14

Solutions to Databases

Questions 1 through 3 refer to the following database schema:

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)

Note that each apartment can have multiple tenants, and each tenant can have multiple apartments. Each apartment belongs to one building, and each building belongs to one complex.

14.1 Multiple Apartments: Write a SQL query to get a list of tenants who are renting more than one apartment.

pg 172

SOLUTION

To implement this, we can use the HAVING and GROUP BY clauses and then perform an INNER JOIN with Tenants.

```
SELECT TenantName
FROM Tenants
INNER JOIN
   (SELECT TenantID FROM AptTenants GROUP BY TenantID HAVING count(*) > 1) C
ON Tenants.TenantID = C.TenantID
```

Whenever you write a GROUP BY clause in an interview (or in real life), make sure that anything in the SELECT clause is either an aggregate function or contained within the GROUP BY clause.

14.2 Open Requests: Write a SQL query to get a list of all buildings and the number of open requests (`Requests` in which `status` equals 'Open').

pg 173

SOLUTION

This problem uses a straightforward join of `Requests` and `Apartments` to get a list of building IDs and the number of open requests. Once we have this list, we join it again with the `Buildings` table.

```
SELECT BuildingName, ISNULL(Count, 0) as 'Count'
FROM Buildings
LEFT JOIN
  (SELECT Apartments.BuildingID, count(*) as 'Count'
   FROM Requests INNER JOIN Apartments
   ON Requests.AptID = Apartments.AptID
   WHERE Requests.Status = 'Open'
   GROUP BY Apartments.BuildingID) ReqCounts
ON ReqCounts.BuildingID = Buildings.BuildingID
```

Queries like this that utilize sub-queries should be thoroughly tested, even when coding by hand. It may be useful to test the inner part of the query first, and then test the outer part.

14.3 Close All Requests: Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

pg 173

SOLUTION

`UPDATE` queries, like `SELECT` queries, can have `WHERE` clauses. To implement this query, we get a list of all apartment IDs within building #11 and the list of update requests from those apartments.

```
UPDATE Requests
SET Status = 'Closed'
WHERE AptID IN (SELECT AptID FROM Apartments WHERE BuildingID = 11)
```

14.4 Joins: What are the different types of joins? Please explain how they differ and why certain types are better in certain situations.

pg 173

SOLUTION

`JOIN` is used to combine the results of two tables. To perform a `JOIN`, each of the tables must have at least one field that will be used to find matching records from the other table. The join type defines which records will go into the result set.

Let's take for example two tables: one table lists the "regular" beverages, and another lists the calorie-free beverages. Each table has two fields: the beverage name and its product code. The "code" field will be used to perform the record matching.

Regular Beverages:

Name	Code
Budweiser	BUDWEISER
Coca-Cola	COCACOLA

Name	Code
`Pepsi`	PEPSI

Calorie-Free Beverages:

Name	Code
`Diet Coca-Cola`	COCACOLA
`Fresca`	FRESCA
`Diet Pepsi`	PEPSI
`Pepsi Light`	PEPSI
`Purified Water`	Water

If we wanted to join `Beverage` with `Calorie-Free Beverages`, we would have many options. These are discussed below.

- `INNER JOIN`: The result set would contain only the data where the criteria match. In our example, we would get three records: one with a `COCACOLA` code and two with `PEPSI` codes.
- `OUTER JOIN`: An `OUTER JOIN` will always contain the results of `INNER JOIN`, but it may also contain some records that have no matching record in the other table. `OUTER JOIN`s are divided into the following subtypes:
    - `LEFT OUTER JOIN`, or simply `LEFT JOIN`: The result will contain all records from the left table. If no matching records were found in the right table, then its fields will contain the `NULL` values. In our example, we would get four records. In addition to `INNER JOIN` results, `BUDWEISER` would be listed, because it was in the left table.
    - `RIGHT OUTER JOIN`, or simply `RIGHT JOIN`: This type of join is the opposite of `LEFT JOIN`. It will contain every record from the right table; the missing fields from the left table will be `NULL`. Note that if we have two tables, A and B, then we can say that the statement `A LEFT JOIN B` is equivalent to the statement `B RIGHT JOIN A`. In our example above, we will get five records. In addition to `INNER JOIN` results, `FRESCA` and `WATER` records will be listed.
    - `FULL OUTER JOIN`: This type of join combines the results of the `LEFT` and `RIGHT JOINS`. All records from both tables will be included in the result set, regardless of whether or not a matching record exists in the other table. If no matching record was found, then the corresponding result fields will have a `NULL` value. In our example, we will get six records.

