SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Established in collaboration with MIT

Computer System Engineering

Submission and due date:

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

Please submit before 11:59 PM, 1st March, 2017.

Relevant material in textbook:

Section7.5.3 Banker's algorithm, P331-334

Operating System Concepts with Java, Eighth Edition

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

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 outlined in textbook (see P331-334, Section 7.5.3, Operating System Concepts with Java, Eighth Edition), whereby it will consider requests from *n* customers for *m* resources. The bank will keep track of the resources using the following data structures(for Java, it is quite similar for C version, please refer to C starting code for details):

```
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
```

The functions of class `BankImpl` are shown below (which need to be implemented for Java, C version is quite similar):

```
/**
 * Constructor
 * @param resources - the initial available amount of each resource
 * @param numberOfCustomers - the number of customers
 */
public BankImpl (int[]resources, int numberOfCustomers);


/**
 * Add a customer
 * @param customerNumber - the number of the customer
 * @param maximumDemand - the maximum demand for this customer
```

```java
 */
public void addCustomer(int customerNumber, int[] maximumDemand);

/**
 * Output the value of available, maximum,
 * allocation, and need
 */
public void getState();

/**
 * Request resources
 * If the request is not granted, this method should print the error message
 * @param customerNumber - the customer requesting resources
 * @param request - the resources being requested
 * @return grant state - whether granting the request leaves the system in
safe or unsafe state
 */
public boolean requestResources(int customerNumber, int[] request);

/**
 * Release resources
 * @param customerNumber - the customer releasing resources
 * @param release - the resources being released
 */
public void releaseResources(int customerNumber, int[] release);

/**
 * Get number of customers
 * @return numberCustomer - number of customers
 */
public int getNumberOfCustomers();
```

The implementation (for Java) of those functions includes adding a constructor that is passed the number of resources initially available, and the number of customers in the bank. For example, suppose we have three resource types with 10, 5, and 7 resources initially available, and 5 customers in the bank. In this case, we can create an implementation of the interface using the following technique:

```java
int[] resource = {10, 5, 7};
int numberOfCustomers = 5;
BankImpl theBank1 = new BankImpl(resource, numberOfCustomers);
```

For C implementation, we use function `initBank()` to initialize status:

```c
void initBank(int resources[], int resourcesNumber, int
customerNumber) {
/**
 * TODO: set numberOfCustomers and numberOfResources
 * TODO: init available/maximum/allocation/need
*/
}
```

The bank will grant a request if the request satisfies the safety algorithm outlined in textbook (see P332, Section 7.5.3.1, Operating System Concepts with Java, Eighth Edition, a pseudo code is also given at the end of this handout). If granting the request does not leave the system in a safe state, the request is denied.

Lab3: Java version

Q1. Implement the BankImpl basic functions

Use the starting code “BankImpl – starting code.java” to implement.

For Q1, your code should have the following constructor and functions:

```
public BankImpl(int[] resources, int numberOfCustomers);

public void addCustomer(int customerNumber, int[] maximumDemand);

public void getState();

public boolean requestResources(int customerNumber, int[] request);

public void releaseResources(int customerNumber, int[] release);

public int getNumberOfCustomers();
```

Note that for Q1, your `requestResources` function does not need to check safety state using safety algorithm. You can leave this part to Q2.

After you finish the implementation, use the test code “TestBankQ1.java” to test your code.

Q2. Check the safety state of your BankImpl class

Continue to use the BankImpl class you implemented in Q1.

For Q2, you need to add a function to your BankImpl class:

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

This function implements the safety algorithm (see P332, Section 7.5.3.1, Operating System Concepts with Java, Eighth Edition, a pseudo code is also given at the end of this handout). When a customer has a new request using the `requestResources` function, you use the `checkSafe` function to check if the request satisfies the safety algorithm. If it satisfies, the request is granted; if not, the request is denied.

After you finish the implementation, use the test code “TestBankQ2.java” to test your code.

Q3. Discussion

Discuss about the complexity of Banker’s algorithm.

The starting codes for Q1 and Q2 can be found in attached folder:

BankImpl – starting code.java

TestBankQ1.java

TestBankQ2.java

Lab3: C version

Q1. Implement the Banker's algorithm without safeCheck in C

Use the starting code "Banker_starting_code.c" to implement.

For Q1, your code should have the following constructor and functions:

```
void InitBank (int resources[], int resourcesNumber, int customerNumber);

int addCustomer(int customerId, int maximumDemand[]);

void showState();

int requestResources(int customerId, int request[]);

int releaseResources(int customerId, int release[]);
```

See the requirements in Banker_starting_code.c.

Note that for Q1, your `requestResources` function does not need to check safety state using safety algorithm. You can leave this part to Q2.

