Suggested solutions IN3030 Spring 2019 Written Exam

Eric Jul

Question 1.1 When all the threads created in the main() method have terminated AND any thread created by any other thread in the program has terminated AND the main() method also has terminated.

Question 1.2

- Java threads execute in parallel: Every core runs a thread until it either terminates or has used up a time slice, whereafter another thread (possibly the same thread) is run on the core - and so on.
- Java threads appear to execute in parallel independently of one another even on single-core machines because the threads are time multiplexed.
- Java threads can execute in parallel independently of one another on multi-core machines.

Question 1.3

- Yes, furthermore, if another thread tries to call another syncrhonized method in the same object, the Java system ensures that only one thread is executing within any of the synchronized methods in the object.
- Yes, if more than one thread calls the method, the Java system ensures that only one thread is executing within the object.

Question 1.4 Note: the question was, due to an error in Inspera, in bokmål.

CyclicBarrier er billigere å bruke enn synchronized nøkkelordet.

Question 3.1

Attached program contains a possible sequential solution.

Question 3.2

Attached program contains a possible parallel solution. It works by splitting the primes into parts, one for each thread, with approximately the same number of primes in each part. Then each thread finds the twin prime pairs presents in its part – with due care to finding twin prime pairs that span across parts – this is done by adjusting the parts to start and end at a prime – and by including the last prime of one part as the first prime of the next part. Each thread finds a list of prime deserts of monotonically increasing length within its part.

Question 4.1

Insertsort can be parallelized in more than one way.

Method using parallel passes.

One way is to use M threads where M is 1-2 x the number of cores.

Insertion sort can be described as a number of passes, the k'th pass inserts the k'th element into the 0 .. k-1 other elements utilizing the fact that they are in non-decreasing order.

Each thread executes one pass of the insertion sort. Because each pass proceeds from the k'th element and downwards toward the zeroth element, we need to make sure that when the k'th pass has reached element i (where 0 < i <= k) then the (k+1)'th pass must not have gotten further down than i+1. We can ensure this by having a semaphore for EACH element in the array. A thread doing pass k then locks every element before accessing it – and releases when it moves to the next lower element. If we ensure that it gets the lock for the k'th element before any pass with a pass number higher than k, then the passes with a higher number cannot "overtake" the thread doing the k'th pass.

The problem with this is that it requires one synchronization per element access, which is very inefficient. Instead, we can let the k'th thread lock a long segment of the array. If, for example, the segments are of length 20 then the number of synchronizations is cut down by a factor of 20 albeit it is still O(N*N). However, if the segment size is, for example, N/M/20, then the number of synchronizations is O(N*M) and because M is much smaller than N, it is essentially O(N).

If M is, say, 2x the number of cores, it is likely that there will be a thread that is able to execute – the faster threads will be delayed and the slower given a chance to barge ahead.

Each thread enters a new pass by locking an entry region.

In the entry region, it grabs the next pass number, grabs the lock of the first segment, and thereafter releasing the lock for the entry region. This ensures that threads are started in the order of the pass numbers. We identify the pass numbers by the number of elements that must be treated in the pass, i.e., the first pass is k because it need to treat elements from 0 to k.

For locking the entry region and each segment, we use a semaphore initialized to one.

Both of these algorithms are faithful to the bubblesort algorithm – and both can achieve speedup for the parallel version.

Question 4.2

The attached program contains a parallel version of insertsort using the parallelization described in 4.1.

Question 5.1 300,000 km/s

Question 5.2 300 mm/ns

Question 5.3

The speed of light in vacuum is constant at 300,000 km/s, which corresponds to 30 cm/ns.

The slow speed of light means high latency for access to data over a distance and this was one reason that caches were developed (the other being the availability of high-speed-but-expensive memory).

As CPUs got faster, the access latency even to cache got relatively larger resulting in multiple level caches as to mitigate this delay.

The latency problem is compounded by the slower speed of propagation in copper and fiber optics albeit only by a constant factor of about 2/3.

In summary, the slow speed of light lead to the development of multiple level caches as to hide the latency caused by the speed of light.

