OPERATING SYSTEM PROJECT REPORT

PROJECT REPORT ON

IMPLEMENTATION OF BANKER'S ALGORITHM IN JAVA

USING MULTITHREADING

BY:

ABHISHEK MADAN (011408969)

Index	Title
1	Executive Summary
2	What is Banker's algorithm
3	Implementation details of Banker's algorithm
4	Safety algorithm and Request-response algorithm
5	An illustrative example
6	Program of Banker's algorithm using multithreading in Java
7	Output and analysis of the output

Executive summary

Objective
Primary objective of this project is to be able to learn and implement Banker's algorithm in Java successfully using multithreading.

Sources
Operating System Concepts (9th Ed) - Gagne, Silberschatz, and Galvin

Findings
We were able to learn and implement Banker's algorithm using OOPS and multithreading features.

Conclusion
Banker's algorithm is an excellent mechanism to avoid situation of deadlock in the system, as it always runs the system In the safe state.

What is Banker's Algorithm?

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes a safe-state check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. The algorithm was developed in the design process for the operating system and originally described (in Dutch) in EWD108. The name is by analogy with the way that bankers account for liquidity constraints.

The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm avoids deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

When a new process enters a system, it must declare the maximum number of instances of each resource type that may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

For the Banker's algorithm to work, it needs to know three things:

1. How much of each resource each process could possibly request
2. How much of each resource each process is currently holding
3. How much of each resource the system currently has available

Resources may be allocated to a process only if it satisfies the following conditions:

1. request ≤ max, else set error condition as process has crossed maximum claim made by it.
2. request ≤ available, else process waits until resources are available.

Some of the resources that are tracked in real systems are memory, semaphores and interface access. It derives its name from the fact that this algorithm could be used in a banking system to ensure that the bank does not run out of resources, because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. By using this algorithm, the bank ensures that when customers request money the bank never leaves a safe state. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated; otherwise the customer must wait until some other customer deposits enough.

Implementation of Banker's Algorithm

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:

1. **Available**: A vector of length m indicates the number of available resources of each type.
   If Available[j]=k, there are k instances of resource type Rj available.

2. **Max**: An n×m matrix defines the maximum demand of each process.
   If Max[i,j]=k, then process Pi may request at most k instances of resource type Rj.

3. **Allocation**: An n×m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j]=k, then process Pi is currently allocated k instances of resource type Rj.

4. **Need**: An n×m matrix indicates the remaining resource need of each process.
   If Need[i,j]=k, then process Pi may need k more instances of resource type Rj to complete its task.

$$Need[i, j]=Max[i,j]-Allocation[i,j]$$

These data structures vary over time in both size and value.
To simplify the presentation of the banker's algorithm, let us establish some notation.

Let X and Y be vectors of length n, $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all i=1, 2, …, n
For example, if X= (1,7,3,2) and Y= (0,3,2,1), then $Y \leq X$. $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocationi and Needi, respectively. The vector Allocationi specifies the resources currently allocated to process Pi; the vector Needi specifies the additional resources that process Pi may still request to complete its task.

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be:

Let Work and Finish be vectors of length m and n, respectively.
Initialize Work := Available and Finish [i]:=false for i=1,2,…,n

1. Find an i such that both
   a. Finish[i]=false
   b. Needi ≤ Work

2. Work := Work + Allocationi
   Finish[i] := true
   go to step 2.

3. If Finish[i]=true for all i, then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to decide whether a state is safe.

Resource-Request Algorithm

Let Requesti be the request vector for process Pi. If Requesti[j]=k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If Requesti ≤ Needi, go to step2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If Requesti ≤ Available, go to step3. Otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

   Available := Available- Requesti;
   Allocationi := Allocationi + Requesti;
   Needi := Needi - Requesti;

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. If the new state is unsafe, then Pi must wait for Requesti and the old resources-allocation state is restored.

An illustrative Example

Consider the system with five processes P0 through P4 and three resource types A, B, C.

Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances.

