Simulation of Banker's algorithm to avoid deadlock.

Group: 2 ETC B3

Roll No: 52, 54.

**Introduction:**

Banker’s algorithm is a process scheduling algorithm which also detects and prevents the deadlock condition. The reason behind the name ‘banker’s algorithm’ is that it is mostly used in banking systems. Banker’s algorithm helps to identify whether a loan should be provided or not. Let’s consider the example. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan, then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money, then the bank can easily do it. Similarly, CPU allows the processes which can be processed using available resources.

When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

**Characteristics of banker’s algorithm:**

1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

**Disadvantages of banker’s algorithm:**

1. It requires a fixed number of processes, and no additional processes can be started in the system while executing the process.
2. The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks.
3. Each process must know and state their maximum resource requirement in advance for the system.
4. The number of resource requests can be granted in a finite time, but the time limit for allocating the resources is one year.

**Example:**

Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resource types: A has 10, B has 5 and the resource type C has 7 instances. In Allocation column, resources already allocated to process are mentioned. In column Max, maximum resources needed for the process are mentioned. In Available column, free resources before scheduling are mentioned.

Graphical user interface, text, application, email

Description automatically generated

Let’s calculate need of resources for different processes.

P1: (7,5 , 3) - (0, 1, 0) = 7,4,3

P2: (3,2,2) - (2,1,0) = 1,2,2

P3: (9,0,2) - (3,0,2) = 6,0,0

P4: (2,2,2) - (2,1,1) = 0,1,1

P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

|  |  |  |  |
| --- | --- | --- | --- |
| Need | A | B | C |
| P1 | 7 | 4 | 3 |
| P2 | 1 | 2 | 2 |
| P3 | 6 | 0 | 0 |
| P4 | 0 | 1 | 1 |
| P5 | 4 | 3 | 1 |

Now, Let’s apply banker’s algorithm.

**Step 1:** For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is false. So, we examine another process, P2.

**Step 2:** For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition true**.** So, resources can be allocated.

New available = available + Allocation => (3, 3, 2) + (2, 0, 0) => 5, 3, 2.

**Step 3:** For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is false. Similarly, we examine another process, P4.

**Step 4:** For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is true. New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

**Step 5:** For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is true. New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

**Step 6:** For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is true. New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

**Step 7:** For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true. New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7.

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like **P2, P4, P5, P1 and P3.**

goto step (2)

4) if Finish [i] = true for all i, then the system is in a safe state.

**Resource-Request algorithm:**

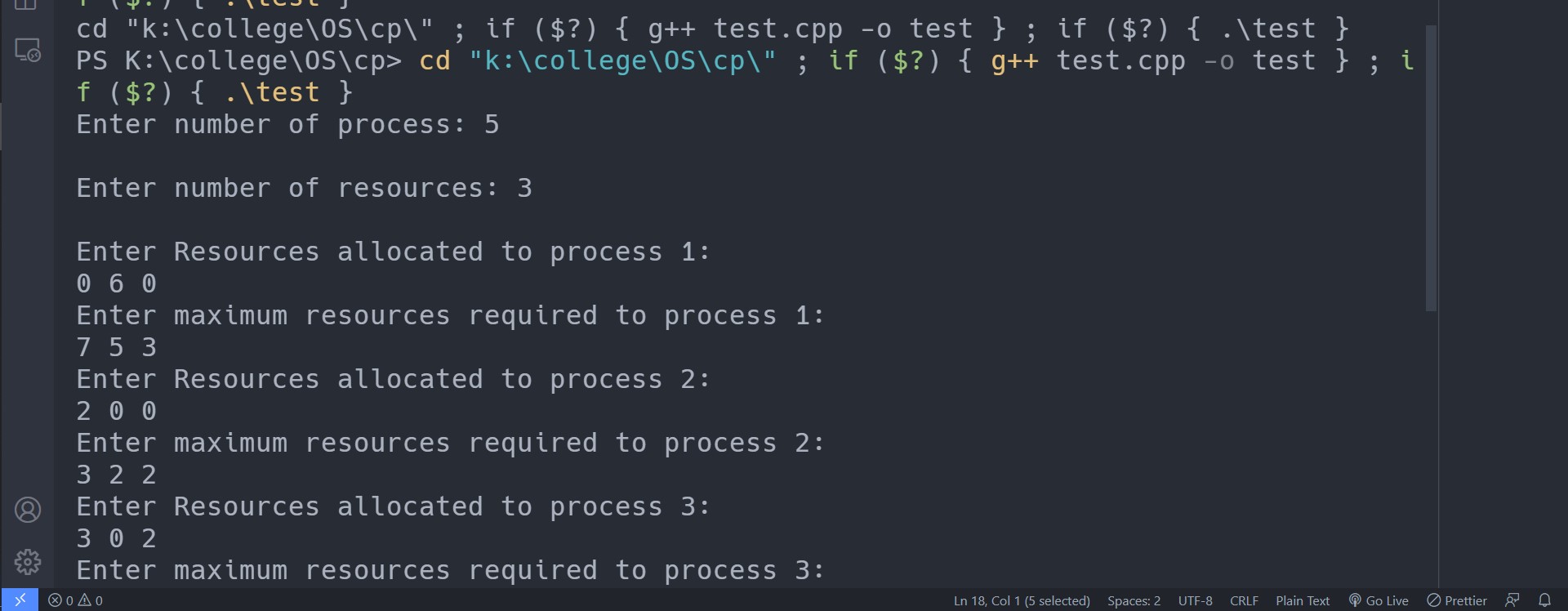
Let Request**i** be the request array for process Pi. Request**i** [j] = k means process Pi wants k instances of resource type R**j**. When a request for resources is made by process P**i**, the following actions are taken:

