Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

Learning Experience:

- Ability to understand and modify code written by someone else (using the skeleton code requires a basic understanding of how things work and inferring how it is supposed to be modified such that it fits the theoretical model that one has in mind)
- Applying information learnt in one context to a similar but different context – using synchronize in a different setting that what it was introduced as.

Suggestion:

On my honor, I have neither given nor received unauthorized aid in doing this assignment.

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

PSEUDOCODE:

Note: This pseduocode only accounts for the critical section portion of the code, and doesn't include things implemented solely by the skeleton code provided.
Note: Some method names and ways comparisons are performed have been changed to make the pseudocode more readable.

```
GET_CHOPSTICKS:

while ( notAcquired chopsticks )
{

     // Wait until rightStick is available
     while (!pickupRightStick())
     {
          synchronized (rightStick)
          {
               wait();
          }
     }

     // Right stick acquired

     // Release rightStick and wait if leftStick is unavailable
     if (!pickUpLeftStick())
     {
          putdown_rightstick

          // Let philosopher on the right know rightStick is
available
          synchronized (rightStick)
          {
               notify();
          }

          // Wait until left stick is available
          synchronized (leftStick)
          {
               wait();
          }

          // Repeat loop when leftStick is finally available
     }
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
     // If reached here, then both Chopsticks have been
acquired. Proceed to eating
}

RETURN_CHOPSTICKS:

putDownRightChopstick();
synchronized(rightStick)
{
     notify();
}
putDownLeftChopstick();
synchronized(leftStick)
{
     notify();
}
```

Proof:

The methods that `pickUpRightStick( )` and `pickUpLeftStick( )` are synchronized on the Chopstick instance object, therefore they provide mutual exclusion – this is already implemented in the skeleton code. Now, the psedudocode given above does the following:
Tries to get right chopstick – blocks if not available.
Right stick available, then tries to get left chopstick – returns right chopstick and blocks until left chopstick is available.
Tries the "two" steps above again until both chopsticks have been acquired.
Once both chopsticks have served their purpose, they are returned and the wait( )ing/blocked threads that require the (just) returned chopsticks can go ahead and run.
The program won't deadlock because there is no situation where the two keys, i.e. leftStick and rightStick are needed and progression is prevented without these keys, while another thread is also experiencing the same issue. Once a thread as acquired a key, which only one thread can do at a time, it will never need another key to finish progression and release the key it acquired.
Starvation is prevented by making sure the thread (philosopher) notifies its neighbor whenever it returns a chopstick it has taken. So, lets take the worst case scenarios for example:
Assume we have n philosophers, where $n > 2$. All philosophers coincidentally finished thinking at the same time and want to eat. They all grab their right chopstick (with enough luck such that context switch happens after each grab their right chopstick), and now none can eat. When this happens, one philosopher (the lucky thread that the OS scheduled to go) will end up putting its rightStick down and notifying any philosopher that happened to be waiting on that specific chopstick (i.e the philosopher on the right). Now, let's say this happens in such a way every thread (philosopher) that is scheduled by the OS is stuck such that its left chopstick is unavailable, therefore making it put down its right chopstick, notifying the thread on the right and waiting until it's left chopstick is available. But, once at least floor(N/2) philosophers have

put down their right chopsticks, then the remaining philosophers would necessarily have their left chopstic available.

… 1 [R/L] 2 [R/L] 3 [R/L] 4 [R/L] 5 [R/L] 1…

Lets assume the above scenario, where philosophers 1-5 are sitting in a circular table and [R/L] indicates the chopstick in the middle of each philosophers where R indicates that it's located to the Right of the philosopher on the same side, and L indicates that it's located to the philosopher on the same side of the square bracket. The … indicates that this is the point where it forms a circle.
In this scenario, if all philosophers picked up their right chopstick at the same time, then say, 3 decides to drop its right stick and wait. Now if 4 is scheduled next, it would have its left stick available and it won't need to block. But, to consider a worst case scenario where each consequent philosopher has bad luck, we'll need to select either philosophers 1, 2, or 5. If we select 2, then 2 would drop its left stick, notify 3, and block. Now when 3 is rescheduled it will be able to go again. But, lets assume 5 was chosen after 3. Now, 5 drops its right stick, notifies 1 and waits. Now, since 3 and 5 are waiting, the only ones remaining are 1,2 and 4. If 1 or 4 was to run next, they will have their left chopstick available and they will progress. No starvation. But if 2 was to run, it will drop its right stick, notify 3 and block. Now since 3 has both left and right sticks available, it will be able to run freely next time it's scheduled. Similarly, 1 and 4 would also be able to run for the same reason mentioned earlier. Therefore, there won't be a starvation where all philosophers pick up the forks, decide the left is not available, drop it, pick it up again … etc.

