גיליון תשובה

מספר נקודות:____________________

<table>
<thead>
<tr>
<th>שאלה 1</th>
<th>(20 נקודות)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Q2</td>
</tr>
<tr>
<td>Q3</td>
<td>Q4</td>
</tr>
<tr>
<td>Q5</td>
<td>Q6</td>
</tr>
<tr>
<td>Q7</td>
<td>TOTAL</td>
</tr>
</tbody>
</table>

שאלה A (10 נקודות)

```java
class Fib {
    private int prev;
    private int curr;
    Fib() {
        prev = 0;
        curr = 1;
    }
    Fib(int p, int c) {
        prev = p;
        curr = c;
    }
    public int synchronized getValue() {
        return prev;
    }
}
```

שאלה B (10 נקודות)

```java
public synchronized boolean test() {
    int tmpPrev = prev;
    int tmpCurr = curr;
    int tmp;

    while (tmpPrev >= 0 && tmpCurr >= 0) {
        if (tmpPrev == 0 && tmpCurr == 1)
            return true;

        tmp = tmpCurr;
        tmpCurr = tmpPrev;
        tmpPrev -= tmp;
    }

    return false;
}
```
public class Summer {

    private int sum; // Automatically initialized to zero.

    public synchronized void up() {
        sum++;
        if (sum == 1)
            notifyAll();
        System.out.print(sum);
    }

    public synchronized void down() {
        while (sum == 0)
            wait();
        sum--;
        System.out.print(sum);
    }
}
Test
a1.sum = 1
a1.sum = 3
double a3 = 10

```cpp
#include <iostream>

class A {
    protected:
        int x_;
    private:
        A* a_;
    public:
        A(int x) : x_(x), a_(0) {}

        // Destructor
        virtual ~A() {}

        // Define setA
        virtual setA(A* a) { a_ = a; }

        // Define sum
        virtual int sum() {
            if (a_ == NULL)
                return x_;  
            else
                return (x_ + a_ -> sum());
        }

        // Define doubled
        virtual int doubled() {
            return (x_ + x_);
        }
    ;

class B : public A {
    protected:
        int y_;

    public:
        // Constructor
        B(int x, int y) : A(x), y_(y) {}

        // Destructor
        virtual ~B() {}

        // Define Doubled
        virtual int doubled() {
            return (2 * (x_ + y_));
        }
    ;

    void main() {
        std::cout << "Test" << std::endl;
    }
```
A* a1 = new A(1);
std::cout << "a1.sum = " << a1->sum() << std::endl;
A* a2 = new A(2);
a1->setA(a2);
std::cout << "a1.sum = " << a1->sum() << std::endl;
A* a3 = new B(2,3);
std::cout << "doubled a3 = " << a3->doubled() << std::endl;
delete a1;
delete a2;
delete a3;
progressor1.progress(remoteElement.3)

remoteElement.increase()
return

remoteElement.increase()
return

remoteElement.increase()
return

return

23
<table>
<thead>
<tr>
<th>Time Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model 4</strong>: The client sends a “clock read” request (an empty UDP message) to the server. Upon the request message arrival, the server reads its clock, and sends the client a (UDP) message that includes the value of the server’s clock. Message lost (requests or replays) are considered; for each server the client sets a timer for un-replayed requests. Upon timeout, client sends another request to the server.</td>
</tr>
<tr>
<td><strong>Model 1</strong>: The client connects to the server. Upon connection establishment the server reads its clock, and writes to the client the value of the server’s clock. Note that TCP may retransmit the message because of packet loss and therefore the client can read an out-of-date clock value.</td>
</tr>
<tr>
<td>We do not use the <strong>pipelining</strong> technique, because at any time there should be at most one request message.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Video Streaming Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model 4</strong>: The server sends the video stream to a multicast address. A client that wishes to receive the video stream joins the multicast group. The <strong>pipelining</strong> technique is used to buffer message so that the user receives a continuous video stream despite of message decoding.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model 3</strong>: A DBMS is both CPU and IO bounded. Therefore, we use different threads to process different tasks (e.g., disk, network, query processing). We use the <strong>pipelining</strong> technique extensively; incoming SQL commands are stored, a buffer stores outgoing results, priority queues schedule internal processing task. Note that, we use transactions (a synchronization technique) to can assure data consistency.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Web Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model 3</strong>: An http (the Internal World Wide Web protocol) server is IO bounded. Therefore, we use different threads to process different tasks (e.g., disk, network). We use the <strong>pipelining</strong> technique extensively; incoming http commands are stored, and buffers stores outgoing HTML files.</td>
</tr>
</tbody>
</table>
### Table: Movies

<table>
<thead>
<tr>
<th>Primary Key: Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Director</td>
</tr>
<tr>
<td>Year</td>
</tr>
<tr>
<td>Length</td>
</tr>
</tbody>
</table>

### Table: Schedule

<table>
<thead>
<tr>
<th>Primary Key: Data+Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
</tr>
<tr>
<td>Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Foreign Key: MovieId</th>
</tr>
</thead>
<tbody>
<tr>
<td>MovieId</td>
</tr>
</tbody>
</table>

SELECT Schedule.Date, Schedule.Time, Movies.Id, Movies.Name, Movies.Director, Movies.Year, Movies.Length
FROM Movies, Schedule
WHERE Movies.Id = Schedule.MovieId AND Schedule.Date = '1/1/2003'
ORDER BY Schedule.Date, Schedule.Time