**CS 3343 Operating Systems Exam 2 100 points**

**Due April 8 at 5pm.**

**No late exams. Exams must be received by April 8 at 5pm.**

**Write all code in the Java programming language to create the following software. Show all screenshots of running programs. Include all source code. Be sure to use the objects using the notation from the book and in class. Do not check for file existence or open errors. No AI or GPT use. Cite all references. Do not use the *“this*” object notation in code. Do not use the *BufferedWriter* class or embed the file opening in parentheses in *try*. Do not write to *System.err*. Microsoft Word only.**

**Email your completed exam to me at** [**harringp@nsuok.edu**](mailto:harringp@nsuok.edu)

**Send email exclusively via** [**https://mail.google.com/**](https://mail.google.com/)

**Sign in with your NSU email and password**

**One exam submission per student.**

**Part 1: Answer the following questions:**

**A black background with white text

Description automatically generated**

1. Dispatch latency is the amount of time the dispatcher takes to stop one thread and put another thread onto the CPU. **(Book: Glossary – G10)**
2. In a database, you have readers and writers. Readers only want to read the data, so it is a lot easier process to go through with, allowing for multiple readers at a time. However, when a writer is involved, it becomes a lot trickier. To keep the problem from occurring, writers need to have exclusive access when writing. The synchronization problem that occurs between the reader and writer is what is known as the Readers-Writers problem. **(Book: 290-291)**
3. The virus could cause the writer to fire when it’s not necessary. Thus, clogging up the processes. It could probably also do the opposite, where it could fire a reader while writing. I’m not well versed on computer viruses, but that is something I could see being a potential issue.
4. This question is sort of similar to number three, in that a virus could cause misinformation thus causing issues. In the book, a semaphore is an integer variable that, apart from initialization, is accessed only through two standard atomic operations (wait() and signal()). The book goes on to state that when one process modifies a semaphore value, no other process can simultaneously modify the same value. This is where I believe a computer virus could invade the system. It could find a way to modify the values and cause issues when normally not possible. **(Book: 272-273)**
5. The book states a deadlock as being a situation where two or more processes are waiting indefinitely for an event that can only be caused by the waiting processes. This leads to the deadlock. Starvation on the other hand is an indefinite block, a process may never be removed from the from the semaphore queue in which it is suspended. **(Book 283 & Ch6 Slides 23)**
6. A computer virus could easily use these two things to lock up an operating system. It could cause deadlocks in situations where there would be none, thus slowing down the operating system. Then there could be situations where a virus could possibly cause starvation and potentially permanently lock up a PC. It could cause a process to permanently sit in the semaphore queue for example.

**Part 2: Java Programming:**

**A screenshot of a computer program

Description automatically generated**

**Copies of the four java programs are below and also included with this email.**

  /\*\*

 \* Semaphore.java

 \*

 \* A basic counting semaphore using Java synchronization.

 \*/

public class Semaphore

{

   private int value;

   public Semaphore(int value) {

      this.value = value;

   }

   public synchronized void acquire() {

      while (value <= 0) {

         try {

            wait();

         }

         catch (InterruptedException e) { }

      }

      value--;

   }

   public synchronized void release() {

      ++value;

      notify();

   }

}

    /\*\*

 \* utilities for causing a thread to sleep.

 \* Note, we should be handling interrupted exceptions

 \* but choose not to do so for code clarity.

 \*/

public class SleepUtilities

{

    /\*\*

     \* Nap between zero and NAP\_TIME seconds.

     \*/

    public static void nap() {

        nap(NAP\_TIME);

    }

    /\*\*

     \* Nap between zero and duration seconds.

     \*/

    public static void nap(int duration) {

            int sleeptime = (int) (NAP\_TIME \* Math.random() );

            try { Thread.sleep(sleeptime\*1000); }

            catch (InterruptedException e) {}

    }

    private static final int NAP\_TIME = 5;

}

  /\*\*

 \* Factory.java

 \*

 \* This class creates the reader and writer threads and

 \* the database they will be using to coordinate access.