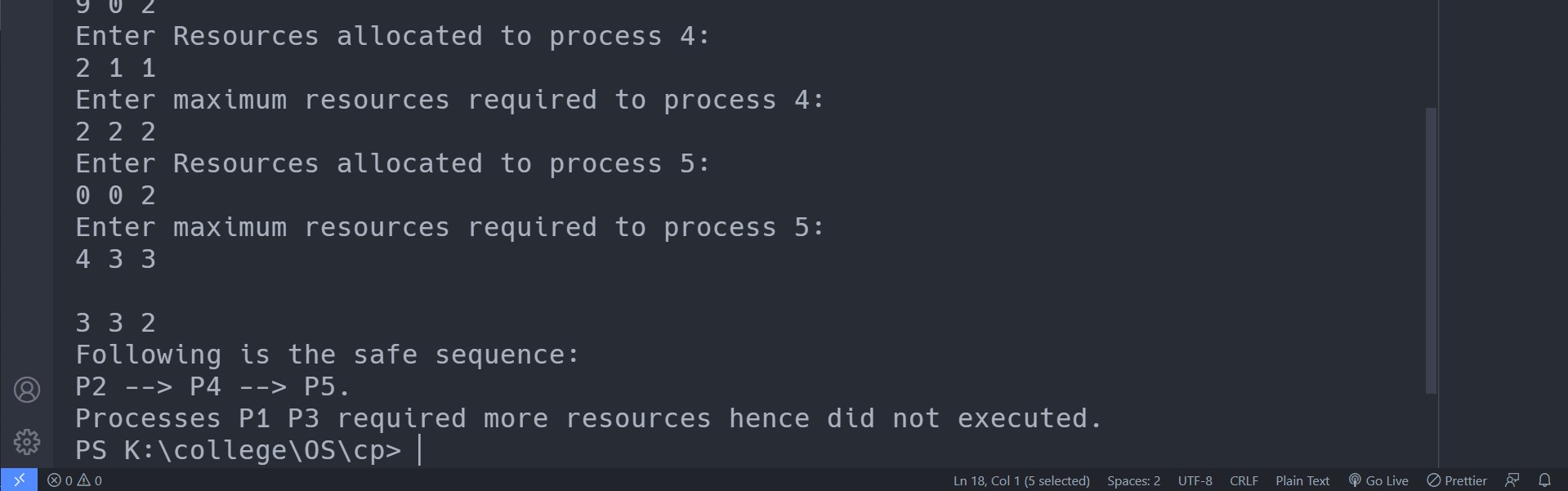
1. If need**i** <= Available

Goto step (3); otherwise, P**i** must wait, since the resources are not available.

1. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
   1. Available = Available –need**i**
   2. Allocation**i** = Allocation**i** + need**i**.

**Output:**





**Code:**

#include<bits/stdc++.h>

using namespace std;

#define pb push\_back

#define sz(*x*) (int)(x.size())

#define nl cout<<"\n"

#define FIO ios\_base::sync\_with\_stdio(false);cin.tie(NULL);cout.tie(NULL);

vector<int>sequence;

vector<int>unexecuted;

struct process{

  bool isFinished = false;

  vector<int> allocated;

  vector<int> maxReq;

  vector<int> need;

  process(vector<int> *allocated*, vector<int> *maxReq*){

    this->allocated = *allocated*;

    this->maxReq = *maxReq*;

    for(int z=0; z<sz(*allocated*); z++){

      int curNeed = max( 0, *maxReq*[z] - *allocated*[z]);

      need.pb(curNeed);

    }

  }

};

vector<process> inputProcess(){

  int totalPro, totalRes;

  cout<<"Enter number of process: "; cin>>totalPro; nl;

  cout<<"Enter number of resources: "; cin>>totalRes; nl;

  vector<process>processes;

  for(int i=0;i<totalPro;i++){

    vector<int> curAllocation(totalRes), curMaxReq(totalRes);

    cout<<"Enter Resources allocated to process "<<i+1<<": \n";

    for(int j=0;j<totalRes;j++){

      cin>>curAllocation[j];

    }

    cout<<"Enter maximum resources required to process "<<i+1<<": \n";

    for(int j=0;j<totalRes;j++){

      cin>>curMaxReq[j];

    }

*//Creating the process.*

    struct process curProcess(curAllocation, curMaxReq);

    processes.pb(curProcess);

  }

  return processes;

}

vector<int> inputAvailableRes(int *totalRes*){

  vector<int> available(*totalRes*);

  for(int i=0; i<*totalRes*; i++){

    cin>>available[i];

  }

  return available;

}

void printInput(vector<process> &*processes*,vector<int> &*available*){

  int totalPro = sz(*processes*);

  int totalRes = sz(*processes*[0].need);

  for(int i=0;i<totalPro;i++){

    cout<<"\nProcess no. "<<i+1<<"\n";

    for(int j=0;j<totalRes;j++) cout<<*processes*[i].allocated[j]<<" ";

    nl;

    for(int j=0;j<totalRes;j++) cout<<*processes*[i].maxReq[j]<<" ";

    nl;

    for(int j=0;j<totalRes;j++) cout<<*processes*[i].need[j]<<" ";

    nl;

  }

}

bool isSafe(vector<process> &*processes*, vector<int> &*available*, int *proNo*){

  int totalRes = sz(*processes*[0].need);

  for(int k=0; k<totalRes; k++){

*//Checking if we have enough available resources for execution.*

    if(*processes*[*proNo*].maxReq[k]> *available*[k]){

      return false;

    }

  }

*//We can execute this process.*

  return true;

}

void addAllocatedRes(vector<process> &*processes*,vector<int> &*available*, int *proNo*){

*// Adding allocated resources of the finished process*

*// to the available resources as they are free to use now.*

  int totalRes = sz(*processes*[0].need);

  for(int z=0;z<totalRes;z++){

*available*[z] += *processes*[*proNo*].allocated[z];

  }

}

void simulate(vector<process> &*processes*, vector<int> &*available*){

  int totalRes = sz(*processes*[0].need);

  int it=0;

  while(true){

    int thisIteration=0;

    for(int i=0;i<sz(*processes*);i++){

      if(*processes*[i].isFinished==false && isSafe(*processes*,*available*,i)){

        sequence.pb(i+1);

        addAllocatedRes(*processes*, *available*, i);

*// for(int k=0;k<totalRes;k++)cout<<available[k]<<" ";*

*// nl;*

*processes*[i].isFinished=true;

        thisIteration++;

      }

    }

    if(thisIteration==0){

      for(int i=0;i<sz(*processes*);i++){

        if(*processes*[i].isFinished==false)unexecuted.pb(i+1);

      }

      break;