Attached programs:

Question 2.1

```
import java.util.concurrent.*;
class CyclicBarrierJoinP {
   static CyclicBarrier cb;

   public static void done() {
      try {
         cb.await();
      } catch (Exception e) { return; }
   }

   public static void main(String[] args) {
      int numberofthreads = 10;

      Thread[] t = new Thread[numberofthreads];
      cb = new CyclicBarrier(numberofthreads+1);

      for (int j = 0; j < numberofthreads; j++) {
         (t[j] = new Thread( new ExThread() )).start();
      }
```

```
//    try {
//       for (int k = 0; k < numberofthreads; k++) t[k].join();
//    } catch (Exception e) { return; }

      done();
   }

   static class ExThread implements Runnable {
      public void run() {
         try {
            TimeUnit.SECONDS.sleep(10);
         } catch (Exception e) { return;};
         done();
      }
   }

}
```

Question 3.1

```
import java.util.*;


///-------------------------------------------------------
//
//     Original File: EratosthenesSil.java
//     implements bit-array (Boolean) for prime numbers
//     written by:  Arne Maus , Univ of Oslo, 2013, 2015
//     Derived for Twin Primes, Eric Jul, Univ. of Oslo 2019
//
//-------------------------------------------------------


/**
* Implements the bitArray of length 'bitLen' [0..bitLen ]
*   1 - true (is prime number)
*   0 - false
*  can be used up to 2 G Bits (integer range)
*/
```

```
public class TwinPrimePair
{
   byte [] bitArr ;
   int bitLen;
   final  int [] bitMask ={1,2,4,8,16,32,64,128};
   final  int [] bitMask2 ={255-1,255-2,255-4,255-8,255-16,255-32,255-
64,255-128};



   TwinPrimePair (int max) {
      bitLen = max;
      bitArr = new byte [(bitLen/16)+1];
      setAllPrime();
      generatePrimesByEratosthenes();
   } // end konstruktor TwinPrimePair


   public static void main(String args[]) {
      ArrayList<int[]> pairs = new ArrayList<int[]>();
      TwinPrimePair sil;
          int num = Integer.parseInt(args[0]);
          int [] pair;


          sil = new TwinPrimePair(num);
          int nextSP, nextEP, lastP;
          lastP = sil.lastPrime();
          nextSP = 3;
          nextEP = sil.nextPrime(nextSP);
          while (nextEP <= lastP) {
             if ((nextEP-nextSP) == 2) {
                pair = new int[2];
                pair[0] = nextSP;
```

```
                pair[1] = nextEP;
                pairs.add(pair);
            }
            nextSP = nextEP;
            nextEP = sil.nextPrime(nextSP);
         };
         for (int i = 0; i < pairs.size(); i++) {
            System.out.println(" "+i+": ["+pairs.get(i)[0]+",
"+pairs.get(i)[1]+"]");
         }
      }

       void setAllPrime() {
            for (int i = 0; i < bitArr.length; i++) {
             bitArr[i] = -1 ;     // alt    ( byte)255;
          }
        }

      void setNotPrime(int i)
      { bitArr[i/16] &= bitMask2[(i%16)>>1]; }


      boolean isPrime (int i) {
            if (i == 2 ) return true;
            if ((i&1) == 0) return false;
            else  return (bitArr[i>>4] & bitMask[(i&15)>>1]) != 0;
       }

       ArrayList<Long> factorize (long num) {
            ArrayList <Long> fakt = new ArrayList <Long>();
            int maks = (int) Math.sqrt(num*1.0) +1;
```

```
        int pCand =2;

        while (num > 1 & pCand < maks) {
              while ( num % pCand == 0){
                 fakt.add((long) pCand);
                 num /= pCand;
              }
              pCand = nextPrime(pCand);
             // maks = (int) Math.sqrt(num*1.0) +1;
        }
        if (pCand>=maks) fakt.add(num);
     return fakt;
  } // end factorize


int nextPrime(int i) {
  // returns next prime number after number 'i'

     int k ;
     if ((i&1)==0) k =i+1; // if i is even, start at i+1
     else k = i+2;       // next possible prime
     while (!isPrime(k)) k+=2;
     return k;

 } // end nextPrime

 int lastPrime() {
       int j = ((bitLen>>1)<<1) -1;
       while (! isPrime(j) ) j-=2;
       return j;
```

```
}// end lastPrime

long largestLongFactorizedSafe () {
      long l;
       int i,j = ((bitLen>>1)<<1) -1;
       while (! isPrime(j) ) j-=2;
       i = j-2;
       while (! isPrime(i) ) i-=2;
       return (long)i*(long)j;
 }// end largestLongFactorizedSafe


void printAllPrimes(){
      for ( int i = 2; i <= bitLen; i++)
       if (isPrime(i)) System.out.println(" "+i);
} // end printAllPrimes

int numberOfPrimesLess(int n){
      int num = 2;  // we know 2 and 3 are primes
      int p ;

      for (p=3 ; p < n;p = nextPrime(p) ){
            num++;
      }
      return num;
} // end numberOfPrimesLess


 void generatePrimesByEratosthenes() {
            int m = 3, m2 = 6,mm = 9;   // next prime
```

```
                    setNotPrime(1);      // 1 is not a prime


                    while ( mm < bitLen) {
                            m2 = m+m;
                            for ( int k = mm; k < bitLen; k +=m2){
                                  setNotPrime(k);
                            }
                            m = nextPrime(m);
                            mm= m*m;
                        }
            } // end generatePrimesByEratosthenes


} // end class Twin Primepair
```