Last 100 lines of code:

```
Hume wants right chopstick (0)
Socrates has right chopstick (8)
Frege is begining to think for 4259 milliseconds
Socrates wants left chopstick (7)
Socrates has both left chopstick (7) and right chopstick (8)
Socrates is begining to eat for 4715 milliseconds
Machiavelli is done thinking
Machiavelli wants right chopstick (6)
Machiavelli has right chopstick (6)
Machiavelli wants left chopstick (5)
Machiavelli has both left chopstick (5) and right chopstick (6)
Machiavelli is begining to eat for 4345 milliseconds
Aristotle is done thinking
Aristotle wants right chopstick (2)
Aristotle has right chopstick (2)
Aristotle wants left chopstick (1)
Aristotle was unable to get the left chopstick (1)
Aristotle politely returned right chopstick (2)
Voltaire is done eating
Voltaire finished using right chopstick (4)
Voltaire finished using left chopstick (3)
Kant wants right chopstick (5)
Epicurius has right chopstick (3)
Voltaire is begining to think for 3249 milliseconds
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
Epicurius wants left chopstick (2)
Epicurius has both left chopstick (2) and right chopstick (3)
Epicurius is begining to eat for 4686 milliseconds
Nietzsche is done thinking
Nietzsche wants right chopstick (7)
Frege is done thinking
Frege wants right chopstick (9)
Frege has right chopstick (9)
Frege wants left chopstick (8)
Frege was unable to get the left chopstick (8)
Frege politely returned right chopstick (9)
Seneca is done eating
Seneca finished using right chopstick (1)
Seneca finished using left chopstick (0)
Aristotle wants right chopstick (2)
Hume has right chopstick (0)
Seneca is begining to think for 2142 milliseconds
Hume wants left chopstick (9)
Hume has both left chopstick (9) and right chopstick (0)
Hume is begining to eat for 4562 milliseconds
Socrates is done eating
Socrates finished using right chopstick (8)
Socrates finished using left chopstick (7)
Frege wants right chopstick (9)
Socrates is begining to think for 4059 milliseconds
Nietzsche has right chopstick (7)
Nietzsche wants left chopstick (6)
Nietzsche was unable to get the left chopstick (6)
Nietzsche politely returned right chopstick (7)
Voltaire is done thinking
Voltaire wants right chopstick (4)
Voltaire has right chopstick (4)
Voltaire wants left chopstick (3)
Voltaire was unable to get the left chopstick (3)
Voltaire politely returned right chopstick (4)
Machiavelli is done eating
Machiavelli finished using right chopstick (6)
Machiavelli finished using left chopstick (5)
Nietzsche wants right chopstick (7)
Kant has right chopstick (5)
Machiavelli is begining to think for 1308 milliseconds
Kant wants left chopstick (4)
Nietzsche has right chopstick (7)
Kant has both left chopstick (4) and right chopstick (5)
Nietzsche wants left chopstick (6)
Kant is begining to eat for 3691 milliseconds
Nietzsche has both left chopstick (6) and right chopstick (7)
Nietzsche is begining to eat for 4882 milliseconds
Seneca is done thinking
Seneca wants right chopstick (1)
Seneca has right chopstick (1)
Seneca wants left chopstick (0)
Seneca was unable to get the left chopstick (0)
Seneca politely returned right chopstick (1)
Machiavelli is done thinking
Machiavelli wants right chopstick (6)
Epicurius is done eating
Epicurius finished using right chopstick (3)
Epicurius finished using left chopstick (2)
Voltaire wants right chopstick (4)
Aristotle has right chopstick (2)
Epicurius is begining to think for 1920 milliseconds
Aristotle wants left chopstick (1)
Aristotle has both left chopstick (1) and right chopstick (2)
Aristotle is begining to eat for 3001 milliseconds
-----------------------------------------------
Seneca ate 6 times (22087 ms) and pondered 7 times (20313ms)
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
Aristotle ate 8 times (20638 ms) and pondered 8 times (24967ms)
Epicurius ate 8 times (21740 ms) and pondered 9 times (19860ms)
Voltaire ate 8 times (24168 ms) and pondered 9 times (24535ms)
Kant ate 7 times (11759 ms) and pondered 7 times (21610ms)
Machiavelli ate 8 times (27887 ms) and pondered 9 times (23040ms)
Nietzsche ate 7 times (20559 ms) and pondered 7 times (21207ms)
Socrates ate 5 times (22170 ms) and pondered 6 times (14229ms)
Frege ate 6 times (18565 ms) and pondered 7 times (18941ms)
Hume ate 6 times (16615 ms) and pondered 6 times (20032ms)
```

