אוניברסיטת בן-גוריון

מדור בחינות

מספר נבחן:_____

רשמו תשובותיכם בגיליון התשובות בלבד
תשובות מחוץ לגיליון לא יבדקו.

בהצלחה!

תאריך הבחינה: 24.1.2011
שם המורה: פרופ' מיכאל אלחדד
ד"ר מני אדלר
ד"ר אנדרי שרף
שם הקורס: תכנות מערכות
מספר הקורס: 202-1-2031
מיועד לתלמידי: מדעי המחשב, הנדסת תוכנה
שנה: תשע"א
סמסטר: א'
מועד: א'
משך הבחינה: שלש שעות
חומר עזר: אסור

שאלה 1 (30 נקודות)

במקומות ציבוריים רבים (כמו לדוגמא, ברכבת התחתית המפורסמת של תל אביב) מותקנת מערכת של מדרגות נעות, לשם עליה וירידה בין מפלסים. בשאלה זו נעסוק בסימולציה של מנגנון זה.

המערכת מורכבת מהאובייקטים הפסיביים הבאים:

- גרם מדרגות (**StairCase**). על כל מדרגה יש מקום להולך רגל אחד.
- במפלס התחתון (בתחתית המדרגות) ובמפלס העליון (בראש המדרגות) יש מקום לכל הולכי הרגל
- קבוצה של הולכי רגל (**Pedestrian**), המצב הפנימי של הולך רגל כולל את הגובה שלו במרחב, ואסטרטגית התקדמות (**Strategy**).

הגובה של הולך הרגל, ניתן על ידי הגובה של המפלס התחתון ומספר המדרגות מהמפלס התחתון למקום בו הוא עומד. לדוגמא, אם גובה המפלס התחתון הוא 5, והולך הרגל נמצא 39 מדרגות מעל המפלס, נאמר כי גובהו הוא 44.

אסטרטגיית ההתקדמות, קובעת את התנהגות הולך הרגל כאשר הוא נמצא בגרם מדרגות.

קיימים שלשה מימושים לממשק **Strategy**, עם מתודת ההתקדמות **next()** המחזירה את הגובה החדש אליו יגיע הולך הרגל:

- **Relaxed** - הולך הרגל אינו עולה או יורד, אלא ממתין על המדרגה בה הוא עומד.
- **HurryUp** - הולך הרגל מנסה להתקדם עצמאית מעלה
- **HurryDown** - הולך הרגל מנסה להתקדם עצמאית מטה.

בנוסף מוגדרים במערכת שני סוגים של אובייקטים אקטיביים:

- Thread ('חוט') הרץ מעל המשימה **StairCaseMovement** על גרם מדרגות: הנעה כלפי מעלה של גרם המדרגות בקצב נתון.
- Thread ('חוט') הרץ מעל המשימה **PedestrianMovement** על הולך רגל: קידום הולך רגל במעלה או במורד גרם מדרגות בהתאם לאסטרטגית ההתקדמות שלו.

בסימולציה הנתונה, קיים גרם מדרגות אחד העולה מעלה, ושני הולכי רגל:

- הולך במפלס התחתון המנסה לעלות מעלה בנוסף להתקדמות המדרגות הנעות (אסטרטגיית **HurryUp**).
- הולך רגל במפלס העליון המנסה לרדת מטה נגד כיוון המדרגות הנעות (אסטרטגית **HurryDown**).