 \*/

public class Factory

{

   public static void main(String args[])

   {

      Semaphore [] chopsticks;

      Philosopher [] philos;

      chopsticks = new Semaphore[5];

      philos = new Philosopher[5];

      for(int i = 0;i < 5;i++)

         chopsticks[i] = new Semaphore(1);

      for(int i = 0;i < 5;i++)

         philos[i] = new Philosopher(chopsticks[i],chopsticks[(i+1)%5],Integer.toString(i));

      for(int i =0;i < 5;i++)

        philos[i].start();

   }

}

  /\*\*

 \*

 \*

 \* This thread is used to demonstrate the operation of a semaphore.

 \*

 \* Figure 6.8

 \*

 \* @author Gagne, Galvin, Silberschatz

 \* Operating System Concepts with Java - Eighth Edition

 \* Copyright John Wiley & Sons - 2010.

 \*/

public class Philosopher extends Thread

{

    private Semaphore LStick,RStick;

    private String name;

    public Philosopher(Semaphore sem1,Semaphore sem2, String name)

    {

        this.name = name;

        LStick = sem1;

        RStick = sem2;

    }

    public void run()

    {

        while (true)

        {

            hungry();

            LStick.acquire();

            RStick.acquire();

            eating();

            LStick.release();

            RStick.release();

            thinking();

        }

    }

    private void eating()

    {

        System.out.println("Philosopher " + name + " is EATING.");

        SleepUtilities.nap(2);

    }

    private void thinking()

    {

        System.out.println("Philosopher " + name + " is THINKING.");

        SleepUtilities.nap(2);

    }

    private void hungry()

    {

        System.out.println("Philosopher " + name + " is HUNGRY.");

        SleepUtilities.nap(2);

    }

}

**JAVA CODE ANSWERS**

1. **Write a java program that starts a thread. (Sources:** [**https://www.geeksforgeeks.org/java-program-to-create-a-thread/**](https://www.geeksforgeeks.org/java-program-to-create-a-thread/) **& Assignment5 code)**

public class Test2Num1 extends Thread

{

public void run()

{

System.out.println("Test2 - 1. This program starts a thread in Java");

System.out.println("Thread Started");

}

public static void main(String[] args)

{

Test2Num1 thread = new Test2Num1();

thread.start();

}

}

**A screen shot of a computer

Description automatically generated**

1. **Given the four java files, compile and run as is.**

A screen shot of a computer

Description automatically generated

What I notice in this code is that it indefinitely runs, it may stop at some point, but at least as far as I’m comfortable leaving it running, it runs indefinitely. The philosophers get hungry first, eat, then think. However, it is not always in order. It starts in an orderly fashion, but then the philosophers start to finish eating and thinking earlier than others. However, it does appear that speeding through earlier than the other philosophers causes a waiting period for the early philosopher.

1. **Longer thinking() and shorter eating()**

private void eating()

{

System.out.println("Philosopher " + name + " is EATING.");

SleepUtilities.nap(1);

}

private void thinking()

{

System.out.println("Philosopher " + name + " is THINKING.");

SleepUtilities.nap(3);

}

private void hungry()

{

System.out.println("Philosopher " + name + " is HUNGRY.");

SleepUtilities.nap(2);

}

**A screen shot of a computer

Description automatically generated**

I notice that at least in the first passthrough, all of the hungry, eating, and thinking is done before anyone is hungry again. I changed the numbers around to be more drastic a couple other times, but I still couldn’t tell much difference happening. I noticed that the program would freeze up a bit at points when I changed the nap times to like 10 or more. Other than that though, I can’t notice any patterns in the data. The first pass through always looks like a pattern, but then after the first pass through, the order falls apart. I will change the data more drastically on the next question.

1. **Opposite changes. Longer eating(), shorting thinking().**

private void eating()

{

System.out.println("Philosopher " + name + " is EATING.");

SleepUtilities.nap(5);

}

private void thinking()

{

System.out.println("Philosopher " + name + " is THINKING.");

SleepUtilities.nap(1);

