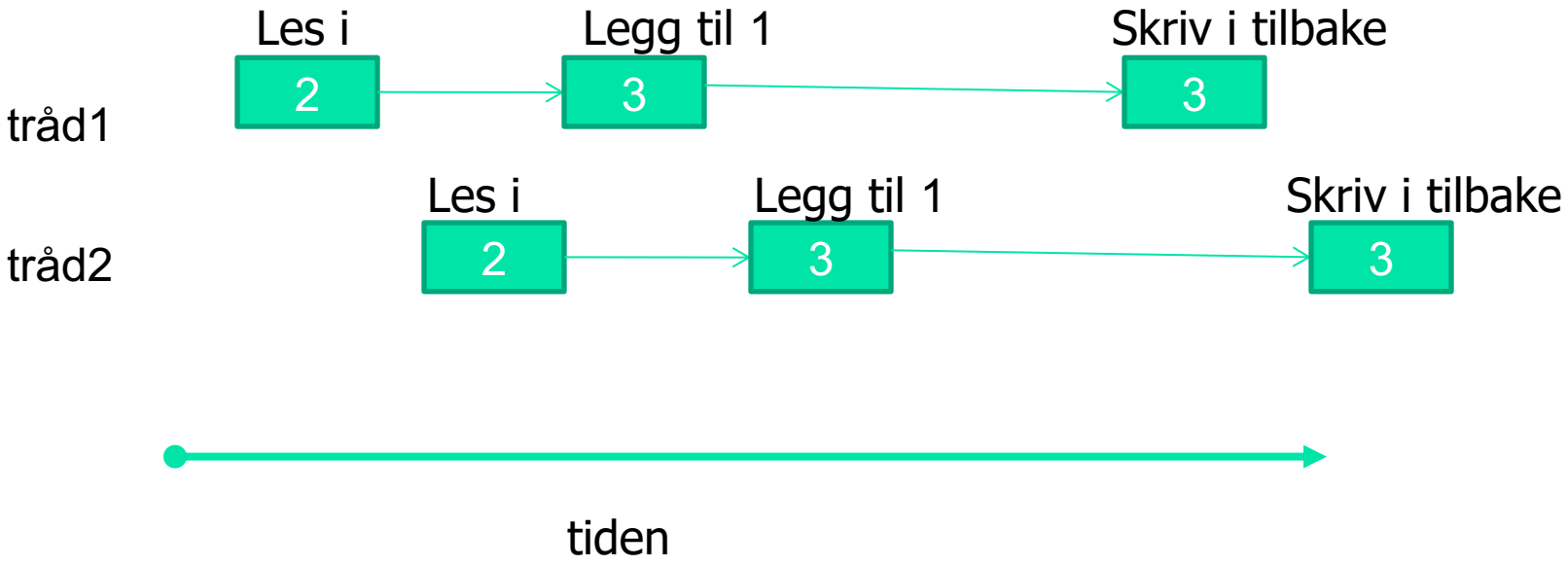
Example: Concurrent update of a variable Let's try! (Spoiler: we will fail!)

```
import java.util.concurrent.*;
class Problem { int i ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () {
                    int j, k;
                    j=10;
                    for (k=0; k< j; k++)  {
                        new Thread(new Arbeider()).start();
                 }

          class Arbeider implements Runnable {
               int i,lokalData; // dette er lokale data for hver tråd
                  public void run() { // denne kalles når tråden er startet
                        i++;
                  }
           } // end indre klasse Arbeider
} // end class Problem
```

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

- Samtidig oppdatering - flere tråder sier gjentatte ganger: **i++ ;** der i er en felles int.
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Anta i =2, og to tråder gjør i++
  - Vi kan få svaret 3 eller 4 (skulle fått 4!)
  - Dette skjer i praksis !