התבוננו בקטעי הקוד הממשים את המערכת וענו על השאלות שאחר כך:

```
interface Strategy {
  int next(int height);
}

class Relaxed implements Strategy {
  public  int next(int height) {
       return height;
  }

class HurryUp implements Strategy {
  public  int next(int height) {
    return height+1;
  }
}

class HurryDown implements Strategy {
  public  int next(int height) {
    return height-1;
  }
}
```

```
interface Pedestrian {
  int getHeight();
  void setHeight(int height);
  Strategy getStrategy();
  void advance(StairCase stairCase);
}

class Passenger implements  Pedestrian  {
  private int _height;
  private final Strategy _strategy;

  Passenger (int height, Strategy strategy)  {  _height  = height;  _strategy  = strategy; }

  public synchronized Strategy getStrategy() { return _strategy; }
  public synchronized int getHeight() { return _height; }
  public synchronized void setHeight(int height) { _height  = height; }
  public void advance(StairCase stairCase) {
    //@TODO
  }
}
```

```
interface StairCase {
  int  fromHeight();
  int toHeight();
  Pedestrian getPedestrian(int i);
  void setPedestrian(Pedestrian pedestrian ,int i);
  int size();
  int capacity();
}

class StairCaseImpl implements StairCase {

  private final Pedestrian[] _pedestrians;
  private final int _fromHeight;
  private final int _toHeight;

  StairCaseImpl(int fromHeight, int  toHeight) {
     _fromHeight = fromHeight;
     _toHeight = toHeight;
     _pedestrians  = new Pedestrian[_toHeight  - _fromHeight + 1] ;
  }

  public int  fromHeight () { return _fromHeight; }
  public int toHeight () { return _toHeight; }
  public int capacity() {  return _pedestrians.length;  }
  public synchronized int size() {
    int size=0;
    for (Pedestrian p : _pedestrians)
      if (p!=null)
        size++;
    return size;
  }

  public synchronized Pedestrian getPedestrian(int i) {
    return _pedestrians[i];
  }

  public synchronized void setPedestrian(Pedestrian p, int i) {
     _pedestrians[i] = p;
  }
}
```

```
class StairCaseMovementTask  implements Runnable {
```

```
  private final StairCase _staircase;
  private final long _speed;
  StairCaseMovementTask(StairCase staircase , long speed) {  _staircase = staircase ; _speed = speed; }
  public void run() {
    while (true) {
      try {
        for (int i=_staircase .capacity()-1; i>0; i--) {
            _staircase.setPedestrian(_staircase.getPedestrian(i-1),i);
            Pedestrian p = _staircase.getPedestrian(i);
            if (p!=null)
              p.setHeight(p.getHeight()+1);
          }
          _staircase.setPedestrian(null,0);
          synchromized(_staircase) { _staircase.notifyAll(); }
        Thread.sleep(_speed);
      } catch (InterruptedException e) {}
    }
  }
}

class PedestrianMovementTask implements Runnable {
  private final Pedestrian _pedestrian;
  private final StairCase _stairCase;
  private final long _speed;
  PedestrianMovementTask(Pedestrian pedestrian, long speed, StairCase stairCase) {
    _pedestrian = pedestrian; _speed = speed;  _stairCase = stairCase ; }
  public void run() {
    while (true) {
      try {
        _pedestrian.advance(_stairCase);
        Thread.sleep(_speed);
      } catch (InterruptedException e) {
        return;
      }
    }
  }
}
```

```
class Simulation {
  public static void main(String[] args) {
    StairCase upStairCase = new StairCaseImpl(1, 39);
    Pedestrian pedestrian1 = new Passenger(1,new HurryUp());
    Pedestrian pedestrian2 = new Passenger(39,new HurryDown());
    new Thread(new StairCaseMovementTask (upStairCase,1000)).start();
```

```
    new Thread(new PedestrianMovementTask (pedestrian1,1000, upStairCase)).start();
    new Thread(new PedestrianMovementTask (pedestrian2,1000, upStairCase)).start();
  }
}
```

א. הגדירו תכונה נשמרת (invariant) עבור הממשק **StairCase**, וציינו (ונמקו) האם המחלקה **StairCaseImpl** הממשת אותה בטוחה (במובן שמירה על האינווריאנטה של **StairCase**) תחת כל חישוב מקבילי [10 נקודות]

ב. הגדירו תנאי התחלה וסיום (pre/post conditions) עבור המתודה **advance()** בממשק **Pedestrian** - המקדמת את הולך הרגל בגרם המדרגות (הניתן כפרמטר) על פי האסטרטגיה שלו - וממשו את המתודה במחלקה **Passenger**. במידה וקיים הולך רגל במדרגה הבאה, יש להמתין עד שהמקום יתפנה. [12 נקודות]

ג. הראו כיצד ביטול הת'רד המבצע את **StairCaseMovementTask**, מביאה את הסימולציה למצב של חֶבֶק (deadlock). [8 נקודות]

```
class Simulation {
  public static void main(String[] args) {
    StairCase upStairCase = new StairCaseImpl(1, 39);
    Pedestrian pedestrian1 = new Passenger(1,new HurryUp());
    Pedestrian pedestrian2 = new Passenger(39,new HurryDown());
    new Thread(new StairCaseMovementTask (upStairCase,1000)).start();
    new Thread(new PedestrianMovementTask (pedestrian1,1000, upStairCase)).start();
    new Thread(new PedestrianMovementTask (pedestrian2,1000, upStairCase)).start();
  }
}
```

שאלה 2 (30 נקודות)

א. ממשו ב C++ את המחלקות **StairCase** ו **StairCaseImpl** משאלה 1.
יש לתמוך בהעברה **by value** של אובייקט מטיפוס **StairCaseImpl** ובביצוע השמה של ערך לאובייקט מסוג **StairCase** (= **copy constructor** ו **assignment operator** במחלקה **StairCaseImpl**, שאינן קיימת ב **Java**).
אין צורך לממש את הסנכרון. [18 נקודות]