    }

  }

}

void printSafeSeq(){

  cout<<"Following is the safe sequence:\n";

  int len = sz(sequence);

  for(int i=0;i<len;i++){

    cout<<"P"<<sequence[i];

    if(i!=len-1)cout<<" --> ";

    else cout<<".";

  }

  if(sz(unexecuted)){

    cout<<"\nProcesses";

    for(auto i: unexecuted)cout<<" P"<<i;

    cout<<" required more resources hence did not executed.";

  }

}

int main(){

#ifndef ONLINE\_JUDGE

  freopen("input.txt","r",stdin);

  freopen("output.txt","w",stdout);

#endif

FIO;

  vector<process>processes = inputProcess();

  vector<int>available = inputAvailableRes(sz(processes[0].need));

  simulate(processes, available);

  printSafeSeq();

}

**Lamport’s Bakery Algorithm**

**Critical Section:** A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

**Race condition:** A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

**Mutual exclusion:** The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

**Introduction:**

The **Bakery algorithm** is one of the simplest known solutions to the mutual exclusion problem for the general case of N process. Lamport’s bakery algorithm is an algorithm that restricts two or more processes from accessing a resource simultaneously.

This algorithm is known as the bakery algorithm as this type of scheduling is adopted in bakeries where token numbers are issued to set the order of customers. When a customer enters a bakery store, he gets a unique token number on its entry. The global counter displays the number of customers currently being served, and all other customers must wait at that time. Once the baker finishes serving the current customer, the next number is displayed. The customer with the next token is now being served. Similarly, in Lamport's bakery algorithm, processes are treated as customers. In this, each process waiting to enter its critical section gets a token number, and the process with the lowest number enters the critical section. If two processes have the same token number, the process with a lower process ID enters its critical section.

**Algorithm:**

do

{

1.     entering[i] :=  true; // show interest in critical section

   // get a token number

2.     number[i] := 1 + max(number[0],  number[1], ..., number[n - 1]);

    entering [i] :=  false;

 3.    for ( j :=  0 ;  j<n; j++)

    {

        while (entering [j])

        { ; } // do nothing

       while (number[j] !=  0 &&  (number[j], j) < (number[i], i)) // token comparison

        { ; } // do nothing

       }

// critical section

   4.       number[i] :=  0;  // Exit section

    } while(1);

**Example:**

Lets consider there are four processes P1, P2, P3, P4 who want to enter critical section.

|  |  |  |
| --- | --- | --- |
| Process | Token no / num[i] | Choosing [i] |
| P1 | 0 | F |
| P2 | 0 | F |
| P3 | 0 | F |
| P4 | 0 | F |

Lets say P3 wants to enter first. Then choosing[i] = True . Then the token number will be assigned i.e the max of all token no. + 1. So process 1 will get token 1. Then choosing[i] will be set to false.

|  |  |  |
| --- | --- | --- |
| Process | Token no / num[i] | Choosing [i] |
| P1 | 0 | F |
| P2 | 0 | F |
| P3 | 1 | F |
| P4 | 0 | F |

Then process will check if any other process has choosing set to True. If not then it will proceed. Then it will check if any other process has token other than 0 which is false in this case. So then it will proceed and enter into critical section

After this lets say that both P1, P2 request to enter the critical section. First the choosing[] will be set to false. Then both the processes will be assigned the same token no. as they entered at the same time. The token no will be 2. Then choosing will be set to false.

|  |  |  |
| --- | --- | --- |
| Process | Token no / num[i] | Choosing [i] |
| P1 | 2 | F |
| P2 | 2 | F |
| P3 | 1 | F |
| P4 | 0 | F |

Then process P1 will check if any other process has choosing set to True. If not then it will proceed. Then it will check if any other process has token other than 0 and if any other process has a smaller token no. For P2 this condition is false because process id of P1 < P2. So then it will check for P3. For P3 this condition will be true because token no of P3 < P1. So P3 will remain in the critical section and P1 will not be able to enter.