Test på i++; parallell

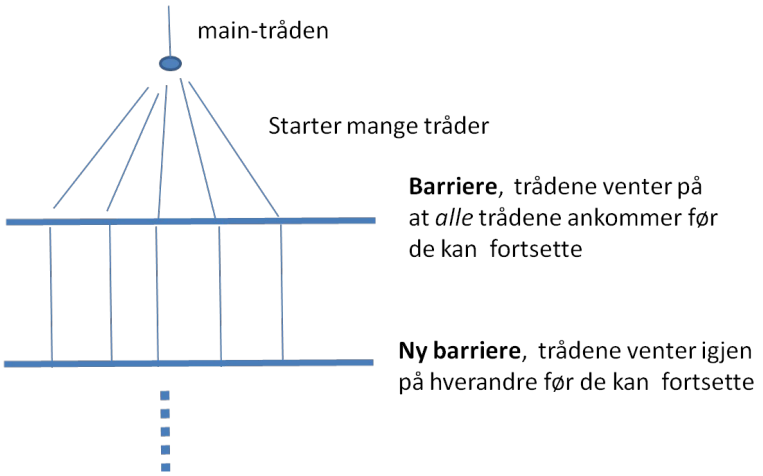
- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

```
for (int j =0; j< 100000; j++) {
        i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

Antall tråder n		**1**	**2**	**20**	**200**	**2000**
Svar	1.gang	100 000	200000	1290279	16940111	170127199
	2.gang	100 000	159234	1706068	16459210	164954894
Tap	1.gang	0 %	0%	35,5%	15,3%	14,9%
	2. gang	0%	20,4%	14,6%	17,7%	17,5%

Kommende program bruker CyclicBarrier. Hva gjør den?



- Man lager først ett, felles objekt `b` av klassen CyclicBarrier med et tall: `ant` til konstruktøren = det antall tråder den skal køe opp før alle trådene slippes fri 'samtidig'.
- Tråder (også main-tråden) som vil køe opp på en CyclicBarrier sier await() på den.
- De `ant-1` første trådene som sier await(), blir lagt i en kø.
- Når tråd nummer `ant` sier await() på `b`, blir alle trådene sluppet ut av køen 'samtidig' og fortsetter i sin kode.
- Det sykliske barriere objektet `b` er da med en gang klar til å være kø for nye, `ant` stk. tråder.

Praktisk: skal nå se på programmet som laget tabellen

```
import java.util.*;
import easyIO.*;
import java.util.concurrent.*;
/** Viser at manglende synkronisering på ett felles objekt gir feil – bare loesning 1) er riktig'*/

public class Parallell {
        int tall;                               // Sum av at 'antTraader' traader teller opp denne
        CyclicBarrier b ;                       // sikrer at alle er ferdige naar vi tar tid og sum
        int antTraader, antGanger ,svar; // Etter summering: riktig svar er:antTraader*antGanger

        // det kommer I alt 4 forsøk på å øke i, bare en av dem er riktig
        //synchronized void inkrTall(){ tall++;}       // 1) –OK fordi synkroniserer på ett objekt (p)
        void inkrTall() { tall++;}                      // 2) - feil

        public static void main (String [] args) {
                if (args.length < 2) {
                        System.out.println("bruk >java Parallell <antTraader> <n= antGanger>");
                }else{  int antKjerner = Runtime.getRuntime().availableProcessors();
                        System.out.println("Maskinen har "+ antKjerner + " prosessorkjerner.");
                         Parallell p =  new Parallell();
                          p.antTraader = Integer.parseInt(args[0]);
                         p.antGanger = Integer.parseInt(args[1]);
                         p.utfor();
        } } // end main
```

```
void utskrift (double tid) {
        svar = antGanger*antTraader;
        System.out.println("Tid "+antGanger+" kall * "+ antTraader+" Traader ="+
                                                    Format.align(tid,9,1)+ " millisek,");
         System.out.println(" sum:"+ tall +", tap:"+ (svar -tall)+" = "+
          Format.align( ((svar - tall)*100.0 /svar),12,6)+"%");