14.5 Denormalization: What is denormalization? Explain the pros and cons.

pg 173

SOLUTION

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database.

By contrast, in a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in the database.

For example, in a normalized database, we might have a `Courses` table and a `Teachers` table. Each entry in `Courses` would store the `teacherID` for a `Course` but not the `teacherName`. When we need to retrieve a list of all `Courses` with the `Teacher` name, we would do a join between these two tables.

In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place.

The drawback, however, is that if the tables are large, we may spend an unnecessarily long time doing joins on tables.

Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

Cons of Denormalization	Pros of Denormalization
Updates and inserts are more expensive.	Retrieving data is faster since we do fewer joins.
Denormalization can make update and insert code harder to write.	Queries to retrieve can be simpler (and therefore less likely to have bugs), since we need to look at fewer tables.
Data may be inconsistent. Which is the "correct" value for a piece of data?	
Data redundancy necessitates more storage.	

In a system that demands scalability, like that of any major tech companies, we almost always use elements of both normalized and denormalized databases.

14.6 Entity-Relationship Diagram: Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).

pg 173

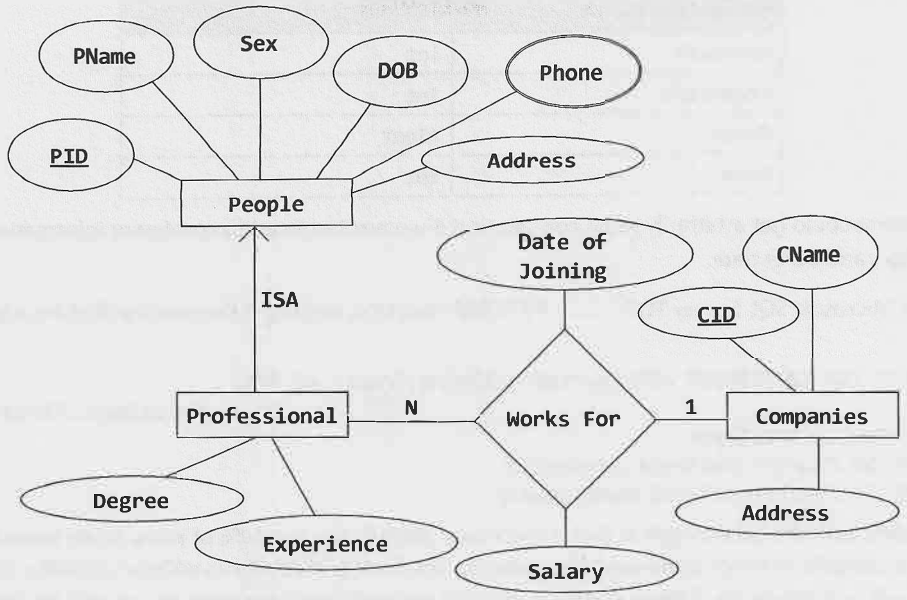
SOLUTION

People who work for `Companies` are `Professionals`. So, there is an `ISA` ("is a") relationship between `People` and `Professionals` (or we could say that a `Professional` is derived from `People`).

Each `Professional` has additional information such as degree and work experiences in addition to the properties derived from `People`.

A `Professional` works for one company at a time (probably—you might want to validate this assumption), but `Companies` can hire many `Professionals`. So, there is a many-to-one relationship between `Professionals` and `Companies`. This "Works For" relationship can store attributes such as an employee's start date and salary. These attributes are defined only when we relate a `Professional` with a `Company`.

A `Person` can have multiple phone numbers, which is why `Phone` is a multi-valued attribute.



14.7 Design Grade Database: Imagine a simple database storing information for students' grades. Design what this database might look like and provide a SQL query to return a list of the honor roll students (top 10%), sorted by their grade point average.

pg 173

SOLUTION

In a simplistic database, we'll have at least three objects: `Students`, `Courses`, and `CourseEnrollment`. `Students` will have at least a student name and ID and will likely have other personal information. `Courses` will contain the course name and ID and will likely contain the course description, professor, and other information. `CourseEnrollment` will pair `Students` and `Courses` and will also contain a field for `CourseGrade`.

Students	
StudentID	int
StudentName	varchar(100)
Address	varchar(500)

Courses	
CourseID	int
CourseName	varchar(100)
ProfessorID	int

CourseEnrollment	
CourseID	int
StudentID	int
Grade	float
Term	int

This database could get arbitrarily more complicated if we wanted to add in professor information, billing information, and other data.