Suppose that, at time T0, the following snapshot of the system has been taken:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

The content of the matrix Need is defined to be Max- Allocation and is

	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

We claim that the system is currently in a safe state. The sequence <P1, P3, P4, P2, P0> satisfies the safety criteria. Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1= (1,0,2). To decide whether this request can be immediately granted, we first check that Request1 ≤Available which is true. Then this request has been fulfilled, and the following new state is arrived:

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

We must determine whether this new system state is safe. To do so, safety algorithm should be executed and the sequence is found out to be <P1, P3, P4, P0, P2> satisfies our safety requirement. Hence, the request of process P1 can be granted immediately.

Banker's algorithm program in Java using multithreading

Bank.java

```
import java.util.Arrays;
import java.util.Random;

public class Bank {
	public static final int RESOURCE = 3;
	public static final int CUSTOMER = 5;
	private static Bank instance = null;
	private static Customer c[];

	private Bank() {}

	public static Bank getInstance() {
		if (instance == null) {
			instance = new Bank();
			initStudents();
		}
		return instance;
	}

	private static void initStudents() {
		c = new Customer[Bank.CUSTOMER];
		initializeCustomer();
	}

	public static void initializeCustomer() {
		c[0]=new Customer(0,7,5,3);
		c[1]=new Customer(1,3,2,2);
		c[2]=new Customer(2,9,0,2);
		c[3]=new Customer(3,2,2,2);
		c[4]=new Customer(4,4,3,3);
	}

	public Customer getCustomer(int i) {
		return c[i];
	}

	public Customer[] getAllCustomer() {
		return c;
	}
}
```

Customer.java

```
public class Customer extends Thread{

        public int clientNo;
        public boolean isRunning;
         private int allocation[]=new int[Bank.RESOURCE];
        private int max[]=new int[Bank.RESOURCE];
        private int need[]=new int[Bank.RESOURCE];

        private static SharedResource sharedResource= new SharedResource();

         Customer() {}

         Customer(int num, int resourceA, int resourceB, int resourceC) {
                clientNo = num;
                max[0] = resourceA;
                max[1] = resourceB;
                max[2] = resourceC;
                for (int index=0 ; index< Bank.RESOURCE; index++  ){
                        allocation[index] =0;
                        need[index] = max[index]-allocation[index];
                }
            isRunning = false;
  }


  @Override
  public void run() {
        sharedResource.allocateResources(clientNo);
  }

  public int[] getClientNeed(){
     return need;
  }

  public int[] getClientAllocation() {
     return allocation;
  }

  public int[] getClientMax() {
        return max;
  }

  public void addToClientAllocation(int n[]) {
        for(int i=0 ; i<n.length; i++) {
                allocation[i] += n[i];
        }
  }
```

```
    public void adjustNeed() {
        for (int index=0 ; index< Bank.RESOURCE; index++  ){
            need[index] = max[index]-allocation[index];
        }
    }

    public void deallocate() {
        for (int index=0 ; index< Bank.RESOURCE; index++  ){
            need[index] = max[index];
            allocation[index] = 0;
        }
    }

        public void printClientStatus() {
            System.out.print("Client "+clientNo+"  Allocation :  ");
            for (int i = 0; i < Bank.RESOURCE; i++)
                System.out.print(allocation[i] + "  ");
            System.out.print("Max :  ");
            for (int i = 0; i < Bank.RESOURCE; i++)
                System.out.print(max[i] + "  ");
            System.out.print("Need :  ");
            for (int i = 0; i < Bank.RESOURCE; i++)
                System.out.print(need[i] + "  ");
            System.out.print("\n");
        }

        public void setRunning(boolean status){
            isRunning = status;
        }
}
```

