UNIVERSITETI I PRISHTINËS “HASAN PRISHTINA”

FAKULTETI I INXHINIERISË ELEKTRIKE DHE KOMPJUTERIKE

Banker’s Algorithm

Lënda: Sisteme Operative

Profesor: Artan Mazrekaj

Asistente: Mihrije Kadriu

Punuan:

Adea Lluhani

Agnesa Hulaj

Rozeta Meshi

Yllka Kastrati

Zana Ademi

Prishtinë, 2024

Table of Contents

Hyrje

Algoritmi i Bankier-it është një metodë e përdorur në sistemet operative për të menaxhuar alokimin e resurseve dhe për të shmangur gjendjen e bllokimit (deadlock). Deadlock – u është një situate ku dy programe kompjuterike që ndajnë te njëjtin resurs bllokojnë njëri-tjetrin nga qasja në resurs, duke rezultuar në ndërprejen e funksionimit të dy programeve.

Ky algoritëm u propozua nga Edsger Dijkstra dhe emri “Bankieri” vjen nga analogjia e një bankieri që vendos nëse një person duhet të marrë një kredi për të ndihmuar një sistem bankar të simulojë në mënyrë të sigurt alokimin e resurseve.

Parimet kryesore të Algoritmit të Bankierit:

1. **Siguria e Sistemit** – Para se t'i japë burimet e kërkuara një procesi, sistemi verifikon nëse do të mbetet në një gjendje të sigurt pas alokimit. Një gjendje e sigurt është ajo ku ka një sekuencë të mundshme ekzekutimi për të gjithë proceset e mbetura, pa shkaktuar deadlock.
2. **Kërkesa për resurse** – Proceset mund të bëjnë kërkesa për resurse në çdo kohë dhe gjithashtu mund të lirojnë resurse. Algoritmi siguron që këto kërkesa të përmbushen vetëm nëse gjendja e sistemit mbetet e sigurt.
3. **Data Structures** – Algoritmi përdor matrica për të përfaqësuar kërkesat maksimale të çdo procesi, resurset aktualisht të alokuara dhe resurset që janë ende të nevojshme. Këto informacione përdoren për të llogaritur nëse kërkesat e reja mund të plotësohen pa rrezikuar sigurinë e sistemit. Ndërsa vargjet përdoren të ruajtur numrin e resurseve të disponueshme.
4. **Kontrolli i Sigurisë** – Për çdo kërkesë të re, algoritmi kontrollon nëse pas alokimit, sistemi do të jetë ende në një gjendje të sigurt. Nëse po, kërkesa aprovohet. Përndryshe, ajo vendoset në pritje deri sa të lirohen mjaft resurse.

Algoritmi i Bankierit iu mundëson sistemeve operative të menaxhojnë resurset në mënyrë efikase dhe të shmangin deadlocks që mund të ndikojnë në performancën dhe stabilitetin e sistemit.

1. Përshkrimi i problemit

1.1. Banker's Algorithm

Si u përmend më lartë Algoritmi Banker në Sistemet Operative përdoret për të parandaluar deadlocks dhe për të siguruar që sistemet të mbeten në një gjendje të sigurtë.
Algoritmi funksionon duke mbajtur gjurmët e resurseve të disponueshme në sistem dhe duke i shpërndarë ato në procese në një mënyrë që siguron që sistemit të mos i mbarojnë resurset dhe të hyjë në një gjendje bllokimi(deadlocku). Algoritmi i bankierit është quajtur kështu sepse përdoret në sistemet bankare për të kontrolluar nëse një kredi mund t'i sanksionohet një personi apo jo.

Për të kuptuar më mirë këtë algoritem le të marrim një shembull të botës reale.Supozojmë se ka n përdorues të llogarive bankare në një bankë të caktuar dhe shuma totale e parave të tyre është S. Tani, nëse një person aplikon për një kredi (të themi, për të blerë një shtëpi), shuma e kredisë e reduktuar nga shuma totale e disponueshme në bankë na jep shumën e mbetur, e cila duhet të jetë më shumë se S në mënyrë që kredia të të sanksionohet nga banka.Kjo analogji përdoret njëjtë në Sistemet Operative për shmangien dhe parandalimin e deadlocks .
Çdo proces brenda sistemit duhet t'i dërgojë sistemit operativ të gjitha detajet e nevojshme të rëndësishme të tilla si kërkesat për resurse, vonesat(delays) etj. Bazuar në këto detaje, sistemi operativ përcakton nëse do të ekzekutojë procesin ose do ta mbajë atë në gjendje pritjeje për të shmangur bllokimet e sistemit.

1.1.1. Si funksionon Banker's Algorithm në SO?

Banker's Algorithm në SO funksionon duke mbajtur një matricë maksimale të resurseve (matrix of maximum) dhe resurseve të alokuara për çdo proces,dhe më pas kontrollon nëse sistemi është në një gjendje të sigurt përpara se të lejojë një proces të kërkojë resurse shtesë. Algoritmi kontrollon nëse kërkesa mund të plotësohet pa kompromentuar sigurinë e sistemit, duke siguruar që kërkesa nuk bën që një proces të tejkalojë nevojat e tij maksimale për resurse dhe se ka resurse të mjaftueshme në dispozicion për të plotësuar kërkesën.