Using the Microsoft SQL Server TOP ... PERCENT function, we might (incorrectly) first try a query like this:

```
SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA,
                                                      CourseEnrollment.StudentID
FROM CourseEnrollment
GROUP BY CourseEnrollment.StudentID
ORDER BY AVG(CourseEnrollment.Grade)
```

The problem with the above code is that it will return literally the top 10% of rows, when sorted by GPA. Imagine a scenario in which there are 100 students, and the top 15 students all have 4.0 GPAs. The above function will only return 10 of those students, which is not really what we want. In case of a tie, we want to include the students who tied for the top 10% -- even if this means that our honor roll includes more than 10% of the class.

To correct this issue, we can build something similar to this query, but instead first get the GPA cut off.

```
DECLARE @GPACutOff float;
SET @GPACutOff = (SELECT min(GPA) as 'GPAMin' FROM (
     SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA
     FROM CourseEnrollment
     GROUP BY CourseEnrollment.StudentID
     ORDER BY GPA desc) Grades);
```

Then, once we have @GPACutOff defined, selecting the students with at least this GPA is reasonably straightforward.

```
SELECT StudentName, GPA
FROM (SELECT AVG(CourseEnrollment.Grade) AS GPA, CourseEnrollment.StudentID
      FROM CourseEnrollment
      GROUP BY CourseEnrollment.StudentID
      HAVING AVG(CourseEnrollment.Grade) >= @GPACutOff) Honors
INNER JOIN Students ON Honors.StudentID = Student.StudentID
```

Be very careful about what implicit assumptions you make. If you look at the above database description, what potentially incorrect assumption do you see? One is that each course can only be taught by one professor. At some schools, courses may be taught by multiple professors.

However, you *will* need to make some assumptions, or you'd drive yourself crazy. Which assumptions you make is less important than just recognizing *that* you made assumptions. Incorrect assumptions, both in the real world and in an interview, can be dealt with *as long as they are acknowledged*.

Remember, additionally, that there's a trade-off between flexibility and complexity. Creating a system in which a course can have multiple professors does increase the database's flexibility, but it also increases its complexity. If we tried to make our database flexible to every possible situation, we'd wind up with something hopelessly complex.

Make your design reasonably flexible, and state any other assumptions or constraints. This goes for not just database design, but object-oriented design and programming in general.

15

Solutions to Threads and Locks

15.1 Thread vs. Process: What's the difference between a thread and a process?

pg 179

SOLUTION

Processes and threads are related to each other but are fundamentally different.

A process can be thought of as an instance of a program in execution. A process is an independent entity to which system resources (e.g., CPU time and memory) are allocated. Each process is executed in a separate address space, and one process cannot access the variables and data structures of another process. If a process wishes to access another process' resources, inter-process communications have to be used. These include pipes, files, sockets, and other forms.

A thread exists within a process and shares the process' resources (including its heap space). Multiple threads within the same process will share the same heap space. This is very different from processes, which cannot directly access the memory of another process. Each thread still has its own registers and its own stack, but other threads can read and write the heap memory.

A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads.

15.2 Context Switch: How would you measure the time spent in a context switch?

pg 179

SOLUTION

This is a tricky question, but let's start with a possible solution.

A context switch is the time spent switching between two processes (i.e., bringing a waiting process into execution and sending an executing process into waiting/terminated state). This happens in multitasking. The operating system must bring the state information of waiting processes into memory and save the state information of the currently running process.

In order to solve this problem, we would like to record the timestamps of the last and first instruction of the swapping processes. The context switch time is the difference in the timestamps between the two processes.

Let's take an easy example: Assume there are only two processes, P_1 and P_2.

P_1 is executing and P_2 is waiting for execution. At some point, the operating system must swap P_1 and P_2—let's assume it happens at the `N`th instruction of P_1. If $t_{x,k}$ indicates the timestamp in microseconds of the `k`th instruction of process `x`, then the context switch would take $t_{2,1} - t_{1,n}$ microseconds.

The tricky part is this: how do we know when this swapping occurs? We cannot, of course, record the timestamp of every instruction in the process.

Another issue is that swapping is governed by the scheduling algorithm of the operating system and there may be many kernel level threads which are also doing context switches. Other processes could be contending for the CPU or the kernel handling interrupts. The user does not have any control over these extraneous context switches. For instance, if at time $t_{1,n}$ the kernel decides to handle an interrupt, then the context switch time would be overstated.

In order to overcome these obstacles, we must first construct an environment such that after P_1 executes, the task scheduler immediately selects P_2 to run. This may be accomplished by constructing a data channel, such as a pipe, between P_1 and P_2 and having the two processes play a game of ping-pong with a data token.

