CSE 520
Fall 1999
Course Project: CCS/Java implementation of Eratosthenes Sieve

Anirudh Modi

December 11, 1999

1 Introduction

This project implements the *Sieve of Eratosthenes* using CCS[1] and Java. This is an algorithm for generating a list of prime numbers, as illustrated in figure 1. It originally consists of a source generating natural numbers, and one sieve filtering out all the multiples of two. When the last sieve sees a number that it cannot filter, it creates a new sieve to filter out all the multiples of that number. Thus, after the sieve filtering out multiples of two sees the number three, it creates a new sieve that filters out the multiples of three. This then continues with the three sieve eventually creating a sieve to filter out all multiples of five (the next prime number), and so on. Thus, after a while, there will be a chain of sieves each filtering out a different prime number. If any number passes through all of the sieves and reaches the end with no sieve waiting, it must be another prime and so a new sieve is created for it.

2 CCS preliminaries

The Calculus of Communicating Systems (CCS) is an algebra for specifying and reasoning about concurrent systems. As an algebra, CCS provides a set of terms, operators and axioms that can be used to write and manipulate algebraic expressions. The expressions define the elements of a concurrent system and the manipulations of these expressions reveal how the system behaves. CCS provides the foundation for answering very pragmatic questions. For example, the equality of two "agents" can be determined. If the two agents represent different implementations, then we can determine if one system is substitutable for the other. CCS also establishes the foundation for determining if an agent possesses certain "safety" or "liveness" properties, such as freedom from deadlock.

[1]Calculus of Communicating Systems

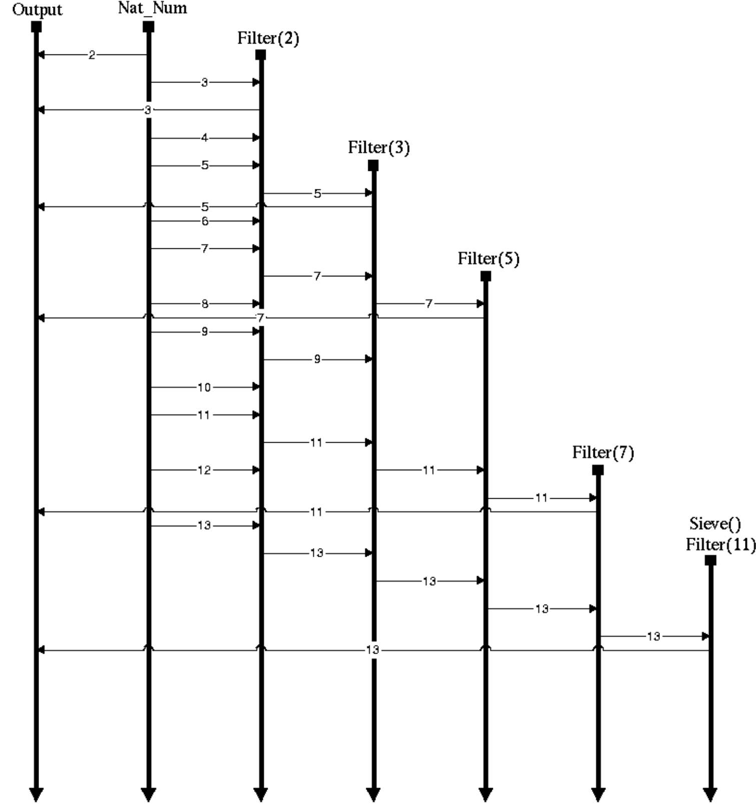

Figure 1: Illustration of the *Sieve of Eratosthenes* for obtaining the first six primes

The CCS grammar (including that for *if-then-else*) specified in the class has been used for the specification of the system.