Question 3.2

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

///---------------------------------------------------------
//     Fil: TwinPrimePairPara.java
//     Implements Sequential and Parallel Prime pair finding
//     written by: Eric Jul, University of Oslo, 2019
//
//     Cloned from:
//     Fil: EratosthenesSil.java
//     implements bit-array (Boolean) for prime numbers
//     written by:  Arne Maus , Univ of Oslo, 2013, 2015
//
//
//---------------------------------------------------------

/**
* Implements the bitArray of length 'bitLen' [0..bitLen ]
*   1 - true (is prime number)
*   0 - false
*  can be used up to 2 G Bits (integer range)
```

```
*/
public class TwinPrimePairPara
{
   byte [] bitArr ;
   int bitLen;
   final  int [] bitMask ={1,2,4,8,16,32,64,128};
   final  int [] bitMask2 ={255-1,255-2,255-4,255-8,255-16,255-32,255-64,255-
128};
   CyclicBarrier readyToGo, allDone;


   TwinPrimePairPara (int max) {
      bitLen = max;
      bitArr = new byte [(bitLen/16)+1];
      setAllPrime();
      generatePrimesByEratosthenes();
   }

   public static void main(String args[]) {
      TwinPrimePairPara sil;

      if ( args.length != 2)  {
         System.out.println("use: >java TwinPrimePairPara <Max>
<threadCount>");
         System.exit(0);
      }
      int num = Integer.parseInt(args[0]);
      int threadCount = Integer.parseInt(args[1]);
      if (!((num >= 5) && (threadCount >= 2))) {
         System.out.println("Bad parameters: Max must be at least 5 and
threadCount at least 2");
         System.exit(0);
      };

      // Here we generate the sil sequentially as we assumme that you already
have a parallel
      // version of the sil generation from Oblig 3
      long silTime = System.nanoTime();
      sil = new TwinPrimePairPara(num);
      silTime = System.nanoTime() - silTime;
      System.out.println("Sil sequential generation time: " +
(silTime/1000000.0) + " ms");

      // now do the Prime pair calculations
      sil.doit(sil, num, threadCount);

   } // end main()

   void doit(TwinPrimePairPara sil, int num, int threadCount) {
      ArrayList<int[]> pairs = new ArrayList<int[]>();
      readyToGo = new CyclicBarrier(threadCount+1); // includes main() thread
      allDone = new CyclicBarrier(threadCount+1);

      // Sequential version of TwinPrimePair
      long seqTime = System.nanoTime();  // Start sequential timing
      int[] pair = new int[2];
      int nextSP, nextEP, lastP;
```

```
      lastP = sil.lastPrime();
      nextSP = 3;
      nextEP = sil.nextPrime(nextSP);
      while (nextEP <= lastP) {
         if ((nextEP-nextSP) == 2) {
            pair = new int[2];
            pair[0] = nextSP;
            pair[1] = nextEP;
            pairs.add(pair);
         }
         nextSP = nextEP;
         nextEP = sil.nextPrime(nextSP);
      };
      seqTime = System.nanoTime()-seqTime;

      System.out.println("Twin Prime Pairs Sequential time " +
(seqTime/1000000.0) + " ms\n");

      // sil.printpairs(pairs);


      // Parallel version

      long paraTime = System.nanoTime();

      // Create a pair list for each thread
      ArrayList<ArrayList<int[]>> pairLists = new
ArrayList<ArrayList<int[]>>();

      for (int i = 0; i < threadCount; i++) {
         pairLists.add(new ArrayList<int[]>());
      }

      // start threads
      int largestPrime = sil.lastPrime();
      int chunkSize = largestPrime/threadCount;
      int lastThread = threadCount - 1;
      int from = 2;
      for (int i = 0; i < threadCount; i++) {
         int fromPrime = sil.nextPrime(from);
         int uptoPrime;

         if (i < lastThread) {
            uptoPrime = sil.nextPrime(from + chunkSize);
         } else uptoPrime = largestPrime;
         // System.out.println("Starting thread "+i+" from "+fromPrime+" upto
"+uptoPrime);
         new Thread(new Para(i, threadCount, sil, fromPrime, uptoPrime,
pairLists.get(i))).start();

         from = from + chunkSize;
      }

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}
```

```
      // Now the threads are doing their thing

      try {
         allDone.await(); // await all worker threads DONE
      } catch (Exception e) {return;}