That is, let's allow P_1 to be the initial sender and P_2 to be the receiver. Initially, P_2 is blocked (sleeping) as it awaits the data token. When P_1 executes, it delivers the token over the data channel to P_2 and immediately attempts to read a response token. However, since P_2 has not yet had a chance to run, no such token is available for P_1 and the process is blocked. This relinquishes the CPU.

A context switch results and the task scheduler must select another process to run. Since P_2 is now in a ready-to-run state, it is a desirable candidate to be selected by the task scheduler for execution. When P_2 runs, the roles of P_1 and P_2 are swapped. P_2 is now acting as the sender and P_1 as the blocked receiver. The game ends when P_2 returns the token to P_1.

To summarize, an iteration of the game is played with the following steps:

1. P_2 blocks awaiting data from P_1.
2. P_1 marks the start time.
3. P_1 sends token to P_2.
4. P_1 attempts to read a response token from P_2. This induces a context switch.
5. P_2 is scheduled and receives the token.
6. P_2 sends a response token to P_1.
7. P_2 attempts read a response token from P_1. This induces a context switch.
8. P_1 is scheduled and receives the token.
9. P_1 marks the end time.

The key is that the delivery of a data token induces a context switch. Let T_d and T_r be the time it takes to deliver and receive a data token, respectively, and let T_c be the amount of time spent in a context switch. At step 2, P_1 records the timestamp of the delivery of the token, and at step 9, it records the timestamp of the response. The amount of time elapsed, T, between these events may be expressed by:

$$T = 2 * (T_d + T_c + T_r)$$

This formula arises because of the following events: P_1 sends a token (3), the CPU context switches (4), P_2 receives it (5). P_2 then sends the response token (6), the CPU context switches (7), and finally P_1 receives it (8).

P_1 will be able to easily compute T, since this is just the time between events 3 and 8. So, to solve for T_c, we must first determine the value of $T_d + T_r$.

How can we do this? We can do this by measuring the length of time it takes P_1 to send and receive a token to itself. This will not induce a context switch since P_1 is running on the CPU at the time it sent the token and will not block to receive it.

The game is played a number of iterations to average out any variability in the elapsed time between steps 2 and 9 that may result from unexpected kernel interrupts and additional kernel threads contending for the CPU. We select the smallest observed context switch time as our final answer.

However, all we can ultimately say that this is an approximation which depends on the underlying system. For example, we make the assumption that P_2 is selected to run once a data token becomes available. However, this is dependent on the implementation of the task scheduler and we cannot make any guarantees.

That's okay; it's important in an interview to recognize when your solution might not be perfect.

15.3 Dining Philosophers: In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

pg 180

SOLUTION

First, let's implement a simple simulation of the dining philosophers problem in which we don't concern ourselves with deadlocks. We can implement this solution by having `Philosopher` extend `Thread`, and `Chopstick` call `lock.lock()` when it is picked up and `lock.unlock()` when it is put down.

```
class Chopstick {
  private Lock lock;

  public Chopstick() {
    lock = new ReentrantLock();
  }

  public void pickUp() {
    void lock.lock();
  }

  public void putDown() {
    lock.unlock();
  }
}

class Philosopher extends Thread {
  private int bites = 10;
  private Chopstick left, right;

  public Philosopher(Chopstick left, Chopstick right) {
    this.left = left;
    this.right = right;
  }
```

```java

  public void eat() {
    pickUp();
    chew();
    putDown();
  }

  public void pickUp() {
    left.pickUp();
    right.pickUp();
  }

  public void chew() { }

  public void putDown() {
    right.putDown();
    left.putDown();
  }

  public void run() {
    for (int i = 0; i < bites; i++) {
      eat();
    }
  }
}
```

Running the above code may lead to a deadlock if all the philosophers have a left chopstick and are waiting for the right one.

Solution #1: All or Nothing

To prevent deadlocks, we can implement a strategy where a philosopher will put down his left chopstick if he is unable to obtain the right one.

```java
public class Chopstick {
  /* same as before */

  public boolean pickUp() {
    return lock.tryLock();
  }
}

public class Philosopher extends Thread {
  /* same as before */

  public void eat() {
    if (pickUp()) {
      chew();
      putDown();
    }
  }

  public boolean pickUp() {
    /* attempt to pick up */
    if (!left.pickUp()) {
      return false;
    }
    if (!right.pickUp()) {
        left.putDown();
        return false;
      }
      return true;
    }
  }
```

In the above code, we need to be sure to release the left chopstick if we can't pick up the right one—and to not call `putDown()` on the chopsticks if we never had them in the first place.