}

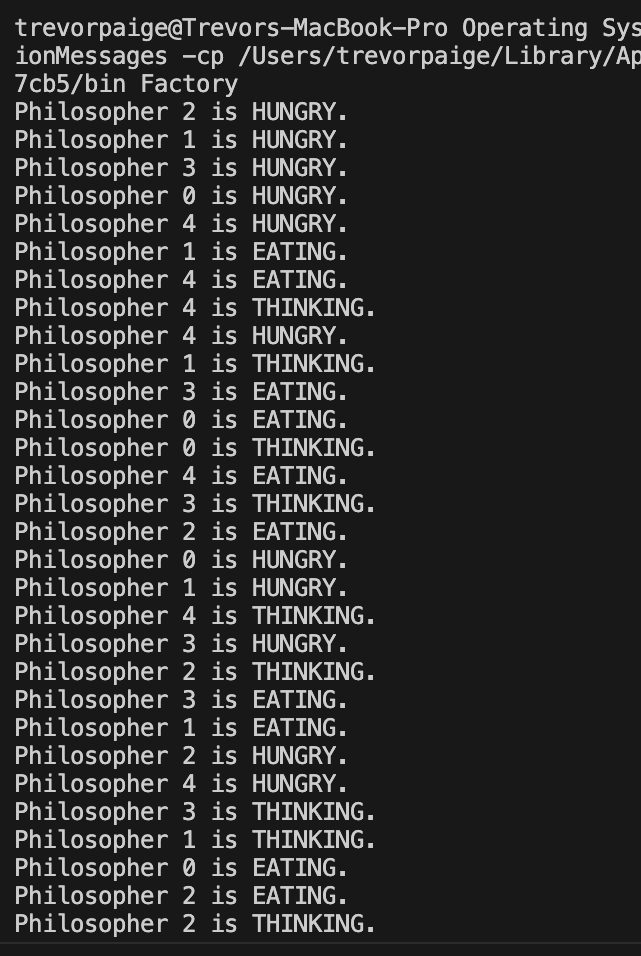
private void hungry()

{

System.out.println("Philosopher " + name + " is HUNGRY.");

SleepUtilities.nap(2);

}



Again, we look at the data and all goes well at the start, however, since I gave this one more drastic number changes in the one I decided to post, I can tell a difference in the ordering of the processes. Thinking happens a lot more sporadically. I think this may be because of the shorter time, so it processes the thinking rather quickly while the others are eating and getting hungry.

1. **Edit data and create ArrayLists (Sources:** [**https://www.geeksforgeeks.org/arraylist-in-java/**](https://www.geeksforgeeks.org/arraylist-in-java/) **&** [**https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)**)**

import java.util.ArrayList;

/\*\*

\* Factory.java

\*

\* This class creates the reader and writer threads and

\* the database they will be using to coordinate access. \*/

public class Factory

{

public static void main(String args[])

{

//Semaphore [] chopsticks;

ArrayList<Semaphore> chopsticks = new ArrayList<Semaphore>();

//Philosopher [] philos;

ArrayList<Philosopher> philos = new ArrayList<Philosopher>();

//chopsticks = new Semaphore[5];

chopsticks = new ArrayList<Semaphore>(10);

//philos = new Philosopher[5];

philos = new ArrayList<Philosopher>(5);

for(int i = 0;i < 10;i++)

//chopsticks[i] = new Semaphore(1);

chopsticks.add(new Semaphore(1));

for(int i = 0;i < 5;i++)

//philos[i] = new Philosopher(chopsticks[i],chopsticks[(i+1)%5],Integer.toString(i));

philos.add(new Philosopher(chopsticks.get(i), chopsticks.get((i + 1) % 5), Integer.toString(i)));

for(int i =0;i < 5;i++)

//philos[i].start();

philos.get(i).start();

}

}

A screenshot of a computer screen

Description automatically generated

For this last question, I had to refresh myself on ArrayList notations a bit. After looking at the oracle docs and a geeksforgeeks page though, it was pretty easy to setup. I believe I set it up correctly in terms of giving more semaphores than philosophers. Something I noticed in doing this run was that there would be very long pauses between some of the processes. It would start fine, run about 6-7 entries, and then pause for a bit, then it would run fine for 6-7 more entries and pause again. I noticed more pausing or “locking up” when I made these changes. Overall, it is still difficult for me to make conclusions on exactly what is going on, if the data was a set pattern each run I would feel more confident in giving a better conclusion, but every run the data changes. All I can notice and take note of are the times when the program pauses/freezes for a bit before continuing. One thing I can note though is that for this run, the philosopher’s did step over each other at times. Sometimes one philosopher would get hungry, eat, think and get hungry again before another philosopher finished their first loop.