ב. ציירו את תמונת הזיכרון, כאשר מריצים את **main** ומגיעים למקום המסומן בתוך קוד המתודה Q [12 נקודות]

```
void Q(const StairCase& staircase) {
  std::cout << "Mr. Memory, What are the 39 steps?";
  //@@ יש לצייר את תמונת הזיכרון כאשר הרצת הקוד מגיעה לכאן
}
void main() {
  StairCaseImpl upStairCase(1, 39);
  StairCase* downStairCase = new StairCaseImpl(1,39);
  StairCaseImpl upStairCase2 = upStairCase;
  Q(upStairCase2);  }
```

שאלה 3 (30 נקודות)

בישיבת הנהלה של החברה המפעילה את המדרגות הנעות הוחלט, כי לשם בקרה על המערכת, הנעת גרם המדרגות – כלומר ביצוע המשימה StairCaseMovementTask - תתבצע בהפעלה מרחוק מתוך תהליך אחר המפעיל את כל המדרגות הנעות של החברה, באתרים השונים.

להלן קוד של התהליך המפעיל ממוקד הבקרה גרם מדרגות הנמצא באתר מרוחק.
התהליך פונה ל rmiregistry על מנת לקבל stub לגרם המדרגות המוגדר בזיכרון של התהליך המריץ את הסימולציה בשאלה הראשונה (המחלקה Simulation בעמוד 4), ולאחר מכן מריץ על גרם מדרגות זה את המשימה StairCaseMovementTask.

```
public class StairCaseControl {
  public static void main(String[] args) {
    try {
        StairCase upStairCase = (StairCase)Naming.lookup("132.87.45.3:4004/StairCase1");
        new Thread(new StairCaseMovementTask (upStairCase,1000)).start();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

א. שנו את המחלקות Simulation, StairCase, ו StairCaseImpl משאלה 1, כך שיתאימו למערכת החדשה. [10 נקודות]
הערה: התעלמו מיישום מחדש של מנגנון wait/notify במחלקות האחרות (כלומר, אין צורך לממש את האופן בו מבוצע wait בזיכרון של תהליך אחד ו notify בזיכרון של תהליך שני). התמקדו בשינוי האופי של StairCase.

ב. לאחר כמה פעולות תקשורת יתקדמו המדרגות הנעות מעלה בשלושה שלבים? (ניתן להניח כי תנאי ההתחלה תמיד מתקיימים). פרטו. [7 נקודות]

ג. האם הקובץ StairCaseImpl.class צריך להיות מוגדר במערכת הקבצים של המחשב בו רץ התהליך StairCaseControl? נמקו. [3 נקודות]

ד. נתון כי מימוש תקשורת ה Skel של StairCase עם ה Stub, מתבסס על תבנית ה Reactor.
כזכור, מתודת ה **run()** במחלקה ProtocolTask מסונכרנת.
סטודנט בקורס תכנות מערכות הציע, לשם ייעול, להוריד את הסנכרון של המתודה, ותחת זאת לסנכרן רק את הבלוק בחלק השני, כמתואר בקוד שלפניכם (השינויים שביצע הסטודנט מודגשים):

```
class ProtocolTask implements Runnable {
  private final ServerProtocol _protocol;
  private final StringMessageTokenizer _tokenizer;
  private final ConnectionHandler _handler;
  /* The fifo queue, which holds data coming from the socket. Access to the queue is serialized, to ensure correct
  *processing order - even if more data is received by the reactor while previous data is still being processed.*/
  private final Vector<ByteBuffer> _buffers = new Vector<ByteBuffer>();
```

```
...
public synchronized void run() {
  // first, add all the bytes we have to the tokenizer
  synchronized (_buffers) {
    while(_buffers.size() > 0) {
      ByteBuffer buf = _buffers.remove(0);
      this._tokenizer.addBytes(buf);
    }
  }
  // now, go over all complete messages and process them.
  synchronized (this) {
    while (_tokenizer.hasMessage()) {
      String msg = _tokenizer.nextMessage();
      String response = this._protocol.processMessage(msg);
      if (response != null) {
        try {
          ByteBuffer bytes = _tokenizer.getBytesForMessage(response);
          this._handler.addOutData(bytes);
        } catch (CharacterCodingException e) { e.printStackTrace(); }
      }
    }
  }
}

