First Name: Ludvig

Last Name: Magnusson

E-mail: lm222ix@student.lnu.se

Dining Philosophers

Solution

To display my solutions i will first divide this chapter into sections, one section for each requirement.

Providing average values of Thinking, Eating and Hungry.

Currently i print the average times in each state at the end of the simulation. It's hard to be sure that it is actually 100% accurate but as far as i can tell the time is at least close to accurate.

Measuring the time for thinking and eating was quite easy, since the time they spend doing these things is randomed by the code. The code snippet below is from my think() method, as you can see i add the sleep time to a list to save it for statistics, then the thread sleeps for exactly that time. This method is used for Eating as well.

```
private void think() throws InterruptedException {
 ...
        int sleeptime = (1+random.nextInt(30))*sleepTimeBound;
        thinkRecords.add(sleeptime);
        Thread.sleep(sleeptime);
...
    }
```

However measuring Hungry state is a little bit harder. For this one i try to measure the time between when the philosopher stops being hungry, to when he starts eating. See code snippet below.

```
public void run() {
        ...
                think();
                //Start measure time for hungry timing.
                long startMeasureTime = System.currentTimeMillis();
                ...
                     //Stop measure time for hungry timing.
                       long endMeasureTime = System.currentTimeMillis();
                       //Calc diffrence
                       int res = ((int) (endMeasureTime - startMeasureTime));
                            hungryRecords.add(res);     //add time hungry to list.
                            eat();      //Eat when chopsticks aquired.
```

```
                                        . . .
        }
```

In the table below i will display an example of the average times in each state. I ran the simulation for a pretty long time to get good results. The simulation was ran with 300ms as the sleepTimeBound. This number will be multiplied by 1-30 wich is randomed. So each wait/eat will range between 300ms and 9seconds.

Table showing average time in ms spent in each state.

State	Philosopher #0	Philosopher #1	Philosopher#2	Philosopher #3	Philosopher #4
Thinking	4260ms	4460ms	4831ms	4931ms	3555ms
Eating	4910ms	5460ms	5083ms	4481ms	4863ms
Hungry	3222ms	6369ms	3340ms	5777ms	4312ms

Table showing number of times each action preformed.

State	Philosopher #0	Philosopher #1	Philosopher#2	Philosopher #3	Philosopher #4
Thinking	20	15	19	16	20
Eating	19	15	18	16	19
Hungry	19	15	18	16	19

Running the simulation with the 300ms sleepTimeBound should make the philosophers eat and think for on average 4350ms. To get the data in the tables i ran the simulation for about 4 minutes, the longer you run it the closer to 4350 the average should be. Judging from the results in my tables it seems that the average times are measured correctly, or at least close to correctly.

The time spent hungry is harder to evaluate. From the second table you can derive that no philosopher has ate more times than it has been hungry, wich is good.

I added up all the time for philosopher #0 (4260*20 + 4910*19 + 3222*19)/1000 to seconds /60 to minutes) and it equals to almost exactly 4 minutes wich tells me that the time measurements are pretty accurate.