3 CCS specification for the system

The specifications for the concurrent system are given below:

$$Queue(\varepsilon) \stackrel{def}{=} inQ(x).Queue(x) + empty.Queue(\varepsilon)$$

$$Queue(s :: v) \stackrel{def}{=} inQ(x).Queue(x :: s :: v) + \overline{outQ}(v).Queue(s)$$

$$Nat_Num \stackrel{def}{=} \overline{outN}(n).incr(n).Nat_Num$$

$$Filter(n) \stackrel{def}{=} inF(x).(\text{if } (mod(x,n) = 0) \text{ then } \overline{outF}(x).Filter(n) \text{ else } 0)$$

$$Sieve \stackrel{def}{=} inS(x).\overline{outS}(x).(Filter(x)[\overline{outF} \mapsto a] \mid Queue[inQ \mapsto a, \overline{outQ} \mapsto b] \mid Sieve[inS \mapsto b]) \backslash L_S$$

$$\text{where } L_S = \{a, b, inQ, \overline{outQ}, \overline{outF}\}$$

Finally, the Eratosthenes sieve can be defined using the above definitions as:

$$Eratosthenes \stackrel{def}{=} (Nat_Num[\overline{outN} \mapsto a] \mid Queue[inQ \mapsto a, \overline{outQ} \mapsto b] \mid Sieve[inS \mapsto b]) \backslash L_E$$

$$\text{where } L_E = \{a, b, \overline{outN}, inQ, \overline{outQ}, inS\}$$

It can be shown by looking at the state transition graph that the above specification is correct, and that it is weakly bisimilar to process $P(1)$ where:

$$P(n) = \overline{out}(Prime(n)).P(n+1)$$

where $Prime(n)$ is the nth prime number starting from 2. Thus, the implementation based on this specification will be free from deadlock.

4 Implementation of the solution

The entire program has been written in Java using the thread libraries for spawning multiple processes. The CCS specifications stated above have been used for the implementation. The `Queue` class has been implemented as a passive object with synchronization, and is thus wait free because of its unbounded nature. This allows the processes to be more independent of each other, in the sense that one process can be faster than than the other (thus filling up queue). The main calling routine is specified in `class Eratosthenes`. The other classes `Nat_Num`, `Sieve` and `Filter` are extended from the class `MyThread` which in turns extends the Java defined class `Thread`. The program does not use any array or bounded objects, thus the number of primes it can generate is only limited by memory. All the processes and objects are created dynamically by the program as and when they are needed.

For the program to exit gracefully after terminating all the threads, the process `Nat_Num()` is made to exit after generating the number -1 (negative one) which acts as a signal to all the filters and the sieve it passes through, thus terminating them. The final program can be invoked by typing "`java Eratosthenes`" and a sample output for an argument of 10 (which specifies the number of primes to be generated) is given below:

```
=========================================================
CCS: Starting Eratosthenes Sieve implementation......
Generating the first 40 Prime Numbers.
=========================================================
1th prime: 2
2th prime: 3
3th prime: 5
4th prime: 7
5th prime: 11
6th prime: 13
7th prime: 17
8th prime: 19
9th prime: 23
10th prime: 29
```

```
11th prime: 31
12th prime: 37
13th prime: 41
14th prime: 43
15th prime: 47
16th prime: 53
17th prime: 59
18th prime: 61
19th prime: 67
20th prime: 71
21th prime: 73
22th prime: 79
23th prime: 83
24th prime: 89
25th prime: 97
26th prime: 101
27th prime: 103
28th prime: 107
29th prime: 109
30th prime: 113
31th prime: 127
32th prime: 131
33th prime: 137
34th prime: 139
35th prime: 149
36th prime: 151
37th prime: 157
38th prime: 163
39th prime: 167
40th prime: 173
========================================================
Clearing all queues.....done.
All 40+2 threads have terminated.
CCS ends.
Written by Anirudh Modi <anirudh@anirudh.net> on 12/7/99
========================================================
```

5 Java code for the program

The entire Java source code for the implemented progam has been attached at the end of this document.

References

[1] Milner, R., “Communication and Concurrency,” Prentice Hall International, 1989.