After you finish the implementation, use the test code for question1 in main function of starting code.

Tips : compile the code using `gcc Banker.c -o Banker`

Q2. Check the safety state of your Banker's algorithm

Continue to use the system you implemented in Q1.

For Q2, you need to add a function to your BankImpl class:

```
int checkSafe(int customerId, int request[]);
```

This function implements the safety algorithm (see P332, Section 7.5.3.1, Operating System Concepts with Java, Eighth Edition, a pseudo code is also given at the end of this handout). When a customer has a new request using the `requestResources` function, you use the `checkSafe` function to check if the request satisfies the safety algorithm. If it satisfies, the request is granted; if not, the request is denied.

After finish coding, use the test code for question2 in main function of starting code.

Q3. Discussion

Discuss about the complexity of Banker's algorithm.

Appendix I: reference test results

Test case in question 1: (take Java as example, same results for C)

Consider a bank with five customers *C*0 through *C*4 and three resource types with 10, 5, and 7 resources initially available. First, we initialize the bank using the `BankImpl` constructor:

```
// set the bank resource
int[] resource1 = {10, 5, 7};
// create a bank based on the resource
Bank theBank1 = new BankImpl(resource1, 5);
```

For C version, we use the similar function `InitBank`.

Bank state:

Customers	Allocation				Max				Need				Available		
*C*0													10	5	7
*C*1															
*C*2															
*C*3															
*C*4															

Then, we add the following maximum demands for the five customers using `addCustomer` function (the same with C version):

```
// get customer maximum demand
int[][] maximum1 = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3},
};
// get the number of customers for the bank
int numberCustomer1 = theBank1.getNumberOfCustomers();
// add each customer
for (int i = 0; i < numberCustomer1; i++){
        theBank1.addCustomer(i, maximum1[i]);
}
```

Bank state:

Customers	Allocation				Max				Need				Available		
*C*0	0	0	0		7	5	3		7	5	3		10	5	7
*C*1	0	0	0		3	2	2		3	2	2				
*C*2	0	0	0		9	0	2		9	0	2				

$C3$	0	0	0		2	2	2		2	2	2				
$C4$	0	0	0		4	3	3		4	3	3				

After add the customers, we add the request from all customers (the same with C version):

```
// request resource
int[] request1_0 = {0, 1, 0};
int[] request1_1 = {2, 0, 0};
int[] request1_2 = {3, 0, 2};
int[] request1_3 = {2, 1, 1};
int[] request1_4 = {0, 0, 2};
theBank1.requestResources(0, request1_0);
theBank1.requestResources(1, request1_1);
theBank1.requestResources(2, request1_2);
theBank1.requestResources(3, request1_3);
theBank1.requestResources(4, request1_4);
```

Bank state:

Customers	Allocation				Max				Need				Available		
$C0$	0	1	0		7	5	3		7	4	3		3	3	2
$C1$	2	0	0		3	2	2		1	2	2				
$C2$	3	0	2		9	0	2		6	0	0				
$C3$	2	1	1		2	2	2		0	1	1				
$C4$	0	0	2		4	3	3		4	3	1				

For Q1, your bank's output state should fits the above table.

Test case in Question 2: (take Java as example, same results for C)

Continue with the above test case. Customer $C1$ has a new request(the same with C version):

```
// request new resource
int[] request1_1_new1 = {1, 0, 2};
theBank1.requestResources(0, request1_1_new1);
```

The new request of customer $C1$ leaves the Bank in a safe state, so it is granted. The Bank state changes.

Bank state:

Customers	Allocation				Max				Need				Available		
*C*0	0	1	0		7	5	3		7	4	3		2	3	0
*C*1	3	0	2		3	2	2		0	2	0				
*C*2	3	0	2		9	0	2		6	0	0				
*C*3	2	1	1		2	2	2		0	1	1				
*C*4	0	0	2		4	3	3		4	3	1				

Now customer *C*0 has a new request(the same with C version):

```
// request new resource
int[] request1_0_new1 = {0, 2, 0};
theBank1.requestResources(0, request1_0_new1);
```

The new request of customer *C*0 leaves the Bank in an unsafe state, so it is denied, and Bank state remains the same.

Bank state:

Customers	Allocation				Max				Need				Available		
*C*0	0	1	0		7	5	3		7	4	3		2	3	0
*C*1	3	0	2		3	2	2		0	2	0				
*C*2	3	0	2		9	0	2		6	0	0				
*C*3	2	1	1		2	2	2		0	1	1				
*C*4	0	0	2		4	3	3		4	3	1				

Appendix II: pseudocode of safety check algorithm

```
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);
}
```