One issue with this is that if all the philosophers were perfectly synchronized, they could simultaneously pick up their left chopstick, be unable to pick up the right one, and then put back down the left one—only to have the process repeated again.

Solution #2: Prioritized Chopsticks

Alternatively, we can label the chopsticks with a number from `0` to `N - 1`. Each philosopher attempts to pick up the lower numbered chopstick first. This essentially means that each philosopher goes for the left chopstick before right one (assuming that's the way you labeled it), except for the last philosopher who does this in reverse. This will break the cycle.

```
public class Philosopher extends Thread {
  private int bites = 10;
  private Chopstick lower, higher;
  private int index;
  public Philosopher(int i, Chopstick left, Chopstick right) {
    index = i;
    if (left.getNumber() < right.getNumber()) {
      this.lower = left;
      this.higher = right;
    } else {
      this.lower = right;
      this.higher = left;
    }
  }

  public void eat() {
    pickUp();
    chew();
    putDown();
  }

  public void pickUp() {
    lower.pickUp();
    higher.pickUp();
  }

  public void chew() { ... }

  public void putDown() {
    higher.putDown();
    lower.putDown();
  }

  public void run() {
    for (int i = 0; i < bites; i++) {
      eat();
      }
    }
  }

  public class Chopstick {
    private Lock lock;
    private int number;

    public Chopstick(int n) {
      lock = new ReentrantLock();
      this.number = n;
    }

    public void pickUp() {
      lock.lock();
    }

    public void putDown() {
      lock.unlock();
    }

    public int getNumber() {
      return number;
    }
  }
```

With this solution, a philosopher can never hold the larger chopstick without holding the smaller one. This prevents the ability to have a cycle, since a cycle means that a higher chopstick would "point" to a lower one.

15.4 Deadlock-Free Class: Design a class which provides a lock only if there are no possible deadlocks.

pg 180

SOLUTION

There are several common ways to prevent deadlocks. One of the popular ways is to require a process to declare upfront what locks it will need. We can then verify if a deadlock would be created by issuing these locks, and we can fail if so.

With these constraints in mind, let's investigate how we can detect deadlocks. Suppose this was the order of locks requested:

```
A = {1, 2, 3, 4}
B = {1, 3, 5}
C = {7, 5, 9, 2}
```

This may create a deadlock because we could have the following scenario:

```
A locks 2, waits on 3
B locks 3, waits on 5
C locks 5, waits on 2
```

We can think about this as a graph, where 2 is connected to 3, 3 is connected to 5, and 5 is connected to 2. A deadlock is represented by a cycle. An edge `(w, v)` exists in the graph if a process declares that it will request lock `v` immediately after lock `w`. For the earlier example, the following edges would exist in the graph: `(1, 2), (2, 3), (3, 4), (1, 3), (3, 5), (7, 5), (5, 9), (9, 2)`. The "owner" of the edge does not matter.

This class will need a `declare` method, which threads and processes will use to declare what order they will request resources in. This declare method will iterate through the declare order, adding each contiguous pair of elements (`v`, `w`) to the graph. Afterwards, it will check to see if any cycles have been created. If any cycles have been created, it will backtrack, removing these edges from the graph, and then exit.

We have one final component to discuss: how do we detect a cycle? We can detect a cycle by doing a depth-first search through each connected component (i.e., each connected part of the graph). Complex algorithms exist to find all the connected components of a graph, but our work in this problem does not require this degree of complexity.

We know that if a cycle was created, one of our new edges must be to blame. Thus, as long as our depth-first search touches all of these edges at some point, then we know that we have fully searched for a cycle.

The pseudocode for this special case cycle detection looks like this:

```
boolean checkForCycle(locks[] locks) {
  touchedNodes = hash table(lock -> boolean)
  initialize touchedNodes to false for each lock in locks
  for each (lock x in process.locks) {
    if (touchedNodes[x] == false) {
      if (hasCycle(x, touchedNodes)) {
        return true;
      }
    }
  }
  return false;
}

boolean hasCycle(node x, touchedNodes) {
  touchedNodes[r] = true;
  if (x.state == VISITING) {
    return true;
  } else if (x.state == FRESH) {
    ... (see full code below)
  }
}
```

In the above code, note that we may do several depth-first searches, but `touchedNodes` is only initialized once. We iterate until all the values in `touchedNodes` are false.