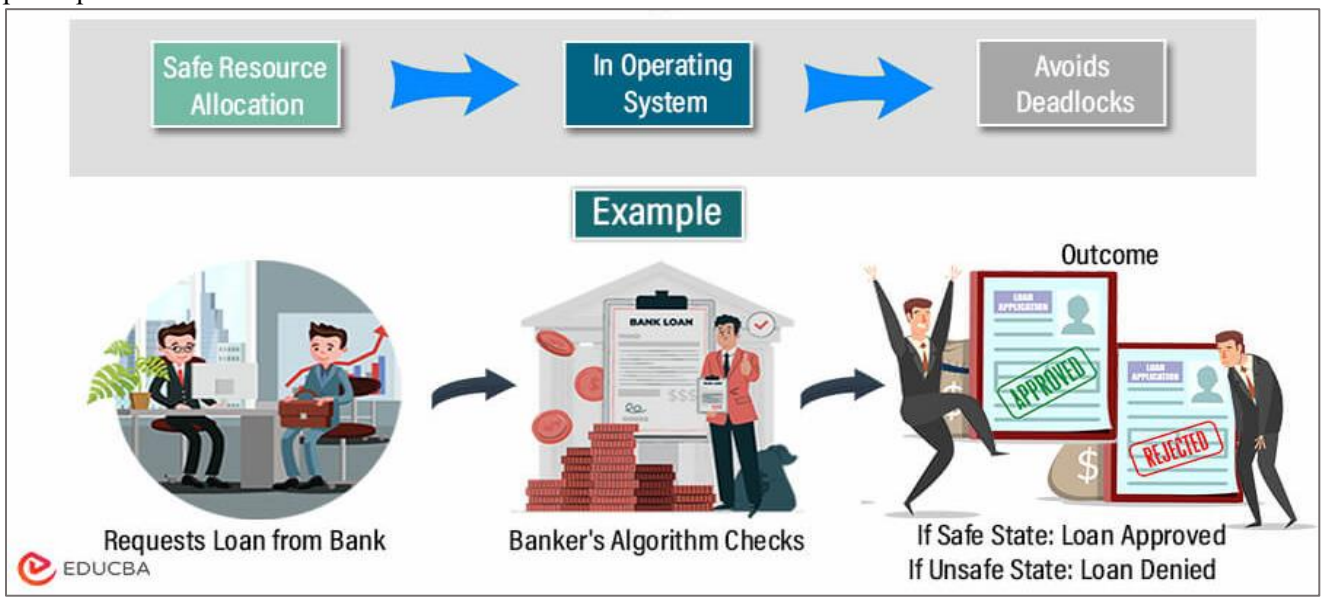


Figura 1: Banker's Algorithm

Kur e implementojmë ose e trajtojmë Algoritmin e Bankers në SO, duhet të kemi parasysh tre faktorë:

1. Matrica **[MAX]** tregon se sa burime të secilit lloj mund të kërkojë një proces.
2. Matrica **[ALOCATION]** tregon se sa burime të secilit lloj një proces mban ose alokon aktualisht.
1. Vargu **[AVAILABLE]** tregon se sa burime të secilit lloj janë aktualisht në dispozicion në sistem.

Algoritmi i Bankers përbëhet nga dy algoritme që janë:

1. Safety Algorithm
2. Resource Request Algorithm

1.1.2. Safety Algorithm

Ky algoritwm përdoret për të përcaktuar nëse një sistem është në një gjendje të sigurt dhe nëse një kërkesë për resurse mund të jepet pa rezultuar në bllokim(deadlock).
Hapat që duhet të ndjeken në këtë algoritëm janë:

1. Le të jenë Work dhe Finish vektorë me gjatësi m dhe n respektivisht. Atëherë këta dy vektorë inicializohen si në vijim:
   Work=Available
   Finish[i]=false për i=1,2,3…n
2. Gjejmë një i për të cilën plotësohen dy kushtet
   Finish[i]=false
   Need[i]<=Work
   Nëse nuk ekziston një i e tillë, shkojmë te hapi 4
3. Work=Work+Allocation[i]
   Finish[i]=true
   Kthehemi te hapi 2
4. Nëse Finish[i]=true për të gjithë i – të, atëherë sistemi është në gjendje të sigurt.

Safety Algorithm kontrollon në mënyrë iteruese nëse ka ndonjë process që mund të përfundojë me resurset aktualisht të lira (Work). Nëse një proces i tillë gjendet, resurset e tij të alokuara kthehen te Work dhe procesi konsiderohet i përfunduar (Finish[i]=true). Ky process vazhdon derisa të përfundojnë të gjitha proceset ose derisa të mos mund të gjendet më asnjë process që mund të përfundojë me resurset e disponueshme. Nëse të gjithë proceset përfundojnë me sukses, sistemi është në gjendje të sigurt. Përndryshe, sistemi nuk është në gjendje të sigurt.