// This is invoked by the ConnectionHandler which runs in the reactor main thread.
public void addBytes(ByteBuffer b) {
  synchronized (_buffers) { _buffers.add(b); }
}
}
```

האם השינוי שהציע הסטודנט עשוי לפגום בפעילותן התקינה של המדרגות הנעות, בסימולציה הנתונה? נמקו תשובתכם. [10 נקודות]

הנחיה: הבהירו לעצמכם אלו ת'רדים בשרת עובדים במקביל על חלקים שונים של המצב הפנימי ב ProtocolTask, ומתי זה קורה.
עימדו על הסיבה לסינכרון כל המתודה **run()** על **this**, איזה עיקרון בפרוטוקול זה בא להבטיח?

למעוניינים (זה לא בהכרח נדרש), ניתן למצוא בנספח (בסוף המבחן) את החלקים הרלבנטיים של המחלקה **ConnectionHandler**

שאלה 4 (10 נקודות)

לשם מעקב ובקרה אחר יעילות ההפעלה של המדרגות הנעות באתרים השונים, החליט זכיין הפעלה לשמור בבסיס נתונים, מידע סטטיסטי על המשתמשים במדרגות.
עבור כל גרם מדרגות:

- מיקום גרם המדרגות (שם הארץ, שם העיר, שם המתחם)
- אוריינטציה (עולה או יורד)
- רשימה של הימים בהם הוא הופעל
  - תאריך
  - מספר עוברים ביום זה
  - זמן ממוצע למעבר ממפלס למפלס

א. הגדירו מודל נתונים (טבלאות ומפתחות) עבור המערכת שתוארה לעיל. [5 נקודות]

ב. הגדירו שאילתת SQL המחזירה את כמות האנשים שעברה בכל אחד מגרמי המדרגות (שם המתחם ומספר העוברים) הממוקמים בב"ש, בתאריך 2/1/11, ממוין ע"פ מספר האנשים בסדר עולה. [5 נקודות]

```
public class ConnectionHandler {
  protected final SocketChannel _sChannel;
  protected final ReactorData _data;
  protected final AsyncServerProtocol _protocol;
  protected final StringMessageTokenizer _tokenizer;
  protected Vector<ByteBuffer> _outData;
  protected final SelectionKey _skey;
  private ProtocolTask _task;
  ...
   // Post data in the pending data queue, so that the connectionHandler will send it through the socket.
   // switchToReadWriteMode() subscribes this handler key to the OP_WRITE event
   // This event will immediately fire because the output buffer of the channel is empty.
   // It will keep firing as long as the output buffer is not filled.
   // When we are done sending pending data, we will unsubscribe from OP_WRITE.
  public synchronized void addOutData(ByteBuffer buf) {
    _outData.add(buf);
    switchToReadWriteMode();
  }

  // Reads incoming data from the client:. Reads some bytes from the SocketChannel
  // Create a protocolTask, to process this data, possibly generating an answer.
  // Inserts the Task to the ThreadPool
  public void read() {
    // Do not read if protocol has terminated. Only write of pending data is
    // allowed when the protocol asked to close the connection.
    if (_protocol.shouldClose())
      return;
    SocketAddress address = _sChannel.socket().getRemoteSocketAddress();
    logger.info("Reading from " + address);
    ByteBuffer buf = ByteBuffer.allocate(BUFFER_SIZE);
    int numBytesRead = 0;
    try {
      numBytesRead = _sChannel.read(buf);
    } catch (IOException e) {
      numBytesRead = -1;
    }
    if (numBytesRead == -1) {      // Is the channel closed?
      // No more bytes can be read from the channel
      logger.info("client on " + address + " has disconnected");
      closeConnection();
      // tell the protocol that the connection terminated.
     _protocol.connectionTerminated();
```

```
        return;
    }
    // Add the buffer to the protocol task
    buf.flip();
    _task.addBytes(buf);
    // Add the protocol task to the reactor which will parse and process the data
    // when a thread becomes available for it.
    _data.getExecutor().execute(_task);
  }

  // Attempts to send data to the client. if all the data has been succesfully sent, the ConnectionHandler will
  // automatically switch to read only mode, otherwise it will stay in its current mode (which is read / write).
  public synchronized void write() {
    if (_outData.size() == 0) { // if nothing left in the output string, go back to read mode
        switchToReadOnlyMode();
        return;
    }
    // If there is something to send - send the first byte buffer
    // We will return to this write() operation very soon because the selector will keep firing the OP_WRITE event
    // after we are done writing this buffer and check if there are more buffers to be sent.
    ByteBuffer buf = _outData.remove(0);
    if (buf.remaining() != 0) {
        _sChannel.write(buf);
        // Check if the buffer contains more data: we could not send all of the buffer in one write
        // (the output buffer of the socket got full). So we remember that there is more data to be sent.
        // We will receive a new OP_WRITE event when the output buffer of the socket
        // is not full anymore and complete the write operation then.
        if (buf.remaining() != 0)
            _outData.add(0, buf);
    }
    // Check if the protocol asked us to close this connection.
    // If it did, we remain open as long as there are pending data to be sent.
    // As soon as all the data has been sent, we can close the connection.
    if (_protocol.shouldClose()) {
        switchToWriteOnlyMode();
        if (buf.remaining() == 0) {
            logger.info("disconnecting client on " + _sChannel.socket().getRemoteSocketAddress());
            closeConnection();
        }
    }
  }
...
}
```