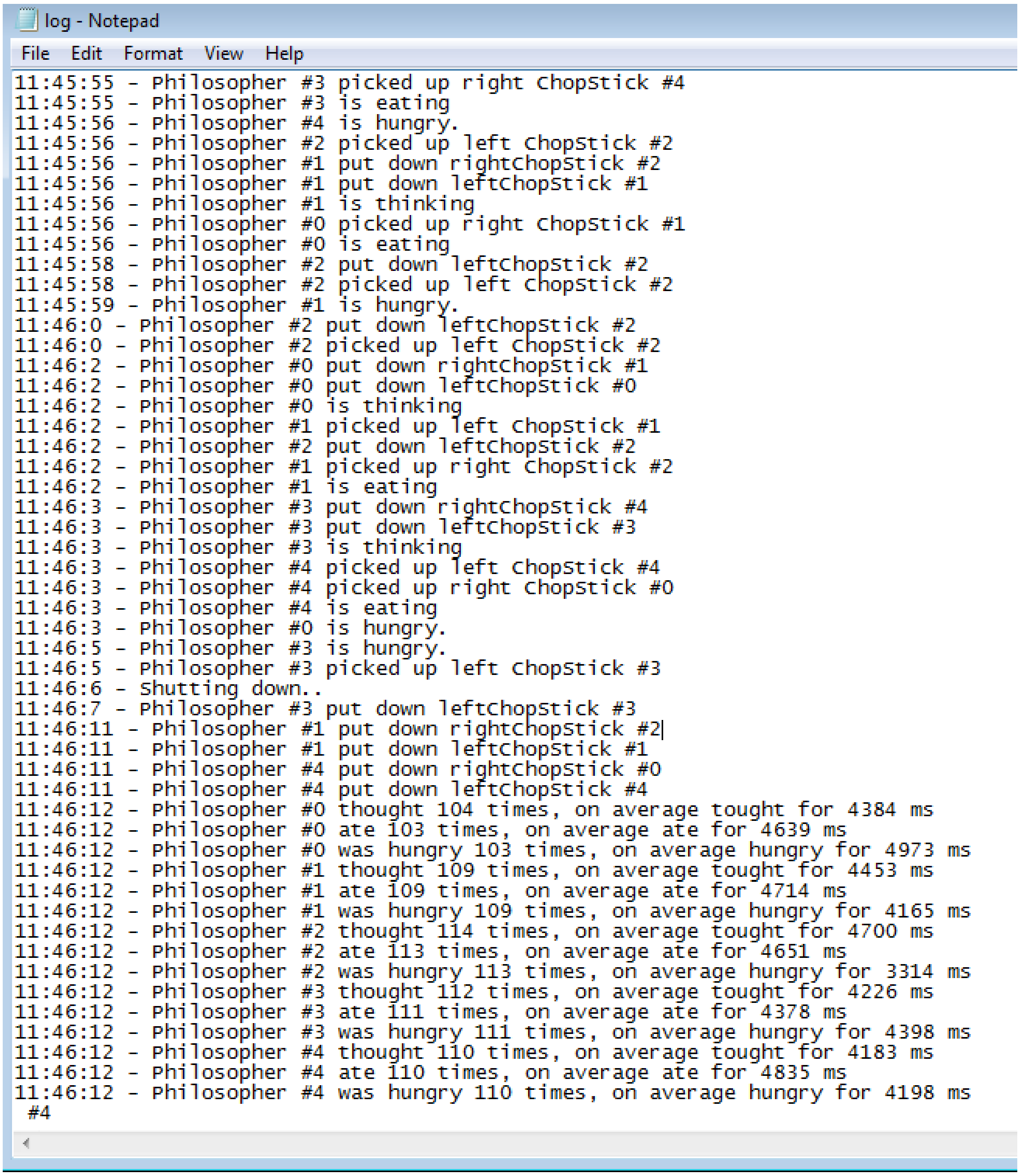
Generate a trace file with major events that occur during the simulation.

When running the program by using the .jar file a file called "log.txt" will be created in the same directory. That file will contain most major events in the simulation. I've done this by the class Logger that prints to both console and log file by default.

The code below shows the log(message) method. This is a good example of where to use Synchronized. Since this method will be used by all philosopher threads it happens quite a lot that they use the method simultaneously. The result without synchronized will be that the log file gets printed wrong, for example two messages would be printed with out newline in between quite often.

```
    public synchronized void log(String message) {
        time.setTime(System.currentTimeMillis());
        if (log) {
            out.write(time.getHours() + ":" + time.getMinutes() + ":" +
time.getSeconds() + " - " + message);
            out.println();
        }
        if (console) {
            System.out.println(message);
        }
    }
    }
```