1.1.3. Resource Request Algorithm

Ky algoritwm përdoret për të menaxhuar shpërndarjen e burimeve. Kur një proces kërkon burime shtesë, ky algoritëm përcakton nëse kërkesa mund të plotësohet pa kompromentuar sigurinë e sistemit.
Nëse shkruajmë me $Request_i$ kërkesën për resurse të procesit P_i, Request[j]=k do të thotë se procesi P kërkon k instancë të resursit Rj. Kur bëhet kërkesë për burime nga procesi P duhet të ndjeken këta hapa:

1. Nëse $Request_i <= Need_i$
   Shkojmë te hapi 2, përndryshe nuk plotësohet kërkesa, meqë procesi ka tejkaluar kërkesën e tij maksimale për resurse.
2. Nëse $Request_i$ <= Available
   Vazhdojmë me hapin 3, përndryshe nuk plotësohet kërkesa, meqë procesi ka bërë kërkesë për resurse më shumë se që ka resurse të disponueshme.
3. Sistemi përkohësisht ia alokon resurset e kërkuara procesit P dhe bën ndryshimet si më poshtë

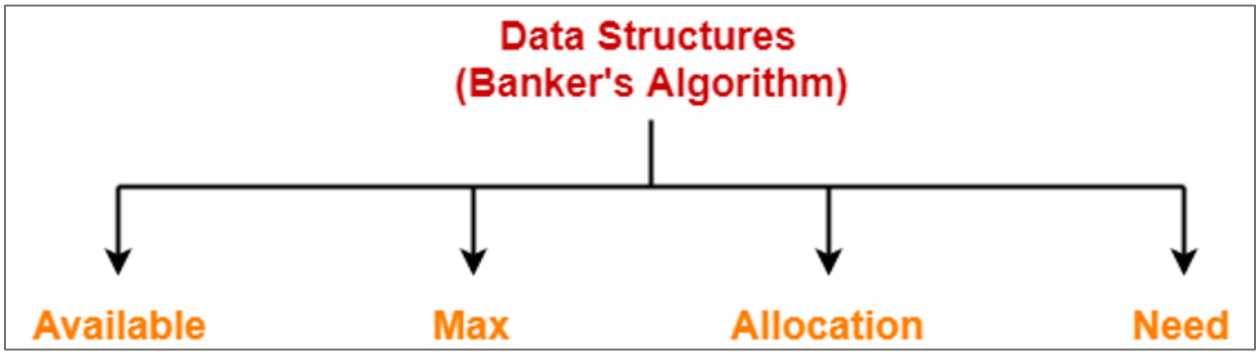
Available=Available-Request$_i$
Allocation$_i$ =Allocation$_i$ +Request$_i$
Need$_i$ =Need$_i$ -Request$_i$

4. Pas këtyre ndryshimeve kontrollohet nëse sistemi është në safe state. Nëse sistemi është në safe state, procesit i alokohen resurset e kërkuara, përndryshe procesit i refuzohet kërkesa për resurse.

Algoritmi kontrollon fillimisht nëse kërkesa e procesit për resurse është brenda kërkesës maksimale të tij (**Need$_i$**). Nëse jo, procesit nuk i alokohen resurset pasi procesi kërkon më shume resurse sesa i janë lejuar. Nëse kërkesa është e vlefshme, algoritmi kontrollon nëse resurse e kërkuara janë të disponueshme aktualisht. Nëse burimet nuk janë të disponueshme, procesi duhet të presë. Nëse resurset janë të disponueshme, sistemi përkohësisht i alokon ato resurse duke përditësuar **Available**, **Allocation$_i$**, dhe **Need$_i$** si më sipër. Pas këtyre ndryshimeve nëse sistemi është në safe state, procesit i alokohen resurset, përndryshe kërkesa e tij për resurse nuk plotësohet.

1.1.4. Data Structures required in the Banker's Algorithm

1. *Vargu i burimeve të disponueshme(Available Resource Array)*: Një varg që ruan numrin aktual të burimeve të disponueshme të secilit lloj.
2. *Matrica e nevojës maksimale:* Një matricë që ruan numrin maksimal të burimeve të çdo lloji të kërkuar nga secili proces.
3. *Matrica e Alokimit(Allocation Matrix)*: Një matricë që ruan numrin e burimeve të secilit lloj të alokuar aktualisht për secilin proces.
4. *Matrica e Nevojës(Need Matrix)*: Një matricë që ruan burimet e mbetura të secilit lloj të kërkuar nga secili proces (llogaritur si diferencë midis matricave te nevojes maksimale dhe matrices se alokimit).
5. *Seti i proceseve të përfunduara(terminuara):* Një grup që ruan ID-të e proceseve që kanë përfunduar dhe lëshuar burimet e tyre.



1.1.5. Çfarë është një gjendje e sigurt në kontekstin e Algoritmit të Bankers?

Një gjendje e sigurt në kontekstin e Algoritmit të Bankers në OS është një gjendje në të cilën ekziston një sekuencë procesesh të tilla që çdo proces mund të terminohet pa hasur në deadlocks. Me fjalë të tjera, është një gjendje në të cilën të gjitha proceset mund të ekzekutohen deri në përfundim pa asnjë bllokim ose mungesë tw resurseve.

1.1.6. Çfarë është një gjendje e pasigurt në kontekstin e Algoritmit të Bankers?

Një gjendje e pasigurt në kontekstin e Algoritmit të Bankers në OS është një gjendje në të cilën ekzekutimi i proceseve mund të rezultojë në bllokime ose mungesë te resurseve. Në një gjendje të pasigurt, proceset mund të bllokohen duke pritur për resurse që nuk do të bëhen kurrë të disponueshme, duke rezultuar në një bllokim.

1.1.7. Avantazhet

- Gjendja e sigurisë: Algoritmi i Bankers kontrollon nëse gjendja aktuale është një gjendje e sigurt, ku nuk do të ndodhë asnjë bllokim edhe nëse të gjitha proceset vazhdojnë të shkojnë drejt përfundimit.
- Stabiliteti i sistemit: Algoritmi i Bankers ndihmon në ruajtjen e stabilitetit të sistemit duke siguruar që proceset të mos qëndrojnë në burime për një kohë të pacaktuar.