SharedResource.java

```
import java.util.Arrays;
import java.util.Random;

public class SharedResource {
 public static int available[]={10,5,7};
 int request[] ={0,0,0};
 static Random select = new Random();

public synchronized void allocateResources(int client) {

	generateClientRequest(client);
	showAvailableResource();
	if (!isOneLessThanOther(request, available)) {
		System.out.println("Request cannot be granted!");
	}else{
		Customer c[] = Bank.getInstance().getAllCustomer();
		String seq = "";
		if ((seq=safeSequenceExist(c, client, request))!="") {
			Bank.getInstance().getCustomer(client).addToClientAllocation(request);
			Bank.getInstance().getCustomer(client).adjustNeed();
			for (int i = 0; i < available.length; i++) {
				available[i] = available[i] - request[i];
			}
				System.out.println("Safe sequence exist! "+seq);
			} else {
				System.out.println("Safe sequence does not exist!");
			}
		}
	showRemainingResource();
	Bank.getInstance().getCustomer(client).setRunning(true);
	clientState(Bank.getInstance().getAllCustomer());
}


public synchronized void deAllocateResources(int client) {
	int[] clientAllocation = Bank.getInstance().getCustomer(client).getClientAllocation();
	for(int i=0 ; i<available.length; i++) {
		available[i]+= clientAllocation[i];
	}
	Bank.getInstance().getCustomer(client).deallocate();
	Bank.getInstance().getCustomer(client).setRunning(false);
	showCompleteStatus(client);
}
```

```
public String safeSequenceExist(Customer c[],int requestingClient,int request[]){
	String sequence="";
	boolean flag[] = new boolean[Bank.CUSTOMER];
	Customer client[] = new Customer[Bank.CUSTOMER];
	for(int i=0 ; i< Bank.CUSTOMER; i++) {
		client[i] = new Customer();
	}
	copyCustomer(client, c);
	int work[]= new int[available.length];
	Arrays.fill(flag, false);

	client[requestingClient].addToClientAllocation(request);
	client[requestingClient].adjustNeed();

	for(int i=0 ; i<available.length; i++) {
		work[i]=available[i]-request[i];
	}

	int streak=0, index=0;
	while(streak<client.length){
		if(!flag[index]&&isOneLessThanOther(client[index].getClientNeed(),work)){
			for(int i=0 ; i< work.length; i++){
				work[i]+=client[index].getClientAllocation()[i];
			}
			flag[index]=true;
			sequence +=(index+1)+ " -> ";
			index = (index+1)%client.length;
			streak=0;
		}else {
			index = (index+1)%client.length;
			streak++;
		}
	}

	for(int i=0 ; i<flag.length; i++){
		if(!flag[i])
		return "";
	}
	return sequence;
}


public boolean isOneLessThanOther(int a[],int b[]) {
	for(int i=0; i< a.length; i++) {
		if(a[i]>b[i])
			return false;
	}
	return true;
}
```

```
public void copyArray(int a[],int b[]) {
        for(int i=0; i< a.length; i++) {
           a[i] = b[i];
        }
}

public void copyCustomer(Customer a[], Customer b[]) {
        for(int i=0 ; i< a.length; i++) {
                copyArray(a[i].getClientAllocation(), b[i].getClientAllocation());
                copyArray(a[i].getClientMax(), b[i].getClientMax());
                copyArray(a[i].getClientNeed(), b[i].getClientNeed());
        }
}

public static void clientState(Customer client[]){
        for(int i=0 ; i<client.length ; i++) {
                client[i].printClientStatus();
        }
        System.out.print("\n");
}

public synchronized void generateClientRequest (int client){
    do{
         for(int i=0;i<Bank.RESOURCE;i++)
                request[i]=select.nextInt(1+
                Math.min(available[i],Bank.getInstance().getCustomer(client).getClientNeed()[i]));

     }while(request[0] == 0 && request[1] == 0 && request[2] == 0);
     System.out.println("Request from client "+(client+1)+" for resource ["
                            +request[0]+"  "+request[1]+"  "+request[2]+" ]");
}


public void showResourceStatus(String type) {
        System.out.println("Resource "+type+" : [ "+available[0]+" "+ available[1]+" "+available[2]+" ]");
}

public void showAvailableResource() {
        showResourceStatus("Available");
}

public void showRemainingResource() {
        showResourceStatus("Remaining");
}
```

```
public void showCompleteStatus(int i) {
	System.out.println("Client "+(i+1)+" completes execution.");
	showAvailableResource();
}

}
```

TestBanker.java

```
public class TestBanker {

	public static void main(String[] args) {
		for(int j=0;j<5;j++) {
			Bank.getInstance().getCustomer(j).start();
		}
	}
}
```

