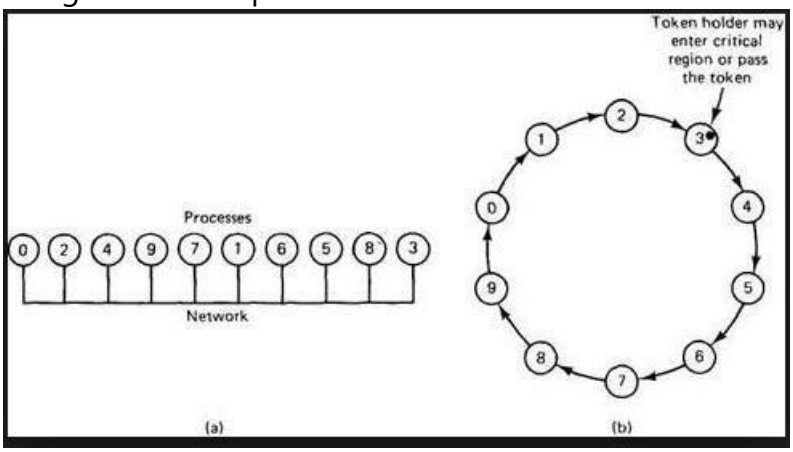
Aim: Implement Mutual Exclusion using Token Ring Algorithm

Theory:

Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes. A logical ring is constructed with these processes and each process is assigned a position in the ring. Each process knows who is next in line after itself. When the ring is initialized, process 0 is given a token. The token circulates around the ring. When a process acquires the token from its neighbour, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token to the next process in the ring. It is not allowed to enter the critical region again using the same token. If a process is handed the token by its neighbour and is not interested in entering a critical region, it just passes the token along to the next process.



Advantages:

- The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a CS.
- Since the token circulates among processes in a well-defined order, starvation cannot occur.

Disadvantages:

- Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter and leave one critical region.
- If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is not a constant. The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.
- The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbour tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can pass the token to the next member down the line.

Code:

```
#include <iostream>
#include <queue>
#include <vector>

using namespace std;
```

```
#define PR 6
int token_holder;

class Token
{
public:
    int id;             //Id of the site having token
    queue<int> token_q; //Token queue
    int ln[PR];         //Token Array for sequence no
    void init()         //Initializing token
    {
        id = 0;
        for (int i = 0; i < PR; i++)
        {
            ln[i] = 0;
        }
    }
} token;

//Description of each Site
class Site
{
public:
    int rn[PR];    //Site's Array for sequence no.
    bool exec;     //For checking whether site is executing
    bool isreq;    //For checking whether site is  requesting
    bool hastoken; //For checking whether site has token
    void init()    //Initializing sites
    {
        exec = 0;
        isreq = 0;
        hastoken = 0;
        for (int i = 0; i < PR; i++)
        {
            rn[i] = 0;
        }
    }
    void req(int pid, int seqno);
} site[PR];

//For a site to execute request of site pid with sequenceno seqno
void Site::req(int pid, int seqno)
{
    int i;
    rn[pid] = max(rn[pid], seqno);
    if (hastoken == 1)
    {
        if (exec == 0 && token.ln[pid] + 1 == rn[pid])
        {
            hastoken = 0;
            token_holder = pid;
```