1.1.8. Disavantazhet

- Zbatueshmëria e kufizuar(Restricted usability): Algoritmi i Bankers në OS është i kufizuar në zbatueshmërinë e tij vetëm për ato sisteme me një numër të kufizuar burimesh dhe mund të mos jetë i përshtatshëm për sisteme me burime të vazhdueshme ose të pafundme.
- Overhead: Algoritmi Bankers kërkon llogaritje shtesë dhe shpenzime të memories për të ruajtur strukturat e të dhënave dhe për të kontrolluar për gjendje të sigurta, duke ndikuar në performancën e sistemit.

1.2. Example

Answer the following questions using the banker's algorithm:

Threads	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
T_0	0	0	1	2	0	0	1	2	1	5	2	0
T_1	1	0	0	0	1	7	5	0				
T_2	1	3	5	4	2	3	5	6				
T_3	0	6	3	2	0	6	5	2				
T_4	0	0	1	4	0	6	5	6				

1.2.1 What is the content of the matrix Need?

The matrix Need is calculated as: Need[i,j]=Max[i,j]-Allocation[i.j]

Need Matrix				
	A	B	C	D
T_0	0	0	0	0
T_1	0	7	5	0
T_2	1	0	0	2
T_3	0	0	2	0
T_4	0	6	4	2

1.2.2 Is the system in a safe state?

To check whether the system is in a safe state we use the Safety Algorithm

m=3 number of resources n=5 number of threads Work=Available Work= (1,5,2,0)	Step 4: Check T_3 Need[T_3] = (0,0,2,0) ≤ Work = (2,8,8,6) T_3 can be satisfied. Allocate resources to T_3 and release them back. Work=Work+Allocation[T_3] = (2,8,8,6) + (0,6,3,2) = (2,14,11,8)
Step 1: Check T_0 Need[T_0] = (0,0,0,0) ≤ Work = (1,5,2,0) T_0 can be satisfied. Allocate resources to T_0 and release them back. Work=Work+Allocation[T_0] = (1,5,2,0) + (0,0,1,2) = (1,5,3,2)	Step 5: Check T_4 Need[T_4] = (0,6,4,2) ≤ Work = (2,14,11,8) T_4 can be satisfied. Allocate resources to T_4 and release them back. Work=Work+Allocation[T_4] = (2,14,11,8) + (0,0,1,4) = (2,14,12,12)
Step 2: Check T_1 Need[T_1] = (0,7,5,0) ≰ Work = (1,5,2,0) T_1 must wait	Step 6: Check T_1 again Need[T_1] = (0,7,5,0) ≤ Work = (2,14,12,12) T_1 can be satisfied. Allocate resources to T_1 and release them back. Work=Work+Allocation[T_1] = (2,14,12,12) + (1,0,0,0) = (3, 14,12,12
Step 3: Check T_2 Need[T_2] = (1,0,0,2) ≤ Work = (1,5,3,2) T_2 can be satisfied. Allocate resources to T_2 and release them back.	Since all threads can be satisfied, the system is in a safe state. The safe sequence is T_0, T_2, T_3, T_4, T_1

Work=Work+Allocation[T_2] = (1,5,3,2) + (1,3,5,4) = (2,8,8,6)	

1.2.3 If a request from T1 arrives for (0,4,2,0), can the request be granted immediately?

To decide whether the request is granted we use the Resource Request Algorithm

Request = (0,4,2,0)

Step 1: Request[T_1] = (0,4,2,0) < Need[T_1] = (0,7,5,0)	Step 3: Available=Available-Request[T_1] Allocation[T_1] =Allocation[T_1] +Request[T_1] Need[T_1] =Need[T_1] -Request[T_1]
Step 2: Request[T_1] = (0,4,2,0) < Available = (1,5,2,0)	

Threads	Allocation				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
T_0	0	0	1	2	0	0	0	0	**1**	**1**	**0**	**0**
T_1	**1**	**4**	**2**	**0**	**0**	**3**	**3**	**0**				
T_2	1	3	5	4	1	0	0	2				
T_3	0	6	3	2	0	0	2	0				
T_4	0	0	1	4	0	6	4	2				

We use the Safety algorithm to determine if the new system is in a safe state.

m=3 number of resources n=5 number of threads Work=Available Work= (1,1,0,0)	Step 4: Check T_3 Need[T_3] = (0,0,2,0) ≤ Work = (2,4,6,6) T_3 can be satisfied. Allocate resources to T_3 and release them back. Work=Work+Allocation[T_3] = (2,4,6,6) + (0,6,3,2) = (2,10,9,8)
Step 1: Check T_0 Need[T_0] = (0,0,0,0) ≤ Work = (1,1,0,0) T_0 can be satisfied. Allocate resources to T_0 and release them back. Work=Work+Allocation[T_0] = (1,1,0,0) + (0,0,1,2) = (1,1,1,2)	Step 5: Check T_4 Need[T_4] = (0,6,4,2) ≤ Work = (2,10,9,8) T_4 can be satisfied. Allocate resources to T_4 and release them back. Work=Work+Allocation[T_4] = (2,10,9,8) + (0,0,1,4) = (2,10,10,12)
Step 2: Check T_1 Need[T_1] = (0,3,3,0) ≰ Work = (1,1,0,0) T_1 must wait	Step 6: Check T_1 again Need[T_1] = (0,3,3,0) ≤ Work = (2,10,10,12) T_1 can be satisfied. Allocate resources to T_1 and release them back. Work=Work+Allocation[T_1] = (2,10,10,12) + (1,4,2,0) = (3, 14,12,12
Step 3: Check T_2 Need[T_2] = (1,0,0,2) ≤ Work = (1,1,1,2) T_2 can be satisfied. Allocate resources to T_2 and release them back. Work=Work+Allocation[T_2] = (1,1,1,2) + (1,3,5,4) = (2,4,6,6)	Since all threads can be satisfied, the system is in a safe state. The safe sequence is T_0, T_2, T_3, T_4, T_1 The request from T_1 can be granted immediately.

