**РУСЕНСКИ УНИВЕРСИТЕТ „АНГЕЛ КЪНЧЕВ“**

**КУРСОВА ЗАДАЧА**

ПО

ОПЕРАЦИОННИ СИСТЕМИ

Студент:

Фак. Номер:

Група:

Специалност:

Дата: Проверил:

**Съдържание**

[1. Задание 3](#_Toc437558975)

[2. Описание на използваните структури и алгоритми 3](#_Toc437558976)

[3. Код на програмата (листинг) 4](#_Toc437558977)

[4. Демо 11](#_Toc437558978)

# Задание

Да се разработи програмна система, симулираща работата на модула за **локализация на deadlock чрез редукция на графа за разпределяне на ресурсите** при следните изходни данни:

* ***Основни необходими структури:***
  + ProcRes [1..P, 1..R] разпределение на ресурсите по процеси:
    - Request заявка от процес P за ресурс от тип R
    - Hold получени ресурси от тип R
* ***Основни обработки:***
  + GetResource(Proc, Res) заявка за нов ресурс Res от процес Proc;
  + FreeResource(Proc, Res) освобождаване на ресурс от процес;
  + LocateDeadlock проверка за наличие на deadlock
  + Show показване на състоянието на основните структури

Разработеният проект да има подходящ потребителски интерфейс за обръщение към основните обработки. Обяснителната записка да съдържа заданието, описание на използваните структури данни и алгоритми и програмен листинг.

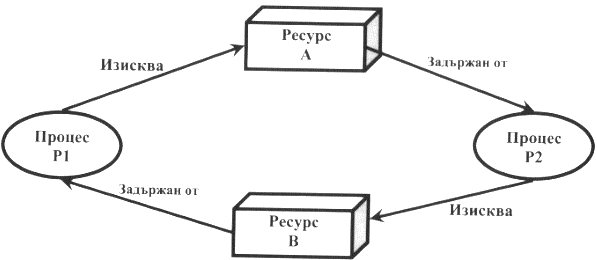
# Описание на използваните структури и алгоритми

За да възникне взаимна блокировка трябва да са налице три необходими условия:

1. Взаимно изключване – в даден момент от време само един процес може да използва ресурса;
2. Задържане и изчакване – процес може да задържи зададения ресурс, докато изчаква присвояване на другите ресурси;
3. Няма преразпределяне на ресурсите – не може ресурс насила да бъде отнет от процес, който го държи.

Взаимна блокировка може да съществува при горните три условия, но те не винаги са достатъчни. Изисква се и четвърто условие:

1. Кръгово (циклично) очакване – съществува затворена верига от процеси, така че всеки процес задържа поне един ресурс, необходим от следващия процес във веригата (Фигура 1).



Фигура 1 Кръгово (циклично) очакване

Ограничения на разработената програмата: може да има само по една инстанция на от всеки тип ресурс.

Класът Graph съдържа 4 член-данни:

* Int resources (броят на ресурсите)
* Int processes (броят на процесите)
* Int V (броят на върховете в графа = resource + processes)
* list<int> \*adj масив от списъци с наследниците на съответните върхове.

Член-функции:

* bool isCyclicUtil(int v, bool visited[], bool \*rs); - помощен метод , който се използва от метода isCyclic();
* bool isCyclic(); - функция за проверка на наличие на цикъл в граф
* Graph(int resources, int processes); - конструкторът на класа
* void addEdge(int v, int w); - метод за добавяне на наследник на съответния връх
* void Show(); - метод за отпечатване на информацията за процесите и ресурсите
* void GetResource(int process, int resource); - заявка за ресурс
* void FreeResource(int process, int resource); - освобождаване на ресурс

# Код на програмата (листинг)

#include <iostream>

#include <list>

#include <limits.h>

using namespace std;

class Graph

{

int resources;

int processes;

int V; // No. of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

bool isCyclicUtil(int v, bool visited[], bool \*rs); // used by isCyclic()

public:

Graph(int resources, int processes); // Constructor

void addEdge(int v, int w); // to add an edge to graph

bool isCyclic(); // returns true if there is a cycle in this graph

void Show(); // print graph

void GetResource(int process, int resource);

void FreeResource(int process, int resource);

};

Graph::Graph(int resources, int processes)

{

this->V = resources + processes;

this->resources = resources;

this->processes = processes;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

// This function is a variation of DFSUytil() in http://www.geeksforgeeks.org/archives/18212

bool Graph::isCyclicUtil(int v, bool visited[], bool \*recStack)

{

if(visited[v] == false)

{

// Mark the current node as visited and part of recursion stack

visited[v] = true;

recStack[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

{

if (!visited[\*i] && isCyclicUtil(\*i, visited, recStack))

return true;

else if (recStack[\*i])

return true;

}

}

recStack[v] = false; // remove the vertex from recursion stack

return false;

}

// Returns true if the graph contains a cycle, else false.

// This function is a variation of DFS() in http://www.geeksforgeeks.org/archives/18212

bool Graph::isCyclic()

{

// Mark all the vertices as not visited and not part of recursion

// stack

bool \*visited = new bool[V];

bool \*recStack = new bool[V];

for(int i = 0; i < V; i++)

{

visited[i] = false;

recStack[i] = false;

}

// Call the recursive helper function to detect cycle in different

// DFS trees

for(int i = 0; i < V; i++)

if (isCyclicUtil(i, visited, recStack))

return true;

return false;

}

void Graph::Show()

{

for(int i = this->processes; i < this->V; ++i)

{

list<int>::iterator j;

for(j = adj[i].begin(); j != adj[i].end(); ++j)

{

cout << "P" << \*j + 1 << " is holding R" << i - this->processes + 1 << endl;

}

}

for(int i = 0; i < this->processes; ++i)

{

list<int>::iterator j;

for(j = adj[i].begin(); j != adj[i].end(); ++j)

{

cout << "P" << i + 1 << " is requesting R" << \*j - this->processes + 1 << endl;

}

}

}

void Graph::GetResource(int process, int resource)

{

bool found = false;

list<int>::iterator j;

for(j = adj[process].begin(); j != adj[process].end(); ++j)

{

if(\*j == resource)

{

found = true;

break;

}

}

if(!adj[resource].empty()) // resource is held

{

bool found\_1 = false;

list<int>::iterator i;

for(i = adj[resource].begin(); i != adj[resource].end(); ++i)

{

if(\*i == process)

{

found\_1 = true;

break;

}

}

if(!found && !found\_1)

{

Graph::addEdge(process, resource);

}

cout << endl << "The resource is already allocated!" << endl;

}

else

{

if(found)

{

adj[process].erase(j);

}

Graph::addEdge(resource, process);

cout << endl << "P" << process + 1 << " got the resource!" << endl;

}

}

void Graph::FreeResource(int process, int resource)

{

bool found = false;

list<int>::iterator j;

for(j = adj[resource].begin(); j != adj[resource].end(); ++j)

{

if(\*j == process)

{

found = true;

break;

}

}

if(adj[process].empty() && found)

{

adj[resource].erase(j);

cout << endl << "Resource is freed!" << endl;

}

else

{

cout << endl << "Resource cannot be freed!" << endl;

}

}

int main()

{

int resNo, procNo;

int resources, processes;

cout << "Number of Resources: ";

cin >> resources;

cout << "Number of Processes: ";

cin >> processes;

Graph g(resources, processes);

bool\* resH = new bool[resources];

bool\* resR = new bool[resources];

int\* procH = new int[resources];

for(int i = 0; i < resources; ++i)

{

resH[i] = false;

resR[i] = false;

procH[i] = -1;

}

int b;

char ch;

for(int i = 0; i < processes; ++i)

{

cout << endl << "Is P" << i + 1 << " holding any resources (y/n)? ";

cin >> ch;

while(tolower(ch) == 'y')

{

cout << "P" << i + 1 << " is holding resource (1.." << resources << "): ";

cin >> b;

if(resH[b - 1] == false)

{

resH[b - 1] = true;

procH[b - 1] = i;

g.addEdge(processes + b - 1, i);

}

else

{

cout << endl << "R" << b << " is already held!" << endl;

}

cout << endl << "Is P" << i + 1 << " holding other resources (y/n)? ";

cin >> ch;

}

cout << endl << "Is P" << i + 1 << " requesting any resources (y/n)? ";

cin >> ch;

while(tolower(ch) == 'y')

{

cout << "P" << i + 1 << " is requesting resource (1.." << resources << "): ";

cin >> b;

if(procH[b - 1] != i && resR[b - 1] == false)

{

resR[b - 1] = true;

g.addEdge(i, processes + b - 1);

}

else

{

cout << endl << "R" << b << " is already held or requested by P" << i + 1 << "!" << endl;

}

cout << endl << "Is P" << i + 1 << " requesting other resources (y/n)? ";

cin >> ch;

}

}

int choice;

do

{

cout << endl;

cout << "[1] Get Resource" << endl;

cout << "[2] Free Resource" << endl;

cout << "[3] Locate Deadlock" << endl;

cout << "[4] Show" << endl;

cout << "[0] Exit" << endl;

cin >> choice;

switch(choice)

{

case 1:

cout << "Process (1.." << processes << "):"<< endl;

cin >> procNo;

cout << "Resource (1.." << resources << "):"<< endl;

cin >> resNo;

g.GetResource(procNo - 1, resNo + processes - 1);

break;

case 2:

cout << "Process (1.." << processes << "):"<< endl;

cin >> procNo;

cout << "Resource (1.." << resources << "):"<< endl;

cin >> resNo;

g.FreeResource(procNo - 1, resNo + processes - 1);

break;

case 3:

if(g.isCyclic())

{

cout << "Graph contains cycle - Deadlock!" << endl;

}

else

{

cout << "Graph doesn't contain cycle - No Deadlock!" << endl;

}

break;

case 4:

g.Show();

break;

}

} while(choice != 0);

return 0;

}

# Демо