After P3 exits the critical section, its token no will be set to 0. Then process P1 will be able to enter the critical section

|  |  |  |
| --- | --- | --- |
| Process | Token no / num[i] | Choosing [i] |
| P1 | 2 | F |
| P2 | 2 | F |
| P3 | 0 | F |
| P4 | 0 | F |

**The Bakery algorithm meets all the requirements of the critical section problem.**

* **Mutual Exclusion:** we know that when no process is executing in its critical section, a process with the lowest number is allowed to enter its critical section. Suppose two processes have the same token number. In that case, the process with the lower process ID among these is selected as the process ID of each process is distinct, so at a particular time, there will be only one process executing in its critical section. Thus the requirement of mutual Exclusion is met.
* **Progress:** After selecting a token, a waiting process checks whether any other waiting process has higher priority to enter its critical section. If there is no such process, P will immediately enter its critical section. Thus meeting progress requirements.
* **Bounded Waiting:** As awaiting, the process will enter its critical section when no other process is in its critical section and
  + If its token number is the smallest among other waiting processes.
  + If token numbers are the same, it has the lowest process ID among other waiting processes.

Code:

// Importing the thread library

#include "pthread.h"

#include "stdio.h"

// Importing POSIX Operating System API library

#include "unistd.h"

#include "string.h"

// This is a memory barrier instruction.

// Causes compiler to enforce an ordering

// constraint on memory operations.

// This means that operations issued prior

// to the barrier will be performed

// before operations issued after the barrier.

#define MEMBAR \_\_sync\_synchronize()

#define THREAD\_COUNT 8

**volatile** **int** tickets[THREAD\_COUNT];

**volatile** **int** choosing[THREAD\_COUNT];

// VOLATILE used to prevent the compiler

// from applying any optimizations.

**volatile** **int** resource;

**void** lock(**int** **thread**)

{

    // Before getting the ticket number

    //"choosing" variable is set to be true

    choosing[**thread**] = 1;

    MEMBAR;

    // Memory barrier applied

**int** max\_ticket = 0;

    // Finding Maximum ticket value among current threads

**for** (**int** i = 0; i < THREAD\_COUNT; ++i) {

**int** ticket = tickets[i];

        max\_ticket = ticket > max\_ticket ? ticket : max\_ticket;

    }

    // Allotting a new ticket value as MAXIMUM + 1

    tickets[**thread**] = max\_ticket + 1;

    MEMBAR;

    choosing[**thread**] = 0;

    MEMBAR;

    // The ENTRY Section starts from here

**for** (**int** other = 0; other < THREAD\_COUNT; ++other) {

        // Applying the bakery algorithm conditions

**while** (choosing[other]) {

        }

        MEMBAR;

**while** (tickets[other] != 0 && (tickets[other]

                                           < tickets[**thread**]

                                       || (tickets[other]

                                               == tickets[**thread**]

                                           && other < **thread**))) {

        }

    }

}

// EXIT Section

**void** unlock(**int** **thread**)

{

    MEMBAR;

    tickets[**thread**] = 0;

}

// The CRITICAL Section

**void** use\_resource(**int** **thread**)

{

**if** (resource != 0) {

**printf**("Resource was acquired by %d, but is still in-use by %d!\n",

**thread**, resource);

    }

    resource = **thread**;

**printf**("%d using resource...\n", **thread**);

    MEMBAR;

    sleep(2);

    resource = 0;

}

// A simplified function to show the implementation

**void**\* thread\_body(**void**\* arg)

{

**long** **thread** = (**long**)arg;

    lock(**thread**);

    use\_resource(**thread**);

    unlock(**thread**);

**return** NULL;

}

**int** main(**int** argc, **char**\*\* argv)

{

**memset**((**void**\*)tickets, 0, **sizeof**(tickets));

**memset**((**void**\*)choosing, 0, **sizeof**(choosing));

    resource = 0;

    // Declaring the thread variables

    pthread\_t threads[THREAD\_COUNT];

**for** (**int** i = 0; i < THREAD\_COUNT; ++i) {

        // Creating a new thread with the function

        //"thread\_body" as its thread routine

        pthread\_create(&threads[i], NULL, &thread\_body, (**void**\*)((**long**)i));

    }

**for** (**int** i = 0; i < THREAD\_COUNT; ++i) {

        // Reaping the resources used by

        // all threads once their task is completed !

        pthread\_join(threads[i], NULL);

    }

**return** 0;

}

**Szymanski’s Algorithm:**

Overview:

This algorithm solves the problem of mutual exclusion. Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free. All processes requesting entry to the critical section at the same time, gather first in the waiting room. Last of them closes the entry door to the wating room and opens the exit door. Last of them closes the entry door and opens the exit door. The last process to leave the critical section, closes the exit door and reopens the entry door, so that the next batch of processes may enter.