} // end utskrift

 void  utfor () {   b = new CyclicBarrier(antTraader+1);       //+1, også main
                    long t = System.nanoTime();                 // start klokke

                    for (int j = 0; j< antTraader; j++) {
                          new Thread(new Para(j)).start();
                     }

                   try{   // main thread venter
                          b.await();
                    } catch (Exception e) {return;}
                    double tid = (System.nanoTime()-t)/1000000.0;
                    utskrift(tid);
   } // utfor
```

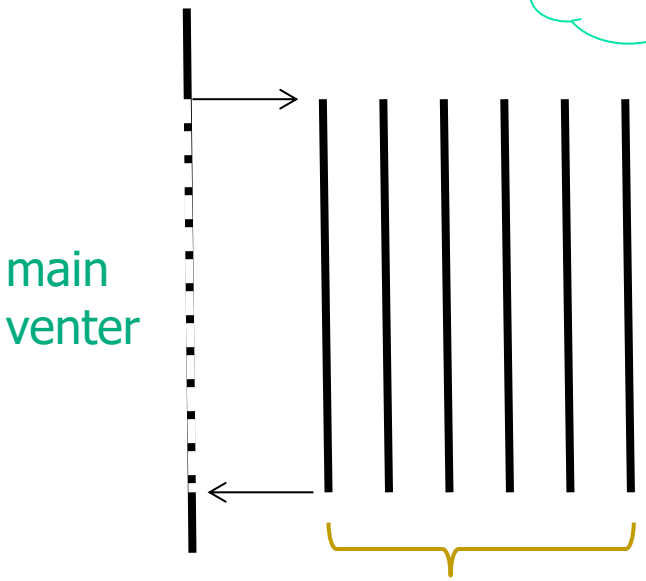
```
class Para implements Runnable{
        int ind;
        Para(int ind) { this.ind =ind;}

        public void run() {
            for (int j = 0; j< antGanger; j++) {
                inkrTall();
              }
              try {  // wait on all other threads + main
                  b.await();
              } catch (Exception e) {return;}
          } // end run

        //    void inkrTall() { tall++;}                    // 3) Feil - usynkronisert
        //    synchronized void inkrTall(){ tall++;}  // 4) Feil – kallene synkroniserer på
                                                                //      hvert sitt objekt
    } // end class Para
} // END class Parallell
```

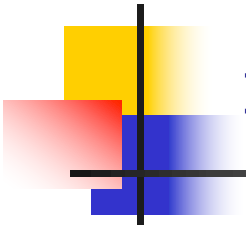
Husk: Vanligste oppsett av main-tråden + k tråder

main, lager k nye tråder



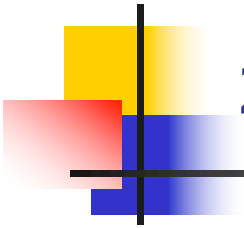
k tråder, leser og skriver i egne og i felles data og løser problemet

Hver av trådene (main + k nye) er sekvensielle programmer.
Problemet er at de samtidig *ikke kan skrive* på felles data



1) Avslutning med en CyclicBarrier

- En CyclicBarrier (cb= new CyclicBarrier (n+1))
  - Er tenkt som et ventested, en bom/grind for et antall (i dette tilfellet for n+1) tråder - de n 'nye' trådene + main. Alle må vente når de sier sier cb.await() til sistemann ankommer køen, og **da** kan alle fortsette.
  - Trådene kan da være ferdige med en beregning kan selv avslutte med å bli ferdige med sin run() -kode. Main-tråden forsetter, og vet at de andre trådene er ferdige. Main-tråden kan da bruke resultatene fra trådene.
  - Den sykliske barrieren cb er da strakt klar til å køe nye n tråder som sier cb.await() , .. osv
  - cb.await() sies inne i en try-catch blokk



2) Avslutning med en Semaphore

- En Semaphore (sf = new Semaphore(-n+1))
  - Administrerer (i dette tilfellet) –n+1 stk. **tillatelser**.
  - To sentrale primitiver:
    - sf.acquire() – ber om **en** tillatelse. Antall tillatelser i sf blir da 1 mindre hvis antallet er >0. Hvis det ikke er noen ledig tillatelse, må tråden vente i en kø (inne i en try-catch blokk)
    - sf.release() – gir **én** tillatelse tilbake til semaforen sf. Ikke try-catch blokk (Den tillatelsen som gis tilbake behøver ikke vært 'fått' ved hjelp av aquire() ; den er bare et tall).
  - Avlutning med Semaphore sf:
    - Maintråden sier sf.acquire() – og må vente på at det er minst en tillatelse i sf.
    - Alle de n nye trådene sier sf.release() når de terminerer, og når den siste sier sf.release() blir det 1 tillatelse ledig og main fortsetter.
    - Ikke syklisk.

3) Avslutning med join() - enklest

- Logikken er her at i den rutinen hvor alle trådene lages, legges de også inn i en array. Main-tråden legger seg til å vente på den tråden som den har peker til skal terminere selv.
  Venter på alle trådene etter tur at de terminerer:

```
 // main –tråden i konstruktøren
Thread [] t = new Thread[n];
for (int i = 0; i < n; i++) {
   t[i] = new Thread (new Arbeider(..));
   t[i].start();
 }
  ……….
  // main vil vente her til trådene er ferdige
for(int i = 0; i < n; i++) {
       try{ t[i].join();
        }catch (Exception e){return;};
}     ……..……..
```

II) Mange ulike synkroniserings primitiver
Vi skal bare lære noen få !

- java.util.concurrent

Classes

AbstractExecutorService
ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
CountDownLatch
CyclicBarrier
DelayQueue
Exchanger
ExecutorCompletionService
ecutor
eadPoolExecutor
ThreadPoolExecutor.AbortPolicy
ThreadPoolExecutor.CallerRunsPolicy
ThreadPoolExecutor.DiscardOldestPolicy
ThreadPoolExecutor.DiscardPolicy

Semaphore
SynchronousQueue
ThreadLocalRandom
ThrExecutors
ForkJoinPool
ForkJoinTask
ForkJoinWorkerThread
FutureTask
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue
Phaser
PriorityBlockingQueue
RecursiveAction
RecursiveTask
ScheduledThreadPoolEx

Interfaces

BlockingDeque
BlockingQueue
Callable
CompletionService
ConcurrentMap
ConcurrentNavigableMap
Delayed
Executor
ExecutorService
ForkJoinPool.ForkJoinWorkerThreadFacto
ry
ForkJoinPool.ManagedBlocker
Future
RejectedExecutionHandler
RunnableFuture
RunnableScheduledFuture
ScheduledExecutorService
ScheduledFuture
ThreadFactory
TransferQueue

java.util.concurrent.atomic

De har samme virkning (semantikk) som volatile variable (forklares senere), men kan gjøre mer sammensatte operasjoner. Mye raskere enn sychronized methods.

Classes

AtomicBoolean
AtomicInteger
AtomicIntegerArray
AtomicIntegerFieldUpdater
AtomicLong
AtomicLongArray
AtomicLongFieldUpdater
AtomicMarkableReference
AtomicReference
AtomicReferenceArray
AtomicReferenceFieldUpdater
AtomicStampedReference

Eksempel på operasjoner i **AtomicIntegerArray:**

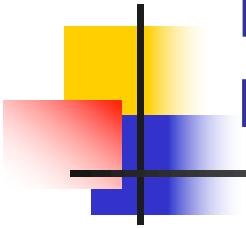
int	**get**(int i) Gets the current value at position i.
int	**getAndAdd**(int i, int delta) Atomically adds the given value to the element at index i.
int	**getAndDecrement**(int i) Atomically decrements by one the element at index
void	**set**(int i, int newValue) Sets the element at position i to the given value.

Vi skal bare lære ett fåtall av dette

- Her er de vi skal konsentrere oss om:
  - new Thread – join()
  - synchronized method
  - Semaphore – aquire() og release()
  - CyclicBarrier – await()
  - ExecutorService pool = Executors.newFixedThreadPool(k);
    med Futures - forklares senere
  - AtomicIntegerArray – get(), set(), getAndAdd(),..
  - ReentrantLock (i pakken: **java.util.concurrent.locks**)
  - volatile variable - forklares senere

- Alle de synkroniseringer vi trenger, kan gjøres med disse!
- De fleste andre har sine måter å gjøre det på, men man har neppe tid til å lære seg alle.
- Bedre å bli flink i et lite og tilstrekkelig sett av synkroniseringsprimitiver, enn halvgod i de fleste.

Kan det gå galt når to tråder samtidig skriver i ulike plasser i en array?



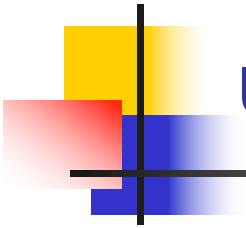
- Et problemet kunne være at når en av tråden lester opp et element i a[i] (int = 4 byte), så er cache-linja 64 byte, så den får med seg flere elementer før og etter a[i].
- Disse 'andre' elementene er det andre tråder som skriver på.
- Vi skriver et testprogram (ParaArray) hvor 10 tråder med indeks : 0,1,2,..,9 som øker hvert sitt element i en array tall[index] 100 000 ganger.

Skriving på nærliggende elementer i en array.

