PARALLEL PROGRAMMING (2)

CSC2002S

AUGUST 26, 2016

STEFFI BAUMGART

BMGSTE001

Introduction

The aim of the assignment is to run a simulation of a party in a room of a specified size, where guests of specified number arrive at random times through a single entrance door. On entering the room (which is represented as a grid), a guest will head to the refreshments bar where they have at least one drink, and then mingle with the other guests until they leave at either of the 2 exit doors.

Requirements

The lecturer provided 8 classes, namely PersonMover.class, Person.class, PeopleCounter, RoomPanel.class, GridBlock.class, PartyApp.class, CounterDislay.class and RoomGrid.class. PersonMover extended Thread and RoomPanel implemented Runnable (as well as extending Jpanel, as it handled the Graphic User Interface.) The majority of the code was written/edited in PartyApp and PersonMover.

The program must take in the following as command-line parameters:

I. the number of people invited to the venue
II. the length of the room
III. the breadth of the room
IV. (optional) the number of people allowed into the venue. (If not supplied, no limit is assumed.)

The location of entrance, exit doors and the bar were hard-coded.

Basic Rules for the Simulation

1. No guests must arrive before the start of the party.
   Location: PartyApp for the first section of code, and the second lies in PersonMover.When the start button is pressed on the GUI, the Action Event in PartyApp is executed which immediately stats creating the threads (PersonMover) and begins the simulation.In PersonMover, a CyclicBarrier is used to allow a set of threads (number of people) to all wait for each other to reach a common barrier point, before being allowed into the entrance door.

```
JButton startB = new JButton("Start");;

// add the listener to the jbutton to handle the "pressed" event
startB.addActionListener(new ActionListener() {
boolean started=false;
public void actionPerformed(ActionEvent e)  {
if (!started) {
started=true;
w.done=false;
PersonMover.done=false;
done =false;

createThreads();

//start threads one at a time
for (int i=0;i<noPeople;i++) {
personMover[i].start();} }
}
 });
```

***

```
PersonMover( Person creature, PeopleCounter score, CyclicBarrier bar, BlockingQueue bq) {
this(creature);
this.counter=score;
this.bar = bar;
this.bq = bq;
}

...

public void run() {
//TODO - add code so threads wait until other threads are ready before starting
//TODO add code to allow for pausing

try {
bar.await();
```

2. After the start of the party, people arrive at random times.

Location: PersonMover in the run() method. This means that after the thread is created, it sleeps for a random amount of time before the peopleOutSide variable in PeopleCounter increments and the "Waiting:" textbox on the GUI updates.

```
while(true){
sleep(rand.nextInt(10000)); //time till arriving at party
if(!PartyApp.paused){
break; } }
counter.personArrived(); //add to counter
```

3. Guest enter only through the entrance door.

Location: The getEntrance() method is situated in RoomGrid and is called from PersonMover. The threads are only allowed to enter through firstBlock, which is synchronized to only let through one thread at a time.

```
public int [] getEntrance() { //hard coded entrance
int [] coords = new int [] {getMaxX()/2,0};
return coords;
}

GridBlock firstBlock = grid.getEntranceBlock(); //enter through entrance
assert (firstBlock != null);
synchronized (firstBlock) {
firstBlock.waitBlock(); //threads should wait until first block is free
bq.take();
thisPerson.initBlock(firstBlock);
counter.personEntered(); //add to counter
```

4. After arrival, guests must wait until the entrance door is free (unoccupied) before entering.

Location: PersonMover. This ensures that no other threads (ie people) try to move to the entrance block so that others can arrive. This is done in the mingle() method by recalling getBlock() with random parameters until the entrance block is not chosen as the next position to move to.

```
//People must NOT go to the entrance block
while(nextBlock== grid.getEntranceBlock()){
nextBlock = grid.getBlock(rand.nextInt(3)-1+thisPerson.getX(),rand.nextInt(3)-1+thisPerson.getY());}
```

5. After entering the venue, guests must occupy a grid square, but no more than one square.

Location: The first method is called in PersonMover and invokes the second method, which is located in Person. If the person is able to move to the new block, it releases the lock on its current block, and the person is moves.

```
…
thisPerson.moveToBlock(nextBlock);
```

***

```
public  boolean moveToBlock(GridBlock newBlock)  {
if(newBlock.getBlock()!=false) {
currentBlock.releaseBlock();
initBlock(newBlock);
return true;
}
else  {
return false; }
}
```

6. People may not share grid squares (at most one person in each grid block).

Location: In Gridblock, the boolean occupied was changed to be an AtomicBoolean, to ensure that multiple threads can access the variable and it will remain thread-safe.

```
 private AtomicBoolean occupied;
occupied = new AtomicBoolean(false);
…
```

7. A person may not move into a grid square that is currently occupied.

Location: Gridblock. If the Person tries to move a position that is already occupied (AtomicBoolean occupied) then they they cannot move there as the value returned is false.

```
public boolean getBlock() {
if (occupied.get()==true) {
return false;
}
occupied.set(true);
return true;
}
```

8. People can move to any unoccupied grid square in the room, but must move one step at a time – they are not allowed to skip squares.