      // Combine results
      ArrayList<int[]> currentList;
      ArrayList<int[]> combinedList = new ArrayList<int[]>();
      for (int i = 0; i < threadCount; i++) {
         currentList = pairLists.get(i);
         // System.out.println("pair List " + i + " number of pairs: " +
currentList.size());
         for (int j = 0; j < currentList.size(); j++) {
            combinedList.add(currentList.get(j));
         }
      }


      paraTime = System.nanoTime() - paraTime;

      System.out.println("Prime pair Parallel time " + (paraTime/1000000.0) +
" ms\nSpeedup "+ seqTime*1.0/paraTime);

      /*
      sil.printpairs(combinedList);

      for (int i = 0; i < threadCount; i++) {
         System.out.println("pair List " + i);
         sil.printpairs(pairLists.get(i));
      }
      */

   }

   void printpairs(ArrayList<int[]> pairs) {
      for (int i = 0; i < pairs.size(); i++) {
         System.out.println(" "+i+": ["+pairs.get(i)[0]+",
"+pairs.get(i)[1]+"]");
      }
      System.out.println("------------------");
   }


   void setAllPrime() {
      for (int i = 0; i < bitArr.length; i++) {
         bitArr[i] = -1 ;     // alt    ( byte)255;
      }
   }

   void setNotPrime(int i) {
      bitArr[i/16] &= bitMask2[(i%16)>>1];
   }

   boolean isPrime (int i) {
      if (i == 2 ) return true;
```

```
   if ((i&1) == 0) return false;
   else  return (bitArr[i>>4] & bitMask[(i&15)>>1]) != 0;
}

ArrayList<Long> factorize (long num) {
   ArrayList <Long> fakt = new ArrayList <Long>();
   int maks = (int) Math.sqrt(num*1.0) +1;
   int pCand =2;

   while (num > 1 & pCand < maks) {
      while ( num % pCand == 0){
         fakt.add((long) pCand);
         num /= pCand;
      }
      pCand = nextPrime(pCand);
      // maks = (int) Math.sqrt(num*1.0) +1;
   }

   if (pCand>=maks) fakt.add(num);
   return fakt;
} // end factorize


int nextPrime(int i) {
   // returns next prime number after number 'i'

   int k;
   if (i < 2) return 2;
   if (i == 2) return 3;
   if ((i&1)==0) k =i+1; // if i is even, start at i+1
     else k = i+2;       // next possible prime
   while (!isPrime(k)) k+=2;
   return k;
} // end nextPrime

int lastPrime() {
   int j = ((bitLen>>1)<<1) -1;
   while (! isPrime(j) ) j-=2;
   return j;
} // end lastPrime

long largestLongFactorizedSafe () {
   long l;
   int i,j = ((bitLen>>1)<<1) -1;
   while (! isPrime(j) ) j -= 2;
   i = j-2;
   while (! isPrime(i) ) i -= 2;
   return (long)i*(long)j;
} // end largestLongFactorizedSafe


    void printAllPrimes(){
          for ( int i = 2; i <= bitLen; i++)
           if (isPrime(i)) System.out.println(" "+i);
    } // end printAllPrimes

    int numberOfPrimesLess(int n){
```

```
            int num = 2;  // we know 2 and 3 are primes
            int p ;

            for (p=3 ; p < n;p = nextPrime(p) ){
                  num++;
            }
            return num;
       } // end numberOfPrimesLess


        void generatePrimesByEratosthenes() {
                    int m = 3, m2=6,mm =9;     // next prime
                    setNotPrime(1);      // 1 is not a prime

                    while ( mm < bitLen) {
                                m2 = m+m;
                                for ( int k = mm; k < bitLen; k +=m2){
                                      setNotPrime(k);
                                }
                                m = nextPrime(m);
                                mm= m*m;
                          }
            } // end generatePrimesByEratosthenes

   class Para implements Runnable {
      int ind, from, upto, threadCount;
      TwinPrimePairPara sil;
      ArrayList<int[]> mypairs;

      Para(int in, int c, TwinPrimePairPara sil, int from, int upto,
ArrayList<int[]> mypairs) {
         ind = in;
         threadCount = c;
         this.sil = sil;
         this.from = from;
         this.upto = upto;
         this.mypairs = mypairs;
      } // konstruktor

      public void run() { // Her er det som kjores i parallell:

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      int[] pair;

      int nextSP, nextEP, lastP;
      lastP = upto;
      nextSP = from;
      nextEP = sil.nextPrime(nextSP);
      while (nextEP <= lastP) {
         if ((nextEP-nextSP) == 2) {
            pair = new int[2];
            pair[0] = nextSP;
            pair[1] = nextEP;
            mypairs.add(pair);
```

```
         }
         nextSP = nextEP;
         nextEP = sil.nextPrime(nextSP);
      };

      // Done

      try {
         allDone.await(); // await all threads done
      } catch (Exception e) {return;}

      } // end run
   } // end class Para

} // end class TwinPrimePair
```