example output in log.txt



```
11:45:55 - Philosopher #3 picked up right ChopStick #4
11:45:55 - Philosopher #3 is eating
11:45:56 - Philosopher #4 is hungry.
11:45:56 - Philosopher #2 picked up left ChopStick #2
11:45:56 - Philosopher #1 put down rightChopStick #2
11:45:56 - Philosopher #1 put down leftChopStick #1
11:45:56 - Philosopher #1 is thinking
11:45:56 - Philosopher #0 picked up right ChopStick #1
11:45:56 - Philosopher #0 is eating
11:45:58 - Philosopher #2 put down leftChopStick #2
11:45:58 - Philosopher #2 picked up left ChopStick #2
11:45:59 - Philosopher #1 is hungry.
11:46:0 - Philosopher #2 put down leftChopStick #2
11:46:0 - Philosopher #2 picked up left ChopStick #2
11:46:2 - Philosopher #0 put down rightChopStick #1
11:46:2 - Philosopher #0 put down leftChopStick #0
11:46:2 - Philosopher #0 is thinking
11:46:2 - Philosopher #1 picked up left ChopStick #1
11:46:2 - Philosopher #2 put down leftChopStick #2
11:46:2 - Philosopher #1 picked up right ChopStick #2
11:46:2 - Philosopher #1 is eating
11:46:3 - Philosopher #3 put down rightChopStick #4
11:46:3 - Philosopher #3 put down leftChopStick #3
11:46:3 - Philosopher #3 is thinking
11:46:3 - Philosopher #4 picked up left ChopStick #4
11:46:3 - Philosopher #4 picked up right ChopStick #0
11:46:3 - Philosopher #4 is eating
11:46:3 - Philosopher #0 is hungry.
11:46:5 - Philosopher #3 is hungry.
11:46:5 - Philosopher #3 picked up left ChopStick #3
11:46:6 - Shutting down..
11:46:7 - Philosopher #3 put down leftChopStick #3
11:46:11 - Philosopher #1 put down rightChopStick #2
11:46:11 - Philosopher #1 put down leftChopStick #1
11:46:11 - Philosopher #4 put down rightChopStick #0
11:46:11 - Philosopher #4 put down leftChopStick #4
11:46:12 - Philosopher #0 thought 104 times, on average tought for 4384 ms
11:46:12 - Philosopher #0 ate 103 times, on average ate for 4639 ms
11:46:12 - Philosopher #0 was hungry 103 times, on average hungry for 4973 ms
11:46:12 - Philosopher #1 thought 109 times, on average tought for 4453 ms
11:46:12 - Philosopher #1 ate 109 times, on average ate for 4714 ms
11:46:12 - Philosopher #1 was hungry 109 times, on average hungry for 4165 ms
11:46:12 - Philosopher #2 thought 114 times, on average tought for 4700 ms
11:46:12 - Philosopher #2 ate 113 times, on average ate for 4651 ms
11:46:12 - Philosopher #2 was hungry 113 times, on average hungry for 3314 ms
11:46:12 - Philosopher #3 thought 112 times, on average tought for 4226 ms
11:46:12 - Philosopher #3 ate 111 times, on average ate for 4378 ms
11:46:12 - Philosopher #3 was hungry 111 times, on average hungry for 4398 ms
11:46:12 - Philosopher #4 thought 110 times, on average tought for 4183 ms
11:46:12 - Philosopher #4 ate 110 times, on average ate for 4835 ms
11:46:12 - Philosopher #4 was hungry 110 times, on average hungry for 4198 ms
 #4
```