2. Teknologjia

Për implementimin e algoritmit Banker kemi përdorur gjuhën programuese Java. Për qëllime thjesht fillimisht kodin e kemi implementuar në Intellij në Windows dhe pastaj të njëjtin kod e kemi shkruar në fajllat përkatës në Ubuntu. Meqenëse për kompajllimin e kodit në Java përdoret Java Development Kit fillimisht kemi instaluar JDK duke përdorur komandat *sudo apt update* dhe *sudo apt install default-jdk.*

3. Implementimi

3.1. Kodi

Për implementimin e algoritmit Banker kemi krijuar një klasë BankersAlgorithm që përmban këto veçori:

- allocation – matricë që shërben për ruajtjen e numrit të resurseve që i janë alokuar secilit thread
- max – matricë që ruan kërkesat maksimale të resurseve për secilin thread
- available – varg që ruan numrin e instancave të lira për secilin proces
- need – matricë që ruan nevojën për resurse të secilit thread
- numThreads – paraqet numrin e threads (ose proceseve)
- numResources – paraqet numrin e resurseve

Pastaj e kemi përdorur konstruktorin për inicializimin e këtyre veçorive, me ç'rast për inicializimin e matricës need e kemi përdorur metodën calculateNeed ().

```
import java.util.Arrays;

public class BankersAlgorithm {
    private final int[][] allocation;
    private final int[][] max;
    private final int[] available;
    private final int[][] need;
    private final int numThreads;
    private final int numResources;

 // Constructor to initialize the Banker's Algorithm with the allocation, max, and available matrices
    public BankersAlgorithm(int[][] allocation, int[][] max, int[] available) {
        this.allocation = allocation;
        this.max = max;
        this.available = available;
        this.numThreads = allocation.length;
        this.numResources = available.length;
        this.need = calculateNeed();
    }
```

Figura 3: Veçoritë dhe konstruktori i klasës BankersAlgorithm

Metoda calculateNeed () shërben për gjetjen e matricës need, që paraqet numrin e resurseve që secilin thread mund ta kërkojë prapë. Kjo matricë kalkulohet si need=max – allocation.

```java
// Calculate the Need matrix: Need[i,j] = Max[i,j] - Allocation[i,j]
private int[][] calculateNeed() {
    int[][] need = new int[numThreads][numResources];
    for (int i = 0; i < numThreads; i++) {
        for (int j = 0; j < numResources; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
    return need;
}
```

Figura 4: calculateNeed() Method

Metoda isSafe () përdoret për të kontrolluar nëse sistemi është në safe state. Brenda kësaj metode inicializohen vargu ëork që ruan numrin e resurseve të lira; vargu finish që shërben për të kontrolluar se cilat procese/threads mund të përfundojnë; vargu safeSequence e ruan safe order of threads dhe count bën numrimin e threads që mund të përfundojnë. Kjo metodë iteron nëpër threads për të gjetur një safe sequence në të cilën secilin thread mund të përfundojë nëse i jepen burimet e nevojshme. Nëse nevojat e thread – it mund të plotësohen nga resurset e lira, atëherë atij thread – i alokohen burimet dhe themi se ky thread ka përfunduar. Pastaj lirohen resurset përsëri. Nëse asnjë thread nuk mund të kryhet themi se sistemi nuk është në safe state. Në fund nëse sistemi është në safe state e printon sekuencën në të cilën sistemi është safe.

```java
// Check if the system is in a safe state with detailed output
public boolean isSafeState() {
    int[] work = Arrays.copyOf(available, available.length);
    boolean[] finish = new boolean[numThreads];
    String[] safeSequence = new String[numThreads];
    int count = 0;

    System.out.println("Initial Available: " + Arrays.toString(work));
    System.out.println("Initial Need Matrix:");
    for (int[] row : need) {
        System.out.println(Arrays.toString(row));
    }

    while (count < numThreads) {
        boolean found = false;
        for (int i = 0; i < numThreads; i++) {
            if (!finish[i]) {
                System.out.println("Step " + (count + 1) + ": Check T" + i);
                System.out.println("Need[T" + i + "] = " + Arrays.toString(need[i]) + " ≤ Available = " + Arrays.toString(work));
                if (canSatisfy(need[i], work)) {
                    System.out.println("T" + i + " can be satisfied. Allocate resources to T" + i + " and release them back.");
                    for (int k = 0; k < numResources; k++) {
                        work[k] += allocation[i][k];
                    }
                    finish[i] = true;
                    safeSequence[count++] = "T" + i;
                    found = true;
                    System.out.println("Available = " + Arrays.toString(work));
                    System.out.println("----------------------------------------");
                } else {
                    System.out.println("T" + i + " must wait.");
                    System.out.println("----------------------------------------");
                }
            }
        }
        if (!found) break;
    }

    boolean isSafe = (count == numThreads);
    if (isSafe) {
        System.out.println("System is in a safe state.");
        System.out.println("Safe sequence is: " + Arrays.toString(safeSequence));
    } else System.out.println("System is not in a safe state.");

    return isSafe;
}
```