```
class ParaArray{
                    int  []tall;
                    CyclicBarrier b ;
                    int antTraader, antGanger ;
....
class Para implements Runnable{
    int indeks;
   Para(int i) { indeks =i;}
       public void run() {
                    for (int j = 0; j< antGanger; j++) {
                            oekTall(indeks);
                     }
                    try {  // wait on all other threads + main
                                    b.await();
                    } catch (Exception e) {return;}
       } // end run
        void oekTall(int i) { tall[i]++; }
  }
} // end ParaArray
```

- Cache-linja er nå 64 byte (og en int er 4 byte)
- Går det greit med at flere tråder (indeks=0,1,…,k-1) skriver på a[tråd.indeks] mange ganger i parallell?
- Tester: Vi lageret program som gjør det :

```
>java ParaArray 10 100000000
Maskinen har 8 prosessorkjerner.
Tid 100000000 kall * 10 Traader =
0.032600 sek,
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
```



Ukeoppgave L02

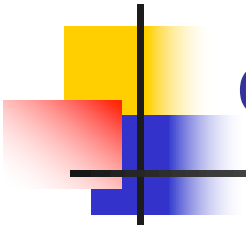
- Find the largest number in a large array

Oblig plan 2024

IN3030/IN4330 Oblig plan 2024:

O1	L2	2 weeks	25/1 – 7/2
O2	L4	2 weeks	8/2 – 21/2
O3	L7	4 weeks	22/2 – 20/3
[Easter Break]			
O4	L11	3 weeks	4/4 – 24/4
O5	L14	1 week	25/4 – 2/5

Written exam 27/5 – 2024



Oblig plan 2024

Lectures L1 – L10 are given in calendar weeks 3 thru 12.

(Add 2 to the lecture number to get the week number.)

No lecture 28/3 (Easter break).

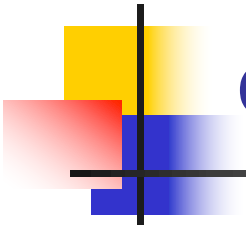
Lectures after Easter and before Kristi himmelfartsdag start with L11 in week 14.

(After Easter, add 4 to the lecture number to get the week number.)

Lectures after Kristi himmelfartsdag start with L17 in week 20.

Obligs are delivered in Devilry no later than 23:59:00 on the deadline date.

NOTE: the deadline is ONE MINUTE before midnight!



Oblig 1

- Oblig 1 presenteres