The code below provides further details. For simplicity, we assume that all locks and processes (owners) are ordered sequentially.

```
class LockFactory {
  private static LockFactory instance;

  private int numberOfLocks = 5; /* default */
  private LockNode[] locks;

  /* Maps from a process or owner to the order that the owner claimed it would
   * call the locks in */
  private HashMap<Integer, LinkedList<LockNode>> lockOrder;

  private LockFactory(int count) { ... }
  public static LockFactory getInstance() { return instance; }

  public static synchronized LockFactory initialize(int count) {
    if (instance == null) instance = new LockFactory(count);
    return instance;
  }

  public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes,
                          int[] resourcesInOrder) {
    / *check for a cycle */
    for (int resource : resourcesInOrder) {
      if (touchedNodes.get(resource) == false) {
        LockNode n = locks[resource];
        if (n.hasCycle(touchedNodes)) {
          return true;
        }
      }
    }
    return false;
  }

  / *To prevent deadlocks, force the processes to declare upfront what order they
   * will need the locks in. Verify that this order does not create a deadlock (a
   * cycle in a directed graph) */
  public boolean declare(int ownerId, int[] resourcesInOrder) {
    HashMap<Integer, Boolean> touchedNodes = new HashMap<Integer, Boolean>();

    / *add nodes to graph */
    int index = 1;
    touchedNodes.put(resourcesInOrder[0], false);
    for (index = 1; index < resourcesInOrder.length; index++) {
      LockNode prev = locks[resourcesInOrder[index - 1]];
      LockNode curr = locks[resourcesInOrder[index]];
      prev.joinTo(curr);
      touchedNodes.put(resourcesInOrder[index], false);
    }

    / *if we created a cycle, destroy this resource list and return false */
    if (hasCycle(touchedNodes, resourcesInOrder)) {
      for (int j = 1; j < resourcesInOrder.length; j++) {
        LockNode p = locks[resourcesInOrder[j - 1]];
        LockNode c = locks[resourcesInOrder[j]];
        p.remove(c);
      }
      return false;
    }

    / *No cycles detected. Save the order that was declared, so that we can
     * verify that the process is really calling the locks in the order it said
     * it would. */
    LinkedList<LockNode> list = new LinkedList<LockNode>();
    for (int i = 0; i < resourcesInOrder.length; i++) {
      LockNode resource = locks[resourcesInOrder[i]];
      list.add(resource);
    }
    lockOrder.put(ownerId, list);

    return true;
  }

  /* Get the lock, verifying first that the process is really calling the locks in
   * the order it said it would. */
  public Lock getLock(int ownerId, int resourceID) {
    LinkedList<LockNode> list = lockOrder.get(ownerId);
    if (list == null) return null;

    LockNode head = list.getFirst();
    if (head.getId() == resourceID) {
      list.removeFirst();
      return head.getLock();
    }
    return null;
  }
}

public class LockNode {
  public enum VisitState { FRESH, VISITING, VISITED };

  private ArrayList<LockNode> children;
  private int lockId;
  private Lock lock;
  private int maxLocks;

  public LockNode(int id, int max) { ... }

  /* Join "this" to "node", checking that it doesn't create a cycle */
  public void joinTo(LockNode node) { children.add(node); }
  public void remove(LockNode node) { children.remove(node); }

  /* Check for a cycle by doing a depth-first-search. */
  public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes) {
    VisitState[] visited = new VisitState[maxLocks];
    for (int i = 0; i < maxLocks; i++) {
      visited[i] = VisitState.FRESH;
    }
    return hasCycle(visited, touchedNodes);
  }

  private boolean hasCycle(VisitState[] visited,
                           HashMap<Integer, Boolean> touchedNodes) {
    if (touchedNodes.containsKey(lockId)) {
      touchedNodes.put(lockId, true);
    }

    if (visited[lockId] == VisitState.VISITING) {
      /* We looped back to this node while still visiting it, so we know there's
       * a cycle. */
      return true;
    } else if (visited[lockId] == VisitState.FRESH) {
      visited[lockId] = VisitState.VISITING;
      for (LockNode n : children) {
        if (n.hasCycle(visited, touchedNodes)) {
          return true;
        }
      }
      visited[lockId] = VisitState.VISITED;
    }
    return false;
  }

  public Lock getLock() {
    if (lock == null) lock = new ReentrantLock();
    return lock;
  }

  public int getId() { return lockId; }
}
```

As always, when you see code this complicated and lengthy, you wouldn't be expected to write all of it. More likely, you would be asked to sketch out pseudocode and possibly implement one of these methods.