Figura 5: isSafe () Method

Kjo metodë shërben për të kontrolluar nëse resurset e lira aktualisht mund të plotësojnë nevojat e një thread.

```java
// Method to check if resources can satisfy the need
private boolean canSatisfy(int[] need, int[] available) {
    for (int j = 0; j < need.length; j++) {
        if (need[j] > available[j]) return false;
    }
    return true;
}
```

Figura 6: canSatisfy () Method

Metoda requestResources () përdoret për të kontrolluar nëse kërkesat për resurse të një thread – i mund të plotësohen. Fillimisht shikon nëse kërkesa për resurse është më e madhe se resurset e nevojshme për atë thread. Pastaj shikon nëse resurset e kërkuara i tejkalojnë resurset e lira. Pas këtyre dy hapave përkohësisht i alokohen resurset thread – it dhe përdoret isSafe () metoda për të kontrolluar nëse sistemi është në safe state. Nëse sistemi është në safe state, atëherë kërkesa për resurse plotësohet, përndryshe kërkesa refuzohet.

```java
// Request resources for a given thread
public boolean requestResources(int threadNum, int[] request) {
    // Check if the request is greater than the need
    for (int j = 0; j < numResources; j++) {
        if (request[j] > need[threadNum][j]) {
            System.out.println("Thread " + threadNum + " has exceeded its maximum claim.");
            return false;
        }
    }

    // Check if the request is greater than the available resources
    for (int j = 0; j < numResources; j++) {
        if (request[j] > available[j]) {
            System.out.println("Thread " + threadNum + "'s request cannot be granted because there are not enough available resources.");
            return false;
        }
    }

    // Allocate the requested resources
    for (int j = 0; j < numResources; j++) {
        available[j] -= request[j];
        allocation[threadNum][j] += request[j];
        need[threadNum][j] -= request[j];
    }

    // Check if the system is in a safe state after allocation
    if (isSafeState()) {
        System.out.println("Request by thread " + threadNum + " can be granted immediately.");
        return true;
    } else {
        // Roll back the allocation
        for (int j = 0; j < numResources; j++) {
            available[j] += request[j];
            allocation[threadNum][j] -= request[j];
            need[threadNum][j] += request[j];
        }
        System.out.println("Request by thread " + threadNum + " cannot be granted immediately. Rolling back.");
        return false;
    }
}
```

Figura 7: requestResources () Method

Në figurën e mëposhtme është paraqitur klasa Main me metodën main (), në të cilën fillimisht i kemi inicializuar matricën allocation, max dhe vargun available për shembullin tonë. Pastaj kemi krijuaj një instancë të Klasës BankersAlgorithm dhe kemi thirrur metodën isSafeState () për të kontrolluar nëse sistemi është në safe state.

Tutje pyesim nëse ndonjë prej threads kërkon resurse shtesë dhe nëse kërkon identifikojmë thread – in që ka kërkuar resurse dhe resurset që ka kërkuar ai thread. Pastaj thërasim metodën requestResources () dhe thread – it i alokohen resurset nëse sistemi është në safe state, që në rastin tonë thread – it T1 i janë alokuar resurset e kërkuara.

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        int[][] allocation = {{0, 0, 1, 2}, {1, 0, 0, 0}, {1, 3, 5, 4}, {0, 6, 3, 2}, {0, 0, 1, 4}};

        int[][] max = {{0, 0, 1, 2}, {1, 7, 5, 0}, {2, 3, 5, 6}, {0, 6, 5, 2}, {0, 6, 5, 6}};

        int[] available = {1, 5, 2, 0};

        BankersAlgorithm ba = new BankersAlgorithm(allocation, max, available);

        ba.isSafeState();

        Scanner sc = new Scanner(System.in);
        System.out.println("Do you want to request resources for a thread? Type y/n");
        char ans = sc.next().charAt(0);
        if (ans == 'y') {
            System.out.println("Enter the number of the thread that requires resources:");
            int threadNum = sc.nextInt();
            System.out.println("Enter the resources required by T[" + threadNum + "]");
            int[] request = new int[available.length];
            for (int i = 0; i < available.length; i++) {
                request[i] = sc.nextInt();
            }
            ba.requestResources(threadNum, request);
        } else {
            System.out.println("No resource request made. Exiting program.");
        }

    }
}
```

Figura 8: Main Class

3.2. Ekzekutimi

Në fgurën e mëposhtme janë paraqitur komandat për krijimin e dy fajllave që përmbajnë klasën BankersAlgorithm dhe Main. Pas krijimit të tyre shkruhet kodi i mësipërm në fajllat përkatës.

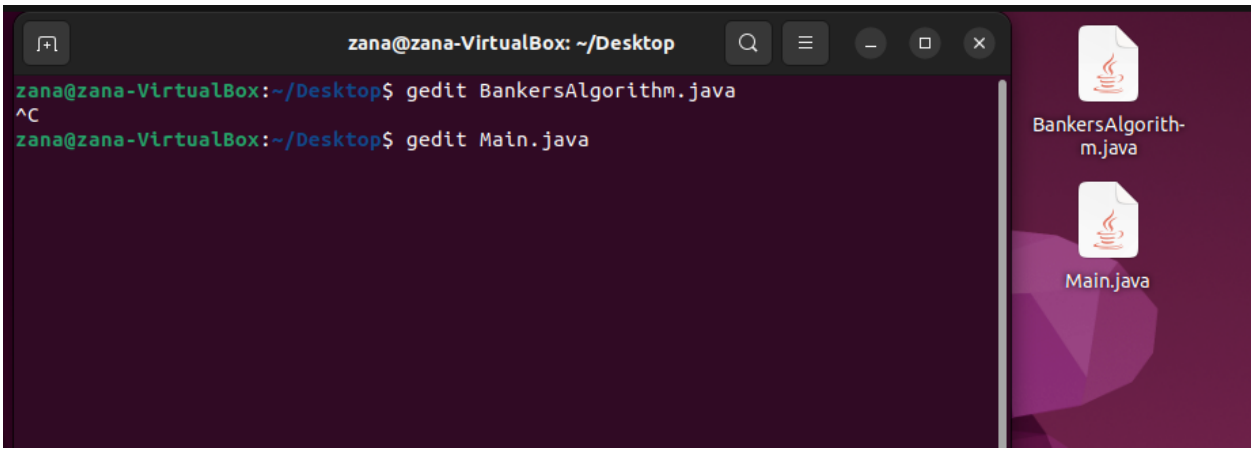


Figura 9: Krijimi i fajllave Main dhe BankersAlgorithm

javac Main.java – përdoret për kompajllimin e fajllit Main.java me ç'rast krijohen dy fajlla të ri sin ë figurë BankersAlgorithm.class dhe Main.class

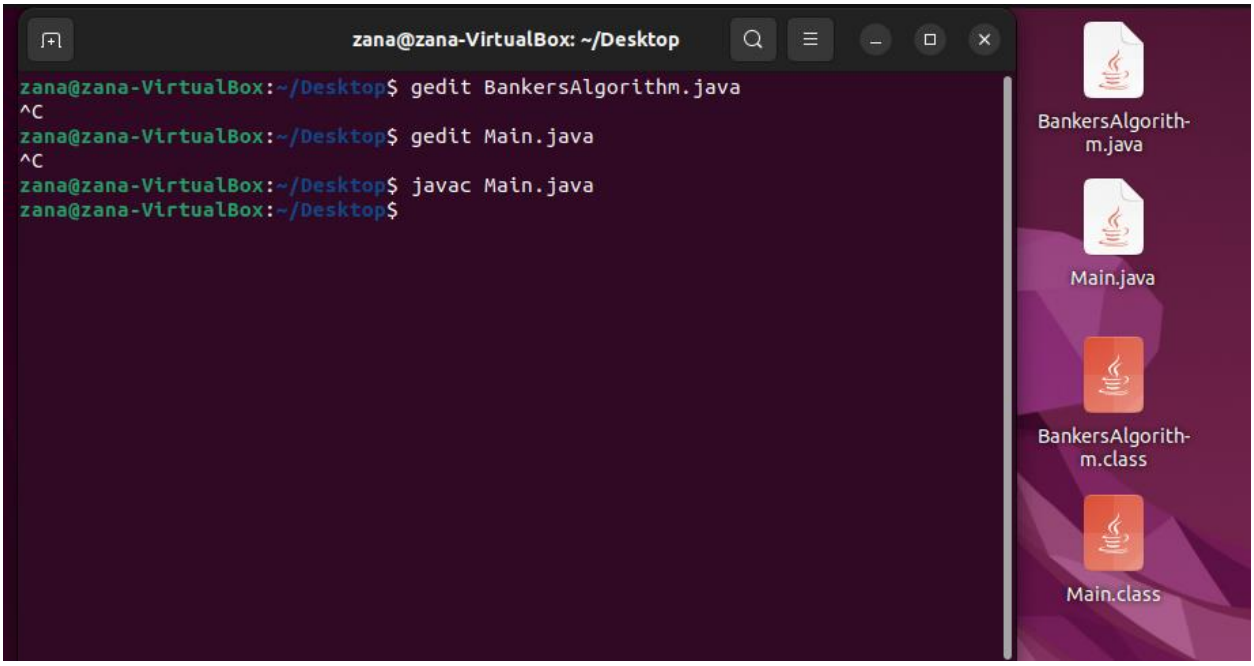


Figura 10: Kompajllimi i fajllit Main

Komanda java Main përdoret për ekzekutimin (run) e fajllit të kompajlluar. Pastaj shfaqet rezultati hap pas hap i alokimit të resurseve për secilin thread. Në fund shfaqet sekuenca në të cilën sistemi është safe.

```
zana@zana-VirtualBox:~/Desktop$ gedit BankersAlgorithm.java
^C
zana@zana-VirtualBox:~/Desktop$ gedit Main.java
^C
zana@zana-VirtualBox:~/Desktop$ javac Main.java
zana@zana-VirtualBox:~/Desktop$ java Main
Initial Available: [1, 5, 2, 0]
Initial Need Matrix:
[0, 0, 0, 0]
[0, 7, 5, 0]
[1, 0, 0, 2]
[0, 0, 2, 0]
[0, 6, 4, 2]
Step 1: Check T0
Need[T0] = [0, 0, 0, 0] ≤ Available = [1, 5, 2, 0]
T0 can be satisfied. Allocate resources to T0 and release them back.
Available = [1, 5, 3, 2]
---------------------------------------
Step 2: Check T1
Need[T1] = [0, 7, 5, 0] ≤ Available = [1, 5, 3, 2]
T1 must wait.
---------------------------------------
Step 2: Check T2
Need[T2] = [1, 0, 0, 2] ≤ Available = [1, 5, 3, 2]
T2 can be satisfied. Allocate resources to T2 and release them back.
Available = [2, 8, 8, 6]
---------------------------------------
Step 3: Check T3
Need[T3] = [0, 0, 2, 0] ≤ Available = [2, 8, 8, 6]
T3 can be satisfied. Allocate resources to T3 and release them back.
Available = [2, 14, 11, 8]
---------------------------------------
Step 4: Check T4
Need[T4] = [0, 6, 4, 2] ≤ Available = [2, 14, 11, 8]
T4 can be satisfied. Allocate resources to T4 and release them back.
Available = [2, 14, 12, 12]
---------------------------------------
Step 5: Check T1
Need[T1] = [0, 7, 5, 0] ≤ Available = [2, 14, 12, 12]
T1 can be satisfied. Allocate resources to T1 and release them back.
Available = [3, 14, 12, 12]
---------------------------------------
System is in a safe state.
Safe sequence is: [T0, T2, T3, T4, T1]
Do you want to request resources for a thread? Type y/n
```

Figura 11: Ekzekutimi i programit – kontrollimi nëse sistemi është në safe state

Në këtë pjesë trajtohet rasti kur një thread kërkon resurse. Thread – it përkohësisht i alokohen resurset e kërkuara, dhe kontrollohet nëse sistemi është në safe state. Në rastin tonë sistemi ka qenë në safe state dhe prandaj thread – it ju kanë alokuar resurset dhe është printuar në fund sekuenca në të cilën sistemi është në safe state.

```
y
Enter the number of the thread that requires resources:
1
Enter the resources required by T[1]
0
4
2
0
Initial Available: [1, 1, 0, 0]
Initial Need Matrix:
[0, 0, 0, 0]
[0, 3, 3, 0]
[1, 0, 0, 2]
[0, 0, 2, 0]
[0, 6, 4, 2]
Step 1: Check T0
Need[T0] = [0, 0, 0, 0] ≤ Available = [1, 1, 0, 0]
T0 can be satisfied. Allocate resources to T0 and release them back.
Available = [1, 1, 1, 2]
---------------------------------------
Step 2: Check T1
Need[T1] = [0, 3, 3, 0] ≤ Available = [1, 1, 1, 2]
T1 must wait.
---------------------------------------
Step 2: Check T2
Need[T2] = [1, 0, 0, 2] ≤ Available = [1, 1, 1, 2]
T2 can be satisfied. Allocate resources to T2 and release them back.
Available = [2, 4, 6, 6]
---------------------------------------
Step 3: Check T3
Need[T3] = [0, 0, 2, 0] ≤ Available = [2, 4, 6, 6]
T3 can be satisfied. Allocate resources to T3 and release them back.
Available = [2, 10, 9, 8]
---------------------------------------
Step 4: Check T4
Need[T4] = [0, 6, 4, 2] ≤ Available = [2, 10, 9, 8]
T4 can be satisfied. Allocate resources to T4 and release them back.
Available = [2, 10, 10, 12]
---------------------------------------
Step 5: Check T1
Need[T1] = [0, 3, 3, 0] ≤ Available = [2, 10, 10, 12]
T1 can be satisfied. Allocate resources to T1 and release them back.
Available = [3, 14, 12, 12]
---------------------------------------
System is in a safe state.
Safe sequence is: [T0, T2, T3, T4, T1]
Request by thread 1 can be granted immediately.
zana@zana-VirtualBox:~/Desktop$
```

Figura 12: Ekzekutimi i programit – kërkimi i resurseve nga T_1

4. Konkluzione

Përdorimi i algoritmit Banker ne sisteme operative sjell përparësi në menaxhimin e burimeve sepse siguron në mënyrë efikase menaxhimin e burimeve si memoria dhe procesori në një mjedis me procese të ndërsjellta. Nëpërmjet këtij algoritmi parandalojmë situata si mbyllja e rreptë ose mungesa e burimeve, duke garantuar qe proceset do të kenë aksesin e nevojshëm për të përfunduar punën e tyre pa ndërprerje. Jo gjithnjë përdorimi i algoritmit Banker është zgjidhje e mirë kjo pasi që mund të ketë kosto të lartë të overhead-it ose nevojë për optimizim të vazhdueshëm për të përballuar ndryshimet në kërkesat e sistemit, për këtë arsye eksplorimi i teknologjisë se menaxhimit të burimeve në sisteme operative vazhdon të jetë një fushë e rëndesishme e kërkimit dhe zhvillimit.

5. Referencat

[1] "Geeksforgeeks". Available: https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system-2/

[2] "Prepbytes Blog". Available: https://www.prepbytes.com/blog/operating-system/bankers-algorithm-in-os/

[3] "Operating System Concepts" Tenth Edition. Available: https://drive.google.com/file/d/14PddLpEYGwIFouQmV97Rl_4snoJNScz3/view?usp=drive_link