Output

```
Request from client 1 for resource [1  3  0 ]
Resource Available : [ 10 5 7 ]
Safe sequence exist! 1 -> 2 -> 3 -> 4 -> 5 ->
Resource Remaining : [ 9 2 7 ]
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  0  0  0  Max :  3  2  2  Need :  3  2  2
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  0  0  0  Max :  2  2  2  Need :  2  2  2
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3


Request from client 4 for resource [2  1  2 ]
Resource Available : [ 9 2 7 ]
Safe sequence exist! 4 -> 1 -> 2 -> 3 -> 5 ->
Resource Remaining : [ 7 1 5 ]
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  0  0  0  Max :  3  2  2  Need :  3  2  2
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3


Request from client 5 for resource [1  1  2 ]
Resource Available : [ 7 1 5 ]
Safe sequence does not exist!
Resource Remaining : [ 7 1 5 ]
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  0  0  0  Max :  3  2  2  Need :  3  2  2
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3


Request from client 2 for resource [2  0  2 ]
Resource Available : [ 7 1 5 ]
Safe sequence exist! 4 -> 1 -> 2 -> 3 -> 5 ->
Resource Remaining : [ 5 1 3 ]
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  2  0  2  Max :  3  2  2  Need :  1  2  0
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3


Request from client 3 for resource [1  0  2 ]
Resource Available : [ 5 1 3 ]
Safe sequence exist! 4 -> 1 -> 2 -> 3 -> 5 ->
Resource Remaining : [ 4 1 1 ]
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  2  0  2  Max :  3  2  2  Need :  1  2  0
Client 3  Allocation :  1  0  2  Max :  9  0  2  Need :  8  0  0
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3
```

Analysis of output

Case 1: SAFE SEQUENCE EXISTS

```
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  0  0  0  Max :  3  2  2  Need :  3  2  2
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3


Request from client 2 for resource [2  0  2 ]
Resource Available : [ 7 1 5 ]

Safe sequence exist! 4 -> 1 -> 2 -> 3 -> 5 ->
Resource Remaining : [ 5 1 3 ]
```

Analysis-

Work = available - request;
Work =[5 1 3]
flag= [false false false false false]

Let us allocate resource to client 2 and determine the need matrix.

```
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  2  0  2  Max :  3  2  2  Need :  1  2  0
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3
```

Check for the existence of a safe sequence-

flag[1] = false; need > work
flag[2] = false; need > work
flag[3] = false; need > work
flag[4] = True; *need <= work work = work+allocation = [7 2 5]*
flag[5] = false; need > work
flag[1] = true; *need<=work work = work+allocation = [8 5 5]*
flag[2] = true; *need<=work work = work+allocation = [10 5 7]*
flag[3] = true; *need<=work work = work+allocation = [10 5 7]*
flag[5] = true; *need<=work work = work+allocation = [10 5 7]*

Hence Safe sequence exists and is 0→ 1→ 2→ 3→ 3 → 4

Case 2: SAFE SEQUENCE DOESN'T EXIST

```
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  0  0  0  Max :  3  2  2  Need :  3  2  2
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  0  0  0  Max :  4  3  3  Need :  4  3  3


Request from client 5 for resource [1  1  2 ]
Resource Available : [ 7 1 5 ]
```

```
Safe sequence does not exist!
```

Analysis-

Work = available - request;
Work =[6 0 3]
flag= [false false false false false]

Let us allocate resource to client 2 and determine the need matrix.

```
Client 1  Allocation :  1  3  0  Max :  7  5  3  Need :  6  2  3
Client 2  Allocation :  0  0  0  Max :  3  2  2  Need :  3  2  2
Client 3  Allocation :  0  0  0  Max :  9  0  2  Need :  9  0  2
Client 4  Allocation :  2  1  2  Max :  2  2  2  Need :  0  1  0
Client 5  Allocation :  1  1  2  Max :  4  3  3  Need :  3  2  1
```

Check for the existence of a safe sequence-

flag[1] = false; need > work
flag[2] = false; need > work
flag[3] = false; need > work
flag[4] = false; need > work
flag[5] = false; need > work

Hence Safe sequence does not exist.

This output is as expected and code is tested for above cases. Hence we were able to implement the Banker's algorithm successfully.