Provide the means (buttons) to start and end the simulation

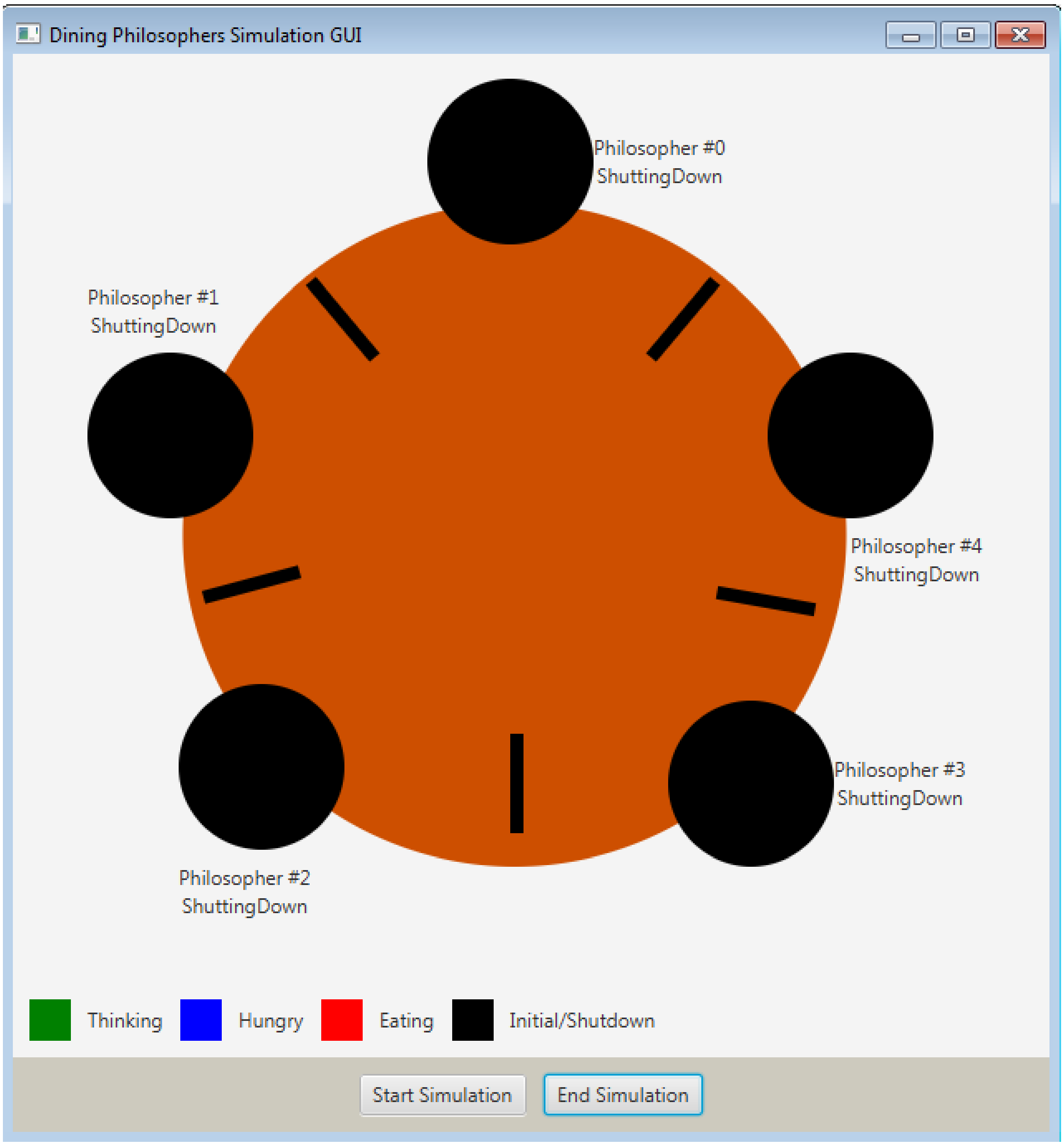
I have two buttons in my GUI called start and stop, simple enough. Code below.

```
Button startButton = new Button("Start Simulation");
        startButton.setOnAction...
                if (executer.isTerminated()) {
                    executer = Executors.newFixedThreadPool(numberOfPhilosophers);
                }
                runPhilosopherThreads();
    ...
```

```
        //End button, shuts down sim.
        Button endButton = new Button("End Simulation");
        endButton.setOnAction(...
                shutdownSimulation();
            ...
```

Provide a graphical animation and show graphically the status of all philosophers and chopsticks

Here is a pictureof what my GUI looks like.



To display my philosophers and chopsticks in the GUI i have extended the classes with Circle and Rectangle respectively. Circle and Rectangle are javafx libraries. This allows me to use methods like .setFill directly on a philosopher object to change the circles color, which makes it very convenient.

All coloring of philosophers and chopsticks are done by the philosopher threads themselves, in the method setState(code below).

```
 public void setState(STATE state) {
        STATE previousState = this.currentState;
        if(previousState.equals(STATE.Eating)) {    //If previous state was eating
the forks should now be black or "free"
            left.setFill(Paint.valueOf("BLACK"));
            right.setFill(Paint.valueOf("BLACK"));
        }
        if(state.equals(STATE.Eating)) {
            setFill(Paint.valueOf("RED"));
            moveChopsticks(); //paints chopsticks RED to show theyre Taken.
        } else if(state.equals(STATE.Hungry)) {
            setFill(Paint.valueOf("BLUE"));
        }else if(state.equals(STATE.Thinking)) {
            setFill(Paint.valueOf("GREEN"));
        } else if(state.equals(STATE.ShuttingDown)) {
            setFill(Paint.valueOf("BLACK"));
        } else if(state.equals(STATE.Initial)) {
            setFill(Paint.valueOf("BLACK"));
        } else{
            setFill(Paint.valueOf("ORANGE"));   //Orange is unknown, should never
happen.
        }
        this.currentState = state;}
```