[2] Magee, J. and Kramer, J., “Concurrency: State Models and Java Programs”, Wiley, 1999. [http://www-dse.doc.ic.ac.uk/concurrency/]

```
//****************************************************************************
public class Eratosthenes
{
//****************************************************************************
Eratosthenes()     // Constructor
{
}
//****************************************************************************
public static void main(String[] args)     // main routine
{
int numPrimes = 100;
// Check for command-line argument
if (args.length == 1)
     {
      try {numPrimes = Integer.parseInt(args[0]);} catch (Exception e)
          {
           System.out.println("Invalid argument \"" + args[0] +
               "\" passed to the program! Exiting...");
           return;
          }
     }

// Print welcome message
System.out.println("========================================================");
System.out.println("CCS: Starting Eratosthenes Sieve implementation......");
System.out.println("Generating the first " + numPrimes + " Prime Numbers.");
System.out.println("========================================================");

Queue outNat = new Queue();
Nat_Num nat_num = new Nat_Num(outNat);
Queue outSieve = new Queue();
Queue inSieve = outNat;     // Relabel outNat as inSieve
Sieve sieve = new Sieve(inSieve, outSieve);

// Start the processes/threads
//nat_num.setLimit(550); // Set limit for generation of numbers (optional)
nat_num.start();     // Start process Nat_Num() to generate natural
               // numbers starting from 2
sieve.start();     // Start process Sieve() which will spawn processes
          // Filter(p_i) as and when necessary

// Print out the prime numbers
for(int i = 0; i < numPrimes; i++)
     System.out.println(i+1 + "th prime: " + outSieve.receive());

// Terminate all processes/threads
System.out.println("========================================================");
System.out.print("Clearing all the queues.....");

// Interrupt the Nat_Num() process.
// Nat_Num() will then generate the integer "-1" in its output buffer before
// terminating which will act as an exit  signal to all other processes
```

```
nat_num.myInterrupt();     // Stop generating natural numbers
while (outSieve.receive() != -1) {} // Receive any remaining numbers till -1
while (sieve.isAlive()) {}     // Wait for Sieve() to exit gracefully

System.out.println("done.");
System.out.println("All " + numPrimes + "+" + 2 + " threads have terminated.");

// Print termination message
System.out.println("CCS ends.");
System.out.println("Written by Anirudh Modi <anirudh@anirudh.net> on 12/7/99");
System.out.println("========================================================");
}
//*****************************************************************************
}     // Class definition for Eratosthenes ends
//*****************************************************************************
public class Queue
{
//*****************************************************************************
protected static class Node {      // Class definition for Node
      protected int value;
      protected Node next = null;
      protected Node(int x) { value = x; } // constructor

      protected boolean compareANDswap(Node assumedNext, Node newNext)
      {
       boolean equality = (next == assumedNext);     // compare
       if (equality)
          next = newNext;                      // swap
       return equality;
      }
     }
// Private variables of class Queue
private Node head;
private Node tail;
private int gInvalidNum = Integer.MIN_VALUE;
//*****************************************************************************
public Queue()           // Constructor
{
// head, tail and dummy intially point to an empty node
Node dummy = new Node(gInvalidNum);
head = new Node(gInvalidNum);
tail = new Node(gInvalidNum);
head.next = dummy;
tail.next = dummy;
}
//*****************************************************************************
private void enqueue(int x) // receive value into queue for sending in future
{
Node newnode = new Node(x);

// Read tail
Node tailNext;
```

```
synchronized(tail)
      tailNext = tail.next;

// Read last.next (last = tail.next)
Node lastNext;
synchronized(tailNext)
      lastNext = tailNext.next;

// only proceed if tail is unchanged since last read
if (tailNext == tail.next)
     {
      if (lastNext == null) // a spot is available to insert node
          {
           if (tailNext.compareANDswap(lastNext, newnode))
               {
                tail.compareANDswap(tailNext, newnode);
                return;
               }
          }
      else
          {
           tail.compareANDswap(tailNext, lastNext);
          }
     }
}
//****************************************************************************
private int dequeue()     // return a value from the end of the queue
{
// atomically read head, tail
Node headNext;
synchronized(head)
      headNext = head.next;

Node tailNext;
synchronized(tail)
      tailNext = tail.next;

Node first = headNext.next;

// only proceed if head still same after reading tail
if (headNext == head.next)
     {
      if (headNext == tailNext)
          {
           if (first == null)
               return gInvalidNum;
           else          // update