Location: PersonMover. This sets the x and y values of the next block chosen to be a random number between 0 and 2, so that it is a block adjacent to the current one the person is standing on .

```
while (nextBlock==null) { // repeat until hit on empty cell
x_mv= rand.nextInt(3)-1+thisPerson.getX();
if (x_mv<0) x_mv=0;
y_mv=rand.nextInt(3)-1+thisPerson.getY();
if (y_mv<0) y_mv=0;
if (!((x_mv==thisPerson.getX())&&(y_mv==thisPerson.getY()))) {
```

```
nextBlock=grid.getBlock(x_mv,y_mv);
```

9. Guest may only leave through an exit door.

Location: The first part is at the end of the Run() method in PersonMover, which calls the leave() method in Person. This ensures that the thread performs all activites required. A.) Consuming at least one drink. B.) Mingling with other guests, and then C.) Leaving the party. The second part is from GridBlock and checks if the block that they're on is the exit door.

```
thisPerson.drink();
} else headForRefreshments();
} else if (thisPerson.atExit()) {
thisPerson.leave();
} else {
mingle();
}
```

***

```
public   boolean atExit() { return currentBlock.isExit();}
```

10. Guests do no leave until after they have had a beverage.

Location: This is in PersonMover and is the first method that occurs. The try-and-catch at the bottom contains the condition to leave, as mentioned above, so it will only be executed afterwards.

```
private void headForRefreshments() {  //next move towards bar
//no need to change this function
int x_mv=-1;
int y_mv=-1;
GridBlock nextBlock =null;
while (nextBlock==null) { // repeat until hit on empty cell
...
```

11. The counters must keep accurate track of the people waiting to enter the venue, those inside, and those who have left.

Location: The entire PeopleCounter class updates the counts of the variables *peopleOutSide, peopleInside, peopleLeft*, provides synchronized accessors and mutators for each, and has other synchronized methods that are used in CounterDisplay.

```
synchronized public int getWaiting() {
return peopleOutSide;
}

synchronized public int getInside() {
return peopleInside;
}

synchronized public int getTotal() {
return (peopleOutSide+peopleInside+peopleLeft);
}
...
```

12. The pause button (not currently working) must pause/resume the simulation correctly on successive presses.
Location: PartyApp for the first section and PersonMover in the second. The PartyApp part handles mostly the aesthetics of the pause functions (changing from "Pause" to "Resume" and visa versa) while changing the boolean *paused*. This paused boolean is checked from within the Run() method of the PersonMover. The threads will sleep for a random amount of time until the the the program is unpaused, which results in the while loop being broken out of.

```
final JButton pauseB = new JButton("Pause ");;

// add the listener to the jbutton to handle the "pressed" event
pauseB.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
//TODO fill in code here to create
if (pauseB.getText().equals("Pause ")) {
pauseB.setText("Resume");
paused = true;
}
else {
pauseB.setText("Pause ");
paused = false;
}}
});
```

***

```
public void run() {
//TODO - add code so threads wait until other threads are ready before starting
//TODO add code to allow for pausing



try {
bar.await();

while(true){
sleep(thisPerson.getSpeed());
if(!PartyApp.paused){
break;
}

...

try {

while(!PartyApp.paused) {


if (thisPerson.thirsty()) {
if (thisPerson.atRefreshmentStation()) {
sleep(thisPerson.getSpeed() * 4);//drinking for a while

thisPerson.drink();
```

```
} else headForRefreshments();
} else if (thisPerson.atExit()) {
thisPerson.leave();
} else {
mingle();
}

sleep(thisPerson.getSpeed());
catch (Exception e) {
System.out.println("Thread "+this.ID + "interrrupted.");
done=true;
}
```

Additional Rules for the Simulation

1. People should enter in the order in which they arrived.
   Location: The first section comes from Run() in PersonMover and the latter is CreateThreads() in PartyApp. An ArrayBlockingQueue is used to control the order of the incoming guests (BlockingQueue bq = new ArrayBlockingQueue(noPeople,true);) which specifies the capacity(the number of PersonMover threads created) and the *true* boolean makes the elements processed in First In First Out (FIFO) order . The put() method inserts the specified element at the tail of this queue, waiting for space to become available if the queue is full, and the take() retrieves and removes the head, waiting if necessary until an element becomes available.

```
counter.personArrived(); //add to counter
bq.put(thisPerson);

GridBlock firstBlock = grid.getEntranceBlock(); //enter through entrance
assert (firstBlock != null);

synchronized (firstBlock) {
firstBlock.waitBlock(); //threads should wait until first block is free
bq.take();
```

***

```
public static void createThreads() {
score.resetScore();
//create people threads

//PeopleCounter is called "score"
CyclicBarrier bar = new CyclicBarrier(noPeople);

//if true, processed in FIFO order
BlockingQueue bq = new ArrayBlockingQueue(noPeople,true);


for (int i=0;i<noPeople;i++) {
personMover[i] = new PersonMover(persons[i],score, bar, bq);
}
```