IN1010 Gruppetime – Uke 9

Gruppe 7

Dagens tema: Tråder (yuck)

Parallellitet

- Hittil har vi kun skrevet sekvensielle programmer
  - Koden blir utført fra topp til bunn, altså er det ingenting som skjer samtidig, men i en bestemt rekkefølge
- Vi kan få til *parallellitet* i programmene ved å lage oppgaver som kan utføres samtidig (i parallell)
- Det kan være naturlig å ta en oppgave, og stykke den inn i flere deloppgaver som kan løses samtidig
  - Divide and conquer (ish)
- Hvis en gjeng lager taco, så er det naturlig at én kutter grønnsaker, én steker kjøttdeig også videre, og at dette skjer samtidig, ikke én etter én

Tråder

- Hvordan oppnå parallellitet i Java? TRÅDER!!!
- En tråd er en sekvens med instruksjoner i et program
  - Noe kjent: main-metoden er en tråd som kjøres
- Vi kan kjøre flere tråder samtidig for å oppnå parallellitet
- Tråder kan vente, sove og avbrytes
- IN1010: Trådene skal ikke kommunisere med hverandre (men det er en mulighet)

Tråder i Java

- Tråder utfører *oppgaver*, og vi kan definere hva disse oppgavene som skal være
  - Eget interface: *Runnable*
  - Runnable tilbyr én metode, *run()*
- Vi kan lage egne klasser som implementerer runnable, og definere hva trådene skal gjøre i run() metoden
- For å lage en tråd bruker vi klassen *Thread*, og gir objektet en instans av oppgaveklassen vi lagde
- Når vi kaller på metoden *start()* på tråden, så vil den kjøre run() metoden

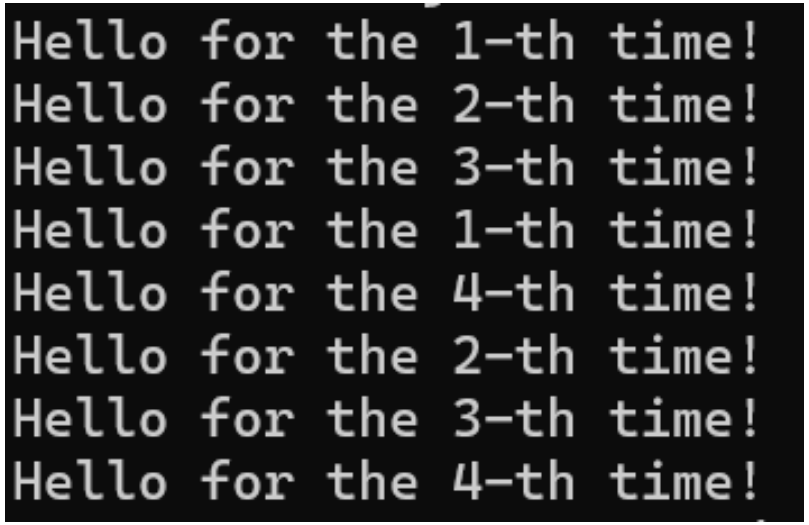
```java
// Eksempel med Runnable

public class MyTask implements Runnable {
    /* Say hello, but think for a while between each time. */
    public void run() {
        for (int i = 1; i < 5000; i++)
            if (i % 1000 == 0)
                System.out.printf("Hello for the %d-th time!\n", i / 1000);
    }
}

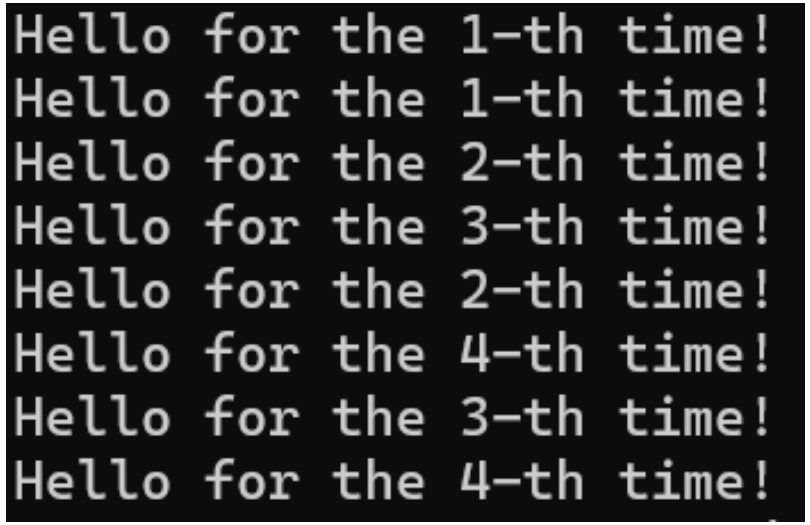
// Eksempel med thread og start fra main-tråden

public Main {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Runnable task = new MyTask();
        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);
        thread1.start(); thread2.start(); // Will make a call to task.run()
    }
}
```