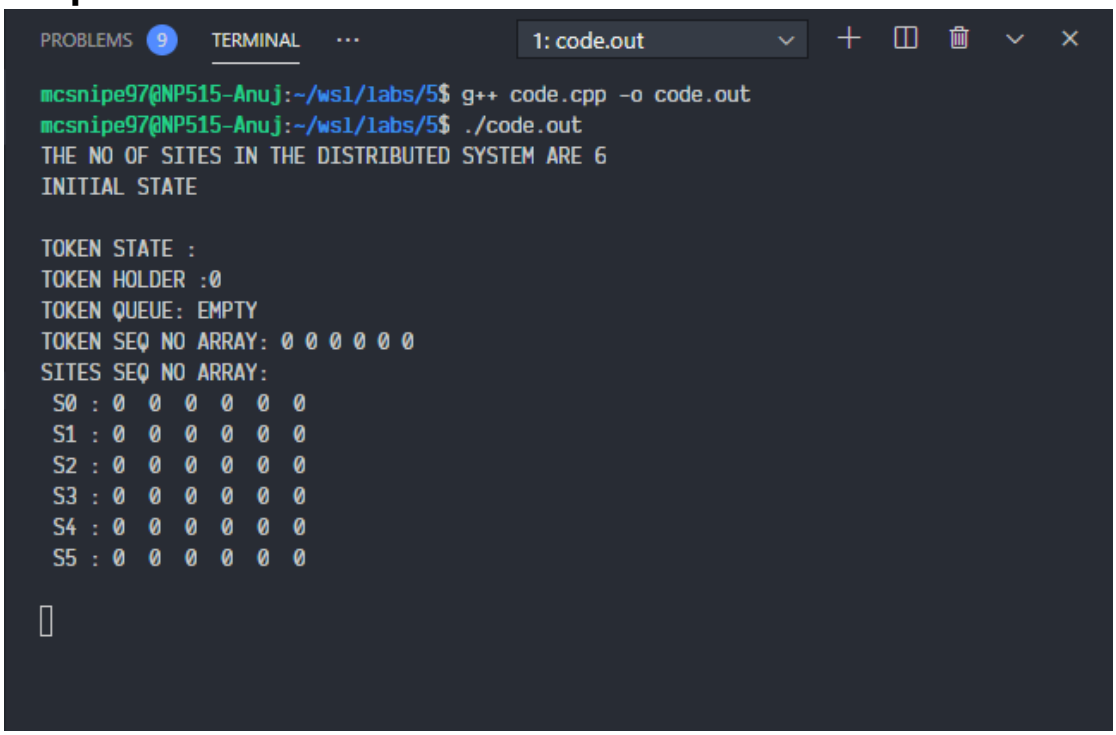
```
        }
        else if (token.ln[pid] + 1 == rn[pid])
        {
            token.token_q.push(pid);
        }
    }
}

//Initialize
void initialize()
{
    int i;
    token.init();
    for (i = 0; i < PR; i++)
    {
        site[i].init();
    }
}

//For a site with id pid to request for C.S.
void request(int pid)
{
    int i, seqno;
    seqno = ++site[pid].rn[pid];
    //Checking whether it has already requested
    if (site[pid].isreq == 1 || site[pid].exec == 1)
    {
        printf("SITE %d is already requesting \n", pid);
        return;
    }
    site[pid].isreq = 1;
    //Checking if it has the token
    if (token_holder == pid)
    {
        site[pid].isreq = 0;
        site[pid].exec = 1;
        printf("SITE  %d  already  has  the  token  and  it  enters  the  critical
section\n", pid);
        return;
    }

    //Sending Request
    if (token_holder != pid)
    {
        for (i = 0; i < PR; i++)
        {
            if (i != pid)
                site[i].req(pid, seqno);
        }
    }
    //Checking if it has got the token
    if (token_holder == pid)
```

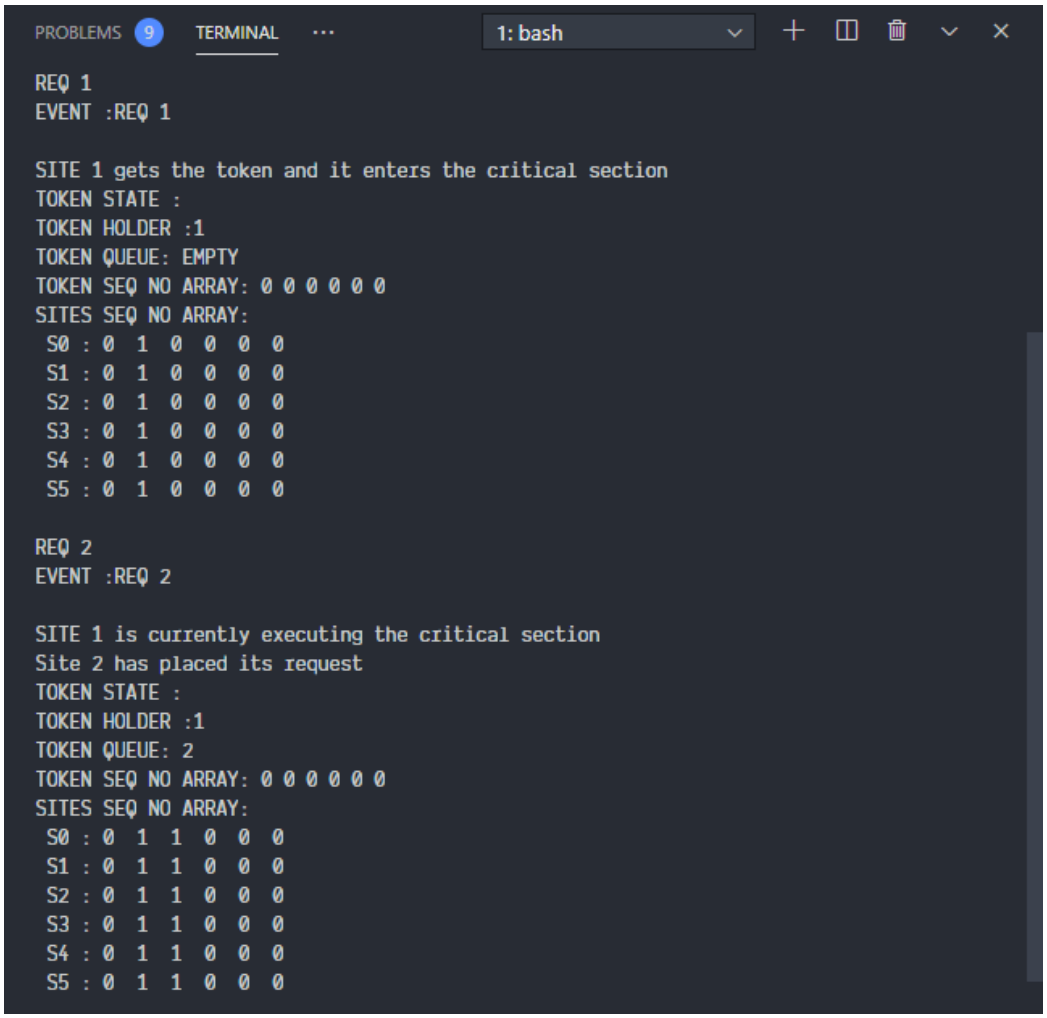
```
    {
        site[pid].hastoken = 1;
        site[pid].exec = 1;
        site[pid].isreq = 0;
        printf("SITE %d gets the token and it enters the critical section\n",
pid);
    }
    else
    {
        printf("SITE %d is currently executing the critical section \nSite %d has
placed its request\n", token_holder, pid);
    }
}

//For a site with id pid to request for C.S.
void release(int pid)
{

    if (site[pid].exec != 1)
    {
        printf("SITE %d is not currently executing the critical section \n", pid);
        return;
    }
    int i, siteid;
    token.ln[pid] = site[pid].rn[pid];
    site[pid].exec = 0;
    printf("SITE %d releases the critical section\n", pid);
    //Checking if deffred requests are there by checking token queue
    //And Passing the token if queue is non empty
    if (!token.token_q.empty())
    {
        siteid = token.token_q.front();
        token.token_q.pop();
        token.id = siteid;
        site[pid].hastoken = 0;
        token_holder = siteid;
        site[siteid].hastoken = 1;
        site[siteid].exec = 1;
        site[siteid].isreq = 0;
        printf("SITE %d gets the token and it enters the critical section\n",
siteid);
        return;
    }
    printf("SITE %d still has the token\n", pid);
}

//Printing the state of the system
void print()
{
    int i, j, k = 0;
    queue<int> temp;
    printf("TOKEN STATE :\n");
```

