAPE1400).

Ekstraoppgave

Siden et trådobjekt bare kan kalle start en gang, må vi slette alle trådobjekter for hvert kall til nextGeneration. Dette kan føre til økt arbeid for Javas garbage collector, noe som kan øke tiden garbage collector bruker på sine operasjoner når den kjører (kan føre til 'lag'). En forbedring til løsningen over er å bruke en *thread pool*. Les om Java Collections sin støtte for thread pools og bruk funksjonalitet som passer til denne oppgaven. Informasjon:

https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html