UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Prøveeksamen i: INF2440— Effektiv parallellprogrammering
Prøveeksamensdag: 1. juni 2016
Tidspunkter: 09.00 – 16.00
Oppgavesettet er på: 4 sider + 2 sider vedlegg
Vedlegg: I) Skisse av Modell2-koden, II) En sorteringsalgoritme
Tillatte hjelpemidler: Alle trykte og skrevne notater, utskrifter, bøker ol.

- Kontroller at oppgavesettet er komplett, og les nøye gjennom oppgavene før du løser dem. Poengangivelsen øverst i hver oppgave angir maksimalt antall poeng.
- Du kan legge dine egne forutsetninger til grunn og gjøre rimelige antagelser, så lenge de ikke bryter med oppgavens "ånd". Gjør i så fall rede for disse forutsetningene og antagelsene.
- Til eksamen skal svarene skrives på gjennomslagspapir. Da må du huske å skrive hardt nok til at besvarelsen blir mulig å lese på alle gjennomslagsarkene, og ikke legge andre deler av eksamensoppgaven under når du skriver.
- Til eksamen skal du selv beholde underste arket etter levering av de to øverste til eksamensinspektøren. Nummerér sidene, og husk å skrive kandidatnummeret ditt på besvarelsen.

I vedlegg I) finner du en litt forenklet versjon av Modell2-koden som ble nyttet i kurset (en ytre klasse med main-tråden og en indre klasse hvor objekter av denne blir egne tråder). I den skissen er det markert med store bokstaver ulike områder av denne koden som det blir referert til i oppgavene.

Oppgave 1 (10 poeng)

Forklar hva som skjer ved en synkronisering:

a) Når to tråder synkroniserer på samme synkroniseringsobjekt samtidig .
b) Når to tråder synkroniserer på hvert sitt synkroniseringsobjekt samtidig.

Oppgave 2 (25 poeng)

a) Anta at du har tre tråder som prøver å oppdatere to globale variable x1 og x2, deklarert i A-området Modell2-koden (vedlagt), begge initielt =0 i, slik:

```
public void run( ) {
     for (int i =0; i < 2; i++){
         x1++;
         x2++;
         System.out.println("  - tråd:"+ind+", x1:"+x1+", x2:"+x2 );
    }
 System.out.println(" Tråd:"+ind+" terminerer"+", x1:"+x1+", x2:"+x2);
} // end run:
```

Like etter at man i metoden utfoer(..) har sagt join() på de tre trådene er følgende utskrifttssetning:

```
System.out.println("Main-tråden  TERMINERER"+", x1:"+x1+", x2:"+x2 );
```

Hver gang man kjører dette programmet, får man nesten alltid veldig forskjellige utskrifter. Her er to:

Utskrift U1:

```
- tråd:2, x1:2, x2:3
- tråd:1, x1:2, x2:3
- tråd:1, x1:4, x2:5
- tråd:0, x1:2, x2:3
Tråd:1 terminerer, x1:4, x2:5
- tråd:2, x1:3, x2:4
Tråd:2 terminerer, x1:5, x2:6
- tråd:0, x1:5, x2:6
Tråd:0 terminerer, x1:5, x2:6
 Main-tråden  TERMINERER, x1:6, x2:6
```

Utskrift U2:

```
- tråd:1, x1:2, x2:2
- tråd:2, x1:3, x2:3
- tråd:2, x1:5, x2:5
- tråd:0, x1:1, x2:1
Tråd:2 terminerer, x1:5, x2:5
- tråd:1, x1:4, x2:4
Tråd:1 terminerer, x1:6, x2:6
- tråd:0, x1:6, x2:6
Tråd:0 terminerer, x1:6, x2:6
Main-tråden  TERMINERER, x1:6, x2:6
```

Oppgave:
Forklar og begrunn kort svarene på følgende spørsmål:

a) Hvorfor ser f.eks tråd:2 i første linje ulike verdier og x1 < x2 i U1?
b) Hvorfor ser ingen av trådene x1 ==6, mens main-tråden ser x1== 6 til sist i U1.
c) Hvorfor ser ingen av trådene i U1 verdien 1?
d) Vil alltid main-tråden skrive ut x1 ogx2 med verdien 6?
e) Hvorfor er utskriftene ulike (hvorfor 'aldri samme resultat' på gjentatte kjøringer)?

Oppgave 3 (25 poeng)

Lag et parallelt program med to tråder og to **`CyclicBarrier`** hvor de to trådene vekselsvis sender en nummerert melding til den andre tråden 20 ganger (f.eks. «Tråd0 sier: Hei nr 14») . Meldingen skrives ut med **`System.out.println(…)`**. Når den ene tråden skriver, skal den andre tråden vente på en av de to **`CyclicBarrier-`**ene og motsatt. ***Bare*** skriv run() – metoden og initieringen av de to **`CyclicBarrier-`**ene.
Kunne denne oppgaven vært løst med bare én **`CyclicBarrier ?`**

Oppgave 4 (15 poeng)

Anta at du har to tråder som aksesserer en felles variabel, deklarert i felt A slik: **`int i =0,`** og at hver av trådene har deklartert i det felles området A for alle trådene:

```
ReentrantLock lock = new ReentrantLock();
```

Tråd0 utfører følgende kode:
```
lock.lock();
try { i++;
 } finally { lock.unlock();}
 i++;
```

før den terminerer, mens

Tråd1 utfører følgende kode:
```
i--;
lock.lock();
try { i--;
} finally { lock.unlock();}
```

før den terminerer,

Hvilke mulige verdier kan **`i`** ha etter at begge trådene er startet, men ***før*** maintråden sier join() på trådene. Der er følgende utskriftssetning:

```
System.out.println(" Main-tråden  OBERVERER i:"+i );
```

Tegn diagrammer med tidsakse som viser ***hvordan*** de to minste verdiene du hevder kan bli verdien av **`i`** i Main-tråden kan fremkomme (når leser og skriver Tråd0 og Tråd1 hvilke verdier?).