Eksempelutskrift 1



Eksempelutskrift 2



Merk at trådene kan oppføre seg litt annerledes hver gang

Hva skjer hvis to tråder prøver prøver å oppdatere en verdi samtidig?

```java
public class MyTask implements Runnable {
    private final int MAX_COUNT = 10000;
    private int sharedCounter = 0;

    public void run() {
        System.out.println("Starting! Shared counter = " + sharedCounter);
        for (int i = 0; i < MAX_COUNT; i++) {
            sharedCounter = sharedCounter + 1;
        }
        System.out.println("Done! Shared counter = " + sharedCounter);
    }
}

public class Hoved {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Runnable task = new MyTask();
        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);
        thread1.start(); thread2.start(); // Will make a call to task.run()
    }
}
```

Utskrift

```
Done! Shared counter = 10770
Done! Shared counter = 16738
```

Forventet utskrift var at tråden som er ferdig sist skriver ut 2 * MAX_COUNT

Problemet oppstår siden to tråder prøver å oppdatere samme verdi samtidig. Dette kalles en race condition og kan løses med låser

Synkronisering med låser

- For å forhindre slike feil, kan vi sørge for at kun én tråd prøver å oppdatere en felles verdi av gangen
  - Andre tråder må vente på sin tur
  - Det kalles en *kritisk region* når kun én tråd kan aksessere en verdi om gangen
- Vi bruker objektet ReentrantLock, som implementerer grensesnittet Lock
  - Bl.a to metoder, lock() og unlock()

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class MyTask implements Runnable {
    private final Lock lock = new ReentrantLock();
    private final int MAX_COUNT = 10000; private int sharedCounter = 0;


    public void run() {
        System.out.println("Starting! Shared counter = " + getSharedCounter());
        for (int i = 0; i < MAX_COUNT; i++) {
            lock.lock();
            try {
                sharedCounter = sharedCounter + 1;
            } finally {
                lock.unlock();
            }
        }
        System.out.println("Done! Shared counter = " + getSharedCounter());
    }


    private int getSharedCounter() {
        lock.lock();
        try {
            return sharedCounter;
        } finally {
            lock.unlock();
        }
    }

}
```

```
Done! Shared counter = 15736
Done! Shared counter = 20000
```

OBS: Det er ikke ofte vi ønsker å bruke Lock inne i en oppgave

Foretrukket å lage en monitor som inneholder de felles dataene som trådene kan aksessere

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;


class SharedCounter {
    private int counter = 0;
    private final Lock lock = new ReentrantLock();
    private final Condition counterReachedFive = lock.newCondition();


    void incrementCounter() {
        lock.lock();
        try {
            counter++;
            System.out.println("Counter incremented to: " + counter);
            if (counter == 5) {
                counterReachedFive.signalAll();
            }
        } finally {
            lock.unlock();
        }
    }


    void waitForCounterToReachFive() throws InterruptedException {
        lock.lock();
        try {
            while(counter < 5) {
                System.out.println("Waiting for the counter to reach 5...");
                counterReachedFive.await();
            }
            System.out.println("The counter has reached 5!");
        } finally {
            lock.unlock();
        }
    }
}
```

```java
class IncrementCounterTask implements Runnable {
    private final SharedCounter sharedCounter;
    IncrementCounterTask(SharedCounter sharedCounter) {
        this.sharedCounter = sharedCounter;
    }


    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            sharedCounter.incrementCounter();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("The counting thread was interrupted. Exits.");
                System.exit(1);
            }
        }
    }
}
```

```java
class WaitForCounterTask implements Runnable {
    private final SharedCounter sharedCounter;
    WaitForCounterTask(SharedCounter sharedCounter) {
        this.sharedCounter = sharedCounter;
    }


    @Override
    public void run() {
        try {
            sharedCounter.waitForCounterToReachFive();
        } catch (InterruptedException e) {
            System.out.println("The waiting thread was interrupted. Exits.");
            System.exit(1);
        }
    }
}
```

Bombespill Livekode