SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Established in collaboration with MIT

Computer Systems Engineering

Week 3: Lab 3 (40 marks)

Objective: Write a Java/C program that implements the banker's algorithm.

Relevant Material:

> Section7.5.3 Banker's algorithm, P331-334,
> Operating System Concepts with Java, Eighth Edition

There are several 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 is denied if it leaves the system in an unsafe state.

The bank will employ the banker's algorithm, whereby it will consider requests from n customers for m resources. The bank will keep track of the resources using the following variables:

Java:

```java
private int numberOfCustomers;  // the number of customers
private int numberOfResources;  // the number of resources

private int[] available;    // the available amount of each resource
private int[][] maximum;    // the maximum demand of each customer
private int[][] allocation; // the amount currently allocated
private int[][] need;       // the remaining needs of each customer
```

C:

```c
int numberOfCustomers; // the number of customers
int numberOfResources; // the number of resources

int *available;       // the available amount of each resource
int **maximum;        // the maximum demand of each customer
int **allocation;     // the amount currently allocated
int **need;           // the remaining needs of each customer
```

This lab is organized into three parts:

1. Implementing a basic bank system (tested with q1.txt) **(20 marks)**.
2. Implementing a safety check algorithm (tested with q2.txt) **(20 marks)**.
3. Analysis of the complexity of the Banker's algorithm **(10 marks)**.

Q1: Implement a basic bank system (20 marks)

The first part of the project is to implement the following functions:

Java:

```java
public Banker (int[] resources, int numberOfCustomers);
public void setMaximumDemand(int customerIndex, int[] maximumDemand);
public void printState();
public synchronized boolean requestResources(int customerIndex, int[] request);
private synchronized boolean checkSafe(int customerIndex, int[] request);
public synchronized void releaseResources(int customerIndex, int[] release);
```

C:

```c
void initBank(int *resources, int m, int n);
void setMaximumDemand(int customerIndex, int *maximumDemand);
void printState();
int requestResources(int customerIndex, int *request);
void releaseResources(int customerIndex, int *release);
```

For Q1, you would not need implement the requestResources to check for a safety state yet.
For now, you can just let it return true, or 1.

Also, you can assume that the customerIndex will be in a valid range, and the array passed to releaseResources is valid.

To help you focus on implementing the algorithms, a function, runFile, has been provided to parse and run the test cases. For Q1, the schema of the test case is:

n,5	(n) Number of customers: 5
m,3	(m) Number of resources: 3
a,10 5 7	(a) Available resources: 10 5 7
c,0,7 5 3	(c) Maximum resources needed by the 0th customer: 7 5 3
c,1,3 2 2	
c,2,9 0 2	
c,3,2 2 2	
c,4,4 3 3	(c) Maximum resources needed by the 4th customer: 7 5 3
r,0,0 1 0	(r) Request for resources by the 0th customer: 0 1 0
r,1,2 0 0	
r,2,3 0 2	
r,3,2 1 1	
r,4,0 0 2	(r) Request for resources by the 4th customer: 0 0 2
f,1,2 0 0	(f) Release of resources by the 1st customer: 1 0 0
p	(p) Print the current state of the bank

To run the test case, you can pass it as the first argument into the program.
(for Java: `java Banker ../q1.txt`) (for C: `./banker q1.txt`)

This test case sets up the bank, initializes the maximum needs of the customers, and attempts to make a sequence of requests and releases. If we were to inspect the state of the bank, we will see that it goes through the following states:

After initialization:

Customers	Allocation				Maximum				Need				Available		
0	0	0	0		0	0	0		0	0	0		10	5	7
1	0	0	0		0	0	0		0	0	0				
2	0	0	0		0	0	0		0	0	0				
3	0	0	0		0	0	0		0	0	0				
4	0	0	0		0	0	0		0	0	0				

After initializing the maximum needs:

Customers	Allocation				Maximum				Need				Available		
0	0	0	0		7	5	3		7	5	3		10	5	7
1	0	0	0		3	2	2		3	2	2				
2	0	0	0		9	0	2		9	0	2				
3	0	0	0		2	2	2		2	2	2				
4	0	0	0		4	3	3		4	3	3				

Final state after the sequence of requests and releases:

Customers	Allocation				Maximum				Need				Available		
0	0	1	0		7	5	3		7	4	3		4	3	2
1	1	0	0		3	2	2		1	2	2				
2	3	0	2		9	0	2		6	0	0				
3	2	1	1		2	2	2		0	1	1				
4	0	0	2		4	3	3		4	3	1				

Hence, the expected output should be:

```
Customer 0 requesting
0 1 0
Customer 1 requesting
2 0 0
Customer 2 requesting
3 0 2
Customer 3 requesting
2 1 1
Customer 4 requesting
0 0 2
Customer 1 releasing
1 0 0
```

```
Current state:
Available:
4 3 2

Maximum:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Allocation:
0 1 0
1 0 0
3 0 2
2 1 1
0 0 2

Need:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1
```

Q2: Implementing a Safety Check algorithm (20 marks)

Now, you will need to implement the checkSafe function.

Java:

```
private synchronized boolean checkSafe(int customerIndex, int[] request);
```

C:

```
int checkSafe(int customerIndex, int *request);
```

This function implements the safety algorithm for the Banker's algorithm.
The pseudocode is as follows:

```
boolean checkSafe(customerNumber, request) {
    temp_avail = available - request;
    temp_need(customerNumber) = need - request;
    temp_allocation(customerNumber) = allocation + request;
    work = temp_avail;
    finish(all) = false;
    possible = true;
    while(possible) {
        possible = false;
        for(customer Ci = 1:n) {
            if (finish(Ci) == false && temp_need(Ci) <= work) {
                possible = true;
                work += temp_allocation(Ci);
                finish(Ci) = true;
            }
        }
    }
    return (finish(all) == true);
}
```

For Q2, the schema of the test case is as follows:

```
n,5
m,3
a,10 5 7
c,0,7 5 3
c,1,3 2 2
c,2,9 0 2
c,3,2 2 2
c,4,4 3 3
r,0,0 1 0
r,1,2 0 0
r,2,3 0 2
r,3,2 1 1
r,4,0 0 2
r,1,1 0 2
p
r,0,0 2 0
p
```

We attempt to make an loan that can leave the bank in unsafe state:

(r) Request for resources by the 0th customer: 0 2 0

(p) This should print the same result as the previous print.

For the very last request, the state of the bank is:

Customers	Allocation				Maximum				Need				Available		
0	0	1	0		7	5	3		7	4	3		2	3	0
1	3	0	2		3	2	2		0	2	0				
2	3	0	2		9	0	2		6	0	0				
3	2	1	1		2	2	2		0	1	1				
4	0	0	2		4	3	3		4	3	1				

Although there is enough available resources to loan out "0 2 0" to customer 0, it will leave the bank in an unsafe state. Hence, this loan should not be approved and the bank state should not change.

The expected output of Q2 should be:

```
Customer 0 requesting
0 1 0
Customer 1 requesting
2 0 0
Customer 2 requesting
3 0 2
Customer 3 requesting
2 1 1
Customer 4 requesting
0 0 2
Customer 1 requesting
1 0 2
```

```
Current state:
Available:
2 3 0

Maximum:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Allocation:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

Need:
7 4 3
0 2 0
6 0 0
0 1 1
4 3 1
```

```
Customer 0 requesting
0 2 0
```

```
Current state:
Available:
2 3 0

Maximum:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Allocation:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

Need:
7 4 3
0 2 0
6 0 0
0 1 1
4 3 1
```

Q3: Discuss about the complexity of Banker’s algorithm (10 marks)

Lab3 submission：

Put your result screenshots and analysis into a pdf file. Your submission should contain: one pdf report; source code (Banker.java / Banker.c).

Please zip them up and submit before 12:00 PM, 22nd Feb 2018.