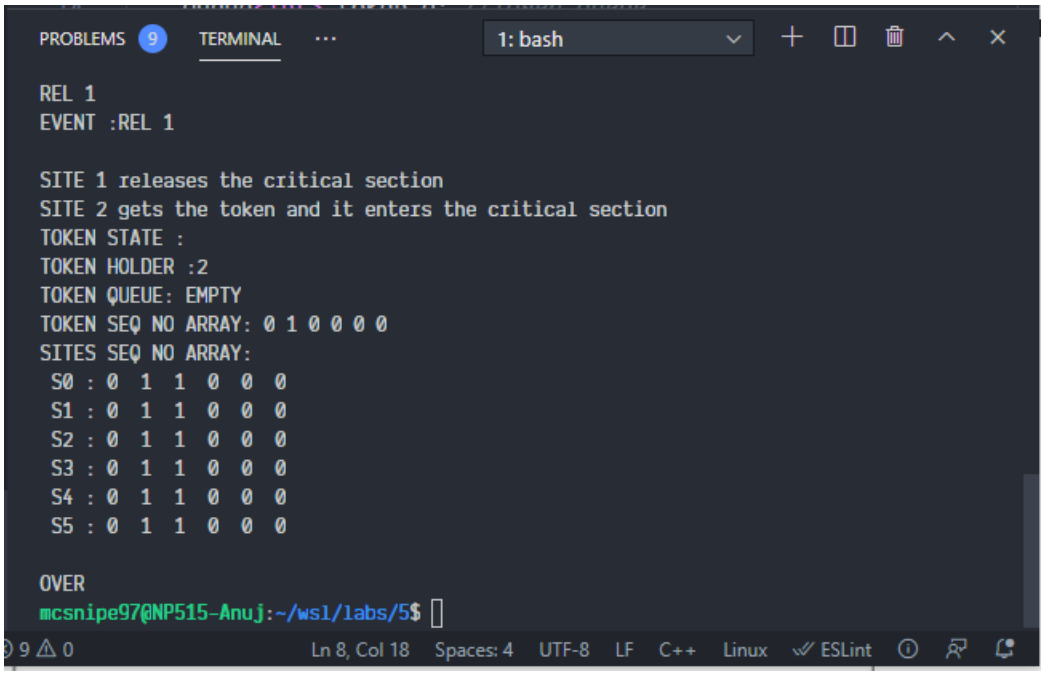
```
    printf("TOKEN HOLDER :%d\n", token_holder);
    printf("TOKEN QUEUE: ");
    if (token.token_q.empty())
    {
        printf("EMPTY");
        j = 0;
    }
    else
    {
        j = token.token_q.size();
    }
    while (k < j)
    {
        i = token.token_q.front();
        token.token_q.pop();
        token.token_q.push(i);
        printf("%d ", i);
        k++;
    }
    printf("