15.5 Call In Order: Suppose we have the following code:

```
public class Foo {
  public Foo() { ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

The same instance of Foo will be passed to three different threads. ThreadA will call first, threadB will call second, and threadC will call third. Design a mechanism to ensure that first is called before second and second is called before third.

pg 180

SOLUTION

The general logic is to check if first() has completed before executing second(), and if second() has completed before calling third(). Because we need to be very careful about thread safety, simple boolean flags won't do the job.

What about using a lock to do something like the below code?

```
public class FooBad {
  public int pauseTime = 1000;
  public ReentrantLock lock1, lock2;

  public FooBad() {
    try {
      lock1 = new ReentrantLock();
      lock2 = new ReentrantLock();

      lock1.lock();
      lock2.lock();
    } catch (...) { ... }
  }

  public void first() {
    try {
      ...
      lock1.unlock(); // mark finished with first()
    } catch (...) { ... }
  }

  public void second() {
    try {
      lock1.lock(); // wait until finished with first()
      lock1.unlock();
      ...

      lock2.unlock(); // mark finished with second()
    } catch (...) { ... }
  }

  public void third() {
    try {
      lock2.lock(); // wait until finished with third()
      lock2.unlock();
      ...
    } catch (...) { ... }
  }
}
```

This code won't actually quite work due to the concept of *lock ownership*. One thread is actually performing the lock (in the FooBad constructor), but different threads attempt to unlock the locks. This is not allowed, and your code will raise an exception. A lock in Java is owned by the same thread which locked it.

Instead, we can replicate this behavior with semaphores. The logic is identical.

```
public class Foo {
  public Semaphore sem1, sem2;

  public Foo() {
    try {
      sem1 = new Semaphore(1);
      sem2 = new Semaphore(1);

      sem1.acquire();
      sem2.acquire();
    } catch (...) { ... }
  }

  public void first() {
    try {
      ...
      sem1.release();
    } catch (...) { ... }
  }

  public void second() {
    try {
      sem1.acquire();
      sem1.release();
      ...
      sem2.release();
    } catch (...) { ... }
  }

  public void third() {
    try {
      sem2.acquire();
```

```
            sem2.release();
            ...
        } catch (...) { ... }
    }
}
```

15.6 Synchronized Methods: You are given a class with `synchronized` method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time? Can they execute A and B at the same time?

pg 180

SOLUTION

By applying the word `synchronized` to a method, we ensure that two threads cannot execute synchronized methods *on the same object instance* at the same time.

So, the answer to the first part really depends. If the two threads have the same instance of the object, then no, they cannot simultaneously execute method A. However, if they have different instances of the object, then they can.

Conceptually, you can see this by considering locks. A synchronized method applies a "lock" on *all* synchronized methods in that instance of the object. This blocks other threads from executing synchronized methods within that instance.

In the second part, we're asked if `thread1` can execute synchronized method A while `thread2` is executing non-synchronized method B. Since B is not synchronized, there is nothing to block `thread1` from executing A while `thread2` is executing B. This is true regardless of whether `thread1` and `thread2` have the same instance of the object.

Ultimately, the key concept to remember is that only one synchronized method can be in execution per instance of that object. Other threads can execute non-synchronized methods on that instance, or they can execute any method on a different instance of the object.

15.7 FizzBuzz: In the classic problem FizzBuzz, you are told to print the numbers from 1 to n. However, when the number is divisible by 3, print "Fizz". When it is divisible by 5, print "Buzz". When it is divisible by 3 and 5, print "FizzBuzz". In this problem, you are asked to do this in a multithreaded way. Implement a multithreaded version of FizzBuzz with four threads. One thread checks for divisibility of 3 and prints "Fizz". Another thread is responsible for divisibility of 5 and prints "Buzz". A third thread is responsible for divisibility of 3 and 5 and prints "FizzBuzz". A fourth thread does the numbers.

pg 180

SOLUTION

Let's start off with implementing a single threaded version of FizzBuzz.

Single Threaded

Although this problem (in the single threaded version) shouldn't be hard, a lot of candidates overcomplicate it. They look for something "beautiful" that reuses the fact that the divisible by 3 and 5 case ("FizzBuzz") seems to resemble the individual cases ("Fizz" and "Buzz").

In actuality, the best way to do it, considering readability and efficiency, is just the straightforward way.