               tail.compareANDswap(tailNext, first);
          }
      else
          {
           int x = first.value;
```

```
          if (head.compareANDswap(headNext, first))
              {
               first.value = gInvalidNum;
               return x;
              }
         }
     }
return gInvalidNum;
}
//*****************************************************************************
public int receive()     // send value from queue
{
int n;
while((n = dequeue()) == gInvalidNum)  // Loop until a valid value is found
     try { wait();} catch (Exception e) {}
return n;
}
//*****************************************************************************
public void send(int x) // receive value into queue for sending in future
{
enqueue(x);
}
//*****************************************************************************
}     // Class definition for Queue ends
//*****************************************************************************
public class MyThread extends Thread
{
boolean gStopFlag;
boolean gDebugThreads;
//*****************************************************************************
public MyThread()      // Constructor
{
gStopFlag = false;
gDebugThreads = false;     // For debugging of threads
}
//*****************************************************************************
public void run()     // The main routine for invocation of the thread
{
}
//*****************************************************************************
public void myInterrupt()
{
gStopFlag = true;
}
//*****************************************************************************
}     // Class definition for MyThread ends
//*****************************************************************************
public class Nat_Num extends MyThread
{
private Queue outN;
private int gMaxNum;
//*****************************************************************************
```

```
public Nat_Num(Queue outn)      // Constructor
{
super();
outN = outn;
gMaxNum = Integer.MAX_VALUE;
}
//****************************************************************************
public void run()     // The main routine for invocation of the thread
{
if (gDebugThreads)
     System.out.println("Starting Nat_Num (" + getName() + ")");

int n = 2;
// Generate consecutive natural numbers until the stop flag is activated
while (!gStopFlag && n < gMaxNum)
     {
      outN.send(n);      // Send "n" to the output queue
      ++n;
     }
outN.send(-1);     // Send "-1" before exiting as a signal to all the filters

if (gDebugThreads)
     System.out.println("Terminating Nat_Num (" + getName() + ")");
}
//****************************************************************************
public void setLimit(int n)
{
gMaxNum = n;
}
//****************************************************************************
}     // Class definition for Nat_Num ends
//****************************************************************************
public class Sieve extends MyThread
{
private Queue inS;
private Queue outS;
//****************************************************************************
public Sieve(Queue ins, Queue outs)      // Constructor
{
super();
inS = ins;
outS = outs;
}
//****************************************************************************
public void run()     // The main routine for invocation of the thread
{
if (gDebugThreads)
     System.out.println("Starting Sieve (" + getName() + ")");

Queue inF = inS;     // Relabel inS as inF
for(;;)          // Loop forever
     {
```

```
      int n = inF.receive(); // receive from input queue
      outS.send(n);          // Send "n" to output queue
      if (n == -1) break;     // Exit as -1 has been detected!!
      Queue outF = new Queue();     // Create new queue
      Filter newfilter = new Filter(n, inF, outF); // Create new filter(n)
       newfilter.start();
      inF = outF;     // Relabel outF as inF
     }

if (gDebugThreads)
     System.out.println("Terminating Sieve (" + getName() + ")");
}
//******************************************************************************
}     // Class definition for Sieve ends
//******************************************************************************
public class Filter extends MyThread
{
private int gn;          // global n
private Queue inF, outF;
//******************************************************************************
public Filter(int n, Queue inf, Queue outf) // Constructor
{
super();
gn = n;
inF = inf;
outF = outf;
}
//******************************************************************************
public void run()     // The main routine for invocation of the thread
{
int n = gn;

if (gDebugThreads)
     System.out.println("Starting Filter(" + n + ") (" + getName() + ")");

int multiple_of_n = n;
int m = 0;
while (m != -1)     // Stop if "-1" is received
     {
      m = inF.receive();     // receive from input queue
      while (multiple_of_n < m)
          multiple_of_n += n;
      if (multiple_of_n > m)          // if m is not a multiple of n
          outF.send(m);          // send to output queue
     }

if (gDebugThreads)
     System.out.println("Terminating Filter(" + n + ") " + getName());
}
//******************************************************************************
}     // Class definition for Filter ends
//******************************************************************************
```