Analysis:
Each philosopher is doing what it's expected to do, i.e. waiting for chopsticks if unavailable, and putting down right chopstick when left is not available, and eating when both chopsticks are acquired. Additionally, no philosopher takes a chopstick that is already being used by another philosopher, which is an important thing to consider. The scheduling of threads by the OS might allow a very small time for a thread to run, which we sometimes see with one philosopher executing "one" action before a context switch occurs. Other times, the philosopher might be deliberately waiting, such as when the right chopstick is being used by the philosopher on the left.

FLESHED OUT CODE:

```
// DiningPhiliphers.java (skeleton)
//
// - a classic synchronization problem
//
// Skeleton code derived from Dave Small's
DiningPhiliphers.java v4.0

import java.util.Random;

//================================================ class
DiningPhilosophers

class DiningPhilosophers
{
  public static void main( String[] arg )
  {
    new DiningPhilosophers( 10, 60000 );
  }

  private String[] name = { "Seneca", "Aristotle", "Epicurius",
"Voltaire",
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
                    "Kant", "Machiavelli", "Nietzsche",
"Socrates",
                    "Frege", "Hume" };

  private Philosopher[] thinker;
  private Chopstick[]   chopstick;

  public DiningPhilosophers( int numPhilosophers, int duration
)
  {
    initialize( numPhilosophers );  // construct the
philosophers & chopsticks
    startSimulation();
    sleep( duration );              // let simulation run for
desired time
    shutdownPhilosophers();         // *gracefully* shut down
the philosophers
    printResults();
  }

  private void initialize( int n )
  {                                 // handles 2 to 10
philosophers
    if ( n > 10 )
      n = 10;
    else if ( n < 2 )
      n = 2;

    thinker = new Philosopher[n];
    chopstick = new Chopstick[n];

    for ( int i = 0 ; i < n ; i++ )
      chopstick[i] = new Chopstick(i);

    for ( int i = 0 ; i < n ; i++ )
      thinker[i] = new Philosopher( name[i], chopstick[i],
chopstick[(i+1)%n] );
  }

  private void startSimulation()
  {
    int n = thinker.length; // the number of philosophers

    System.out.print( "Our " + n + " philosophers (" );
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
    for ( int i = 0 ; i < (n-1) ; i++ )
      System.out.print( name[i] + ", " );
    System.out.println( "and " + name[n-1] +
                ") have gather to think and dine" );

    System.out.println( "-------------------------------------
---------");

    for ( int i = 0 ; i < n ; i++ )
      thinker[i].start();
  }

  private void sleep( int duration )
  {
    try
    {
      Thread.currentThread().sleep( duration );
    }
    catch ( InterruptedException e )
    {
      Thread.currentThread().interrupt();
    }
  }

  private void shutdownPhilosophers()
  {
    int n = thinker.length;   // number of philosophers

    try
    {
      // Interrupt philosophers and wait till done
      for(int i = 0; i < n; ++i)
      {
        thinker[i].interrupt();
        thinker[i].join();
      }
    }
    catch ( InterruptedException e)
    {
      Thread.currentThread().interrupt();
    }
  }

  private void printResults()
```

```
  {
    System.out.println( "----------------------------------------
---------");

    int n = thinker.length; // the number of philosophers

    for ( int i = 0 ; i < n ; i++ )
      System.out.println( thinker[i] );

    System.out.flush();
  }
}

//================================================ class
Philosopher

class Philosopher extends Thread
{
  static private Random random = new Random();

  private String name;
  private Chopstick leftStick;
  private Chopstick rightStick;

  private int eatingTime   = 0;
  private int thinkingTime = 0;
  private int countEat     = 0;
  private int countThink   = 0;

  public Philosopher( String name, Chopstick leftStick,
Chopstick rightStick )
  {
    this.name = name;
    this.leftStick = leftStick;
    this.rightStick = rightStick;
  }

  public String toString()
  {
    return name + " ate " + countEat + " times (" +