```
void fizzbuzz(int n) {
```

```java
  for (int i = 1; i <= n; i++) {
    if (i % 3 == 0 && i % 5 == 0) {
      System.out.println("FizzBuzz");
    } else if (i % 3 == 0) {
      System.out.println("Fizz");
    } else if (i % 5 == 0) {
      System.out.println("Buzz");
    } else {
      System.out.println(i);
    }
  }
}
```

The primary thing to be careful of here is the order of the statements. If you put the check for divisibility by 3 before the check for divisibility by 3 and 5, it won't print the right thing.

Multithreaded

To do this multithreaded, we want a structure that looks something like this:

FizzBuzz Thread	Fizz Thread
`if i div by 3 && 5`	`if i div by only 3`
`   print FizzBuzz`	`   print Fizz`
`   increment i`	`   increment i`
`repeat until i > n`	`repeat until i > n`

Buzz Thread	Number Thread
`if i div by only 5`	`if i not div by 3 or 5`
`   print Buzz`	`   print i`
`   increment i`	`   increment i`
`repeat until i > n`	`repeat until i > n`

The code for this will look something like:

```java
1  while (true) {
2    if (current > max) {
3      return;
4    }
5    if (/* divisibility test */) {
6      System.out.println(/* print something */);
7      current++;
8    }
9  }
```

We'll need to add some synchronization in the loop. Otherwise, the value of `current` could change between lines 2 - 4 and lines 5 - 8, and we can inadvertently exceed the intended bounds of the loop. Additionally, incrementing is not thread-safe.

To actually implement this concept, there are many possibilities. One possibility is to have four entirely separate thread classes that share a reference to the current variable (which can be wrapped in an object).

The loop for each thread is substantially similar. They just have different target values for the divisibility checks, and different print values.

	FizzBuzz	Fizz	Buzz	Number
current % 3 == 0	true	true	false	false
current % 5 == 0	true	false	true	false
to print	FizzBuzz	Fizz	Buzz	current

For the most part, this can be handled by taking in "target" parameters and the value to print. The output for the Number thread needs to be overwritten, though, as it's not a simple, fixed string.

We can implement a FizzBuzzThread class which handles most of this. A NumberThread class can extend FizzBuzzThread and override the print method.

```java
Thread[] threads = {new FizzBuzzThread(true, true, n, "FizzBuzz"),
                    new FizzBuzzThread(true, false, n, "Fizz"),
                    new FizzBuzzThread(false, true, n, "Buzz"),
                    new NumberThread(false, false, n)};
for (Thread thread : threads) {
  thread.start();
}

public class FizzBuzzThread extends Thread {
  private static Object lock = new Object();
  protected static int current = 1;
  private int max;
  private boolean div3, div5;
  private String toPrint;

  public FizzBuzzThread(boolean div3, boolean div5, int max, String toPrint) {
    this.div3 = div3;
    this.div5 = div5;
    this.max = max;
    this.toPrint = toPrint;
  }

  public void print() {
    System.out.println(toPrint);
  }

  public void run() {
    while (true) {
      synchronized (lock) {
        if (current > max) {
          return;
        }

        if ((current % 3 == 0) == div3 &&
            (current % 5 == 0) == div5) {
          print();
          current++;
        }
      }
    }
  }
}

public class NumberThread extends FizzBuzzThread {
  public NumberThread(boolean div3, boolean div5, int max) {
    super(div3, div5, max, null);
  }

  public void print() {
    System.out.println(current);
  }
}
```

Observe that we need to put the comparison of `current` and `max` before the if statement, to ensure the value will only get printed when `current` is less than or equal to `max`.

Alternatively, if we're working in a language which supports this (Java 8 and many other languages do), we can pass in a `validate` method and a `print` method as parameters.

```
int n = 100;
Thread[] threads = {
  new FBThread(i -> i % 3 == 0 && i % 5 == 0, i -> "FizzBuzz", n),
  new FBThread(i -> i % 3 == 0 && i % 5 != 0, i -> "Fizz", n),
  new FBThread(i -> i % 3 != 0 && i % 5 == 0, i -> "Buzz", n),
  new FBThread(i -> i % 3 != 0 && i % 5 != 0, i -> Integer.toString(i), n)};
for (Thread thread : threads) {
  thread.start();
}

public class FBThread extends Thread {
  private static Object lock = new Object();
  protected static int current = 1;
  private int max;
  private Predicate<Integer> validate;
  private Function<Integer, String> printer;
  int x = 1;

  public FBThread(Predicate<Integer> validate,
                  Function<Integer, String> printer, int max) {
    this.validate = validate;
    this.printer = printer;
    this.max = max;
  }

  public void run() {
    while (true) {
      synchronized (lock) {
        if (current > max) {
          return;
        }
        if (validate.test(current)) {
          System.out.println(printer.apply(current));
          current++;
        }
      }
    }
  }
}
```

There are of course many other ways of implementing this as well.