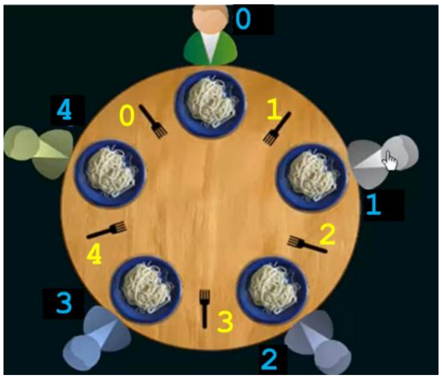
Lab 5: Deadlocks

Total points: 100

Assignment 1 (25 points) – Dining Philosophers 1



Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.

The life of a philosopher consists of alternating periods of eating and thinking. When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.

The key question is: Can we write a program for each philosopher that does what it is supposed to do and never gets stuck?

Assignment 2 (25 points) – Exercise 8.24 - Dining Philosophers 2

Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

Assignment 3 (20 points) – Programing Problem 8.33

Please show disadvantages the following solution.

Account.java

```java
public class Account {
    private int id;
    private double balance;

    public Account(int id, double balance) {
        this.id = id;
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

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

Bank.java

```java
public class Bank {
    private ArrayList<Account> accounts = new ArrayList<>();

    public Bank(int accountNum, int balance) {
        for(int i = 0; i < accountNum; i++) {
            Account acc = new Account(i, balance);
            this.accounts.add(acc);
        }
    }

    private Account find(int id) {
        for(int i = 0; i < this.accounts.size(); i++)
            if(this.accounts.get(i).getId() == id)
                return this.accounts.get(i);
        return null;
    }

    public boolean transaction(int fromId, int toId, double amount) {
        Account from = this.find(fromId);
        if(from == null)
            return false;
        Account to = this.find(toId);
        if(to == null)
            return false;
        return this.transaction(from, to, amount);
    }
```

```java

    private synchronized boolean transaction(Account from, Account to, double amout) {
        if(from.getBalance() < amout)
            return false;
        from.setBalance(from.getBalance() - amout);
        to.setBalance(to.getBalance() + amout);
        return true;
    }
}
```

In Figure 8.7, we illustrate a `transaction()` function that dynamically acquires locks. In the text, we describe how this function presents difficulties for acquiring locks in a way that avoids deadlock. Using the Java implementation of `transaction()` that is provided in the source-code download for this text, modify it using the `System.identityHashCode()` method so that the locks are acquired in order. You should develop an implementation that each Account instance has a ReentrantLock and these lock objects are ordered using values returned by the `System.identityHashCode()` method .

```
void transaction(Account from, Account to, double amount)
{
  mutex lock1, lock2;
  lock1 = get_lock(from);
  lock2 = get_lock(to);

  acquire(lock1);
    acquire(lock2);

      withdraw(from, amount);
      deposit(to, amount);

    release(lock2);
  release(lock1);
}
```

Figure 8.7 Deadlock example with lock ordering.

Assignment 4 (30 points) – Programing Project – Banker’s Algorithm

For this project, you will write a program that implements the banker’s algorithm used in deadlock avoidance, discussed in Section 8.6.3. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied.

Banker.java

```java
import java.util.ArrayList;

public class Banker {
    private int resourceTypeNum;    //hold the number of resource types
    private int customerNum;        //hold the number of customers

    private int[] available;    //number of resources for each resource type
    private int[][] maximum;    //the maximum number of requests for each customer
                                //Number of rows: the number of customer,
                                //number of columns: the number of resource types
    private int[][] allocation; //the number of resources of each type currently allocated to each customer
                                //Number of rows: the number of customer,
                                //number of columns: the number of resource types

    public Banker(int[] avail, int[][] max, int[][] alloc) throws Exception {

    //The system is a in safe state ?
    public ArrayList<Integer> isSafeState() {

    // customer id requests resources; returns true if the banker can allocate resources for this
    //customer and leaves a safe state
    public ArrayList<Integer> request(int custId, int[] request) {

    public int[] getAvailable() {

    public int[][] getMaximum() {

    public int[][] getAllocation() {
```

Suppose that a snapshot at time T_0 is shown below:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- Run the program and show whether the system is in a safe state or not.
- Suppose that P1 requests (1, 0, 2). Can the request be granted?