Oppgave5 (30 poeng)

Anta at du har deklarert en array: **`int tallene[] = new int [n];`** og du kan anta at elementene i **`tallene[]`**er sortert ***stigende***.

Du skal nå skrive først en sekvensiell og så en parallell metode (anta at du har k kjerner) som snur innholdet av **`tallene[]`**slik at tallene blir sortert ***synkende***. Skriv heller ikke her hele programmet, men bare de metodene og evt. data som du trenger i A og B området i vedlegget + hva innholdet av run()-metoden i trådene er.

Oppgave 6 (30 poeng)

Du har gitt en metode for å sortere (i Appendix II er en slik algoritme gitt basert på innstikk-sortering, men vi kunne like godt ha brukt radix eller quicksort i steden):

```
public static void sort (int a[], int left, int right) {..}
```

Du skal nå skrive en ny metode **`sort2`**, basert på denne som sorterer to arrayer med følgende forståelse:

```
public static void sort2 (int a[], int [] b, int left, int right) {..}
```

Du sorterer innholdet av arrayen **`a[]`** som i den første metoden "**`sort`**" , men med det tillegget at elementene i den andre arrayen **`b[]`** flyttes helt tilsvarende som elementene i **`a[]`** flyttes. Eksempel: hvis elementet **`a[0]`** ble flyttet til **`a[23]`** etter sortering så vil også det opprinnelige elementet **`b[0]`** nå stå på plass **`b[23]`**. Dette oppnår du ved at hver gang du flytter **`a[]`** –

elementene, flytter du **b[]** elementene med samme indeks helt tilsvarende.
Skriv en sekvensiell metode "**sort2**" som løser denne oppgaven. Denne vil du så bruke for å løse oppgave 7.

Oppgave 7 (80 poeng)

Du har en foreleser som tror at han har en genial idé til å lage en raskere innstikksortering-metode kalt **medianSort**. Problemet med innstikksortering, tenkte han, er at det er en del store elementer tidlig (med lave indekser) i arrayen som må skyves *langt* avgårde mot enden av arrayen, og motsatt, mange små verdier alt for langt ut i array-en som sorteres mot begynnelsen. Disse må flyttes (byttes med hverandre) før vi gjør instikksortering til sist, som da vil gå svært mye raskere.

Den sekvensielle algoritmen for dette er slik:

a) Vi starter med å dele opp arrayen i en rekke **m** mindre områder (f.eks i **m** = $n/5$ deler) og du kan for enkelthets syld anta at n er delbar med m uten rest (n%m ==0).
Vi sorterer så hver og en av dem med "**sort**" fra Appendix II samtidig som vi noterer verdien av det midterste elementet i hver av de sorterte delene i:
int[] median = new int[m]. Samtidig har du laget en annen array **int[] index = new int [m]** som initielt inneholder **0,1,2,..,m**-1.

b) Sorter nå arrayene **median** og **index** med **sort2** fra oppgave 6. Du kan her anta at du har løst oppgave 6 om å lage en sortering for to arrayer slik at når du sorterer på den ene arrayen (her: **median**) og har med en annen array **indeks** (initielt: 01,2,…, $m-1$) som parameter, så deltar elementene i **indeks** i de samme ombyttingene som elementene i **median**. Når da median-arrayen er sortert, vil f.eks **indeks[0]** si hvilken av delene av **a[]** som hadde minst median, **indeks[1**] hvilken som hadde nest minst median, … osv.

c) Vi flytter om små-områdene i **a[]** over i en like lang array **b[]** slik at det området som har minst median kommer først i **b[]**, nest minst kommer som område nr. 2, osv

d) Når vi er ferdige med alle disse omflyttingene kopieres **b[]** tilbake til **a[]** og

e) det gjøres ett kall på **sort** fra AppendixII av hele arrayen **a[]**.

Oppgaver:

7.1 Programmer denne algoritmen sekvensielt.

7.2 Lag en parallell versjon av dette sekvensielle programmet med Modell2 koden, bare parallelliser stegene a) , c) og d) - ikke stegene b) eller e) som begge er kall på innstikksortering. Skriv bare de tilleggene du trenger og forklar hvor du plasserer dem i Modell2-koden,

Appendix I – Modell2 kodeskisse:

```
import java.util.concurrent.*;
class Problem {
    //  felles data og metoder A
     public static void main(String [] args) {
          Problem p = new Problem();
          p.utfoer(8);
     }
     void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();
           try{
             for (int i =0; i< antT; i++) t[i].join();
          } catch(Exception e) {}
     }

     class Arbeider implements Runnable {
         int ind; // lokale data og metoder B

            Arbeider (int in) {ind = in;}
          public void run( ) {
                 // kalles når tråden er startet
          } // end run
     } // end indre klasse Arbeider

} // end class Problem
```

Appendix II

```
  /** sorts a [left .. right]  by Insertion sort alg. Sub-alg for short segments */
 public static void sort (int a[], int left, int right) {
        int i,k;
        int t;

        for (k = left ; k < right; k++) {
              if (a[k] > a[k+1]){
                  t = a[k+1];
                  i = k;

                 while (i >= left && a[i] > t) {
                       a[i+1] = a[i];
                       i--;
                 }
                 a[i+1] = t;
           }
       }
   } // end sort
```