As you can see i also display a couple of Labels in my GUI. The first label displays the philosopher ID to keep track of who is who, and the second label represents the state of that philosopher. To keep those labels updated constantly i use an AnimationTimer. Starting an animationtimer in the javafx thread will allow you to do updates once every rendered frame. The code example below displays how i update my labels with an AnimationTimer.

```
private void runPhilosopherThreads() {
//Code up here starts philosopher threads
...
    ...
        new AnimationTimer() {
            @Override
            public void handle(long now) {
                updateLabels();
            }
        }.start();    //Start animationTimer wich calls updateLabels every frame.
    }
    private void updateLabels() {
        for(philosopherContainer pc : pcs) {
            //Get current State and set it to Label
            pc.stateLabel.setText(pc.owner.getState().toString());
        }
    }
```

Describe how the deadlock detection has been addressed.

To prevent deadlock my chopsticks hold a reentrantlock wich will be locked by a philosopher who eats then unlocked when he is done eating. To aquire the lock i use tryLock(2 seconds) wich means if the lock is taken already the philosopher will try for 2 seconds. If the lock is not aquired during those seconds, the the philosopher will come back to it shortly after and try again. This is happening in a while(philosopher is hungry) loop, so once a philosopher gets hungry it will stay hungry until it gets to eat. See the code below to get a clearer image of what i mean. Note that the code here is simplified, for full code see the source code i sent.

run() method in Philosopher class.

```
 public void run() {
        try {
            while(!stopPhilosopher) {
                think();
                while(currentState.equals(STATE.Hungry)) {
                    if(left.pickUpChopStick(this, "left")) {
                        if(right.pickUpChopStick(this, "right")) {
                            eat();      //Eat when chopsticks aquired.
                            right.putDownChopStick(this, "right");
                        }
                        left.putDownChopStick(this, "left");
                    }
                }
            }
   ...
```

pickUpChopStick() method in Chopstick class

```
public boolean pickUpChopStick(Philosopher whoPickedUp, String leftOrRight) {
        if(!whoPickedUp.getState().equals(Philosopher.STATE.ShuttingDown)) {
            try {
      //Try to aquire the lock, if not avalible it will try for 2 seconds time.
                if (lock.tryLock(2000, TimeUnit.MILLISECONDS)) {
                    logger.log(whoPickedUp + " picked up " + leftOrRight + " " +
this);
                    return true;
                } else
        return false;
     }
```

Now that ive showed the code you might notice that while a philosopher is hungry it will iterate through the loop come to the tryLock() method over and over again, and it may seems like this will lock the chopstick forever. It accualy never locks the chopstick and this is due to the very short time between trying to lock, stopping, then coming back to trying to lock. During this time another philosopher who is currently in the trylock(2sec) will have time to try to aquire it.

I realise this solution might not be optimal, but it works pretty well in practice. It also makes my philosophers stay hungry infinetly until they get to eat wich i think is good.

Instructions for running and testing the application

There is a jarfile generated by Maven to run application. its located in DiningPhilosophers/target.

If the .jar doesn't work for some reason import the classes in “src\main\java\DiningPhilosophers” to a project in an java IDE of your choice, then run the program by running the class called main.

Summary, TODO and known bugs.

There is a couple of bugs in my program that i am aware of. The biggest one is probably the GUI freezing when you click the end simulation button. The GUI will un-freeze once the philosopher threads has shut down. The second one is that on rare occasions for a reason i havent figured out the color of a philosopher will stay at green color despite the state changing.

Overall i have enjoyed this assignment alot and i feel like i accomplished most of the goals. Things i want to implement until next deadline: 1. Preform any chages suggested in the feedback. 2. Show the average time spent in each state in the GUI. Preferably in the form of a table, updating "live" as you run the simulation. 3. Fix GUI freeze bug 4. Fix the color getting stuck bug.