      eatingTime + " ms) and pondered " + countThink + " times
(" +
      thinkingTime + "ms)";
  }
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
  public void run()
  {

    while (true)
    {
      try
      {
        countThink++;
        thinkingTime += doAction( "think" );
        pickupChopsticks();
        countEat++;
        eatingTime += doAction( "eat" );
        putdownChopsticks();
      }
      catch(InterruptedException e)
      {
        return;
      }
    }

  }

  private int doAction( String act ) throws
InterruptedException
  {
    int time = random.nextInt( 4000 ) + 1000 ;
    System.out.println( name + " is begining to " + act + " for
" + time +
      " milliseconds" );
    sleep( time );

    System.out.println( name + " is done " + act + "ing" );

    return time;
  }

  private void pickupChopsticks() throws InterruptedException
  {
    boolean haveChopsticks = false;


    while (!haveChopsticks)
    {
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
      System.out.println( name + " wants right " + rightStick
);

      // Try to get rightStick
      while(!rightStick.pickUp(this))
      {
        synchronized (rightStick)
        {
          try    // Wait for right stick to be available
          {
            rightStick.wait();
          }
          catch (InterruptedException e)
          {
            throw e;
          }
        }
      }

      System.out.println( name + " has right " + rightStick );
      System.out.println( name + " wants left " + leftStick );

      // Try to get leftStick
      if (!leftStick.pickUp(this))
      {
        // --- Left stick is unavailable --

        // Put down rightStick
        rightStick.putDown(this);

        System.out.println( name + " was unable to get the left
" + leftStick );
        System.out.println( name + " politely returned right "
+ rightStick );

        synchronized (rightStick)
        {
          // Let philosopher to the right know rightStick is
available
          rightStick.notify();
        }

        synchronized (leftStick)
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
          {
            try   // Wait until leftStick is available
            {
               leftStick.wait();
            }
            catch (InterruptedException e)
            {
              throw e;
            }

          }

          // Try to get both chopsticks again
          continue;
        }

        // Mission accomplished!
        System.out.println( name + " has both left " + leftStick
+
          " and right " + rightStick );

        haveChopsticks = true;
      }

    }

    private void putdownChopsticks()
    {
      try
      {
        rightStick.putDown(this);
        System.out.println( name + " finished using right " +
rightStick );

        synchronized (rightStick)
        {
          // Let the philosopher to the right know chopstick is
available
          rightStick.notify();
        }

        leftStick.putDown(this);
        System.out.println( name + " finished using left " +
leftStick );
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
      synchronized (leftStick)
      {
        // Let the philosopher to the left know chopstick is
available
        leftStick.notify();
      }
    }
    catch (RuntimeException e)
    {
      Thread.currentThread().interrupt();
    }
  }
}

//========================================== class
Chopstick

class Chopstick
{
  private final int id;
  private Philosopher heldBy = null;

  public Chopstick( int id )
  {
    this.id = id;
  }

  public String toString()
  {
    return "chopstick (" + id + ")";
  }

  synchronized public boolean pickUp( Philosopher p)
  {
    // Chopstick being used by a philosopher
    if (heldBy != null)
    {
      // Unable to pick up
      return false;
    }

    // Pick up chopstick
    heldBy = p;
```

Dawit Woldegiorgis
COP 4600
Section: 0467
October 13, 2015
Homework #3

```
    return true;
  }

  synchronized public void putDown( Philosopher p)
  {
    if (heldBy == p)
      heldBy = null;

    else
      throw new RuntimeException( "Exception: " + p + "
attempted to put " +
        "down a chopstick he wasn't holding." );
  }
}
```