Algorithm:

*# Entry protocol*

flag[self] ← 1 *# Standing outside waiting room*

**await** (all flag[1..N] ∈ {0, 1, 2}) *# Wait for open door*

flag[self] ← 3 *# Standing in doorway*

**if** any flag[1..N] = 1: *# Another process is waiting to enter*

flag[self] ← 2 *# Waiting for other processes to enter*

**await**(any flag[1..N] = 4) *# Wait for a process to enter and close the door*

flag[self] ← 4 *# The door is closed*

**await** (all flag[1..self-1] ∈ {0, 1}) *# Wait for everyone of lower ID to finish exit protocol*

*# Critical section*

*# ...*

*# Exit protocol*

**await** (all flag[self+1..N] ∈ {0, 1, 4}) *# Ensure everyone in the waiting room has*

*# realized that the door is supposed to be closed*

flag[self] ← 0 *# Leave. Reopen door if nobody is still in the waiting room*

Code:

*/\**

*PSEUDO CODE:*

*# Entry protocol*

*1) flag[self] ← 1                    # Standing outside waiting room*

*2) await(all flag[1..N] ∈ {0, 1, 2}) # Wait for open door*

*3) flag[self] ← 3                    # Standing in doorway*

*4) if any flag[1..N] = 1:            # Another process is waiting to enter*

*flag[self] ← 2                # Waiting for other processes to enter*

*await(any flag[1..N] = 4)     # Wait for a process to enter and close the door*

*5) flag[self] ← 4                    # The door is closed*

*6) await(all flag[1..self-1] ∈ {0, 1})   # Wait for everyone of lower ID to finish exit protocol*

*7) Critical section*

*# Exit protocol*

*8) await(all flag[self+1..N] ∈ {0, 1, 4}) # Ensure everyone in the waiting room has*

*# realized that the door is supposed to be closed*

*9) flag[self] ← 0 # Leave. Reopen door if nobody is still in the waiting room*

*Flag values:*

*0 - Not interested in entering CS*

*1 - Process wants to enter cs and it is waiting outside the waiting room*

*2 - Process has entered the waiting room but waiting for other process to enter*

*3 - Process has entered the waiting room and any other process is not allowed to enter.*

*4 - Waiting rooms door is closed.*

*\*/*

#include <iostream>

#include <thread>

using namespace std;

const int n = 8;

int flag[n];

void entryProtocol(int *i*)

{

    int index, index1,index3,index4;

    flag[i] = 1; *// 1)*

    index = 0;

    while (index <n) *// 2)*

        {

            if (flag[index] >=3)

                index = 0;

            else

                index = index + 1;

        }

    flag[i] = 3; *// 3)*

    index1=0;

    while(index1<n) *// 4)*

    {

        index3=0;

        if (flag[index1]==1)

        {

            flag[i]=2;

            while (flag[index3]!=4)

            {

                index3 = (index3+1) % n;

            }

        }

        else

            index1=index1+1;

    }

    flag[i] = 4; *// 5)*

    index4=0;

    while(index4<i) *//  6)*

    {

        if(flag[index4]<2)

            index4=index4+1;

        else

            index4=0;

    }

}

void exitProtocol(int *i*)

{

    int idx=i;

    while(idx<n) *// 8)*

    {

        if(flag[idx]<2 || flag[idx]==4)

            idx = idx+1;

        else

            idx=i;

    }

    flag[i]=0; *//  9)*

}

void run(int *i*)

{

    entryProtocol(i);

    cout << "thread " << i << " is running\n"; *// 7)*

    exitProtocol(i);

}

int main()

{

    for (int index = 0; index < n; index++) {

        flag[index] = 0;

    }

    thread ts[n];

    for (int i = 0; i < n; i++) {

        ts[i] = thread(run, i);

    }

    for (int i = 0; i < n; i++) {

        ts[i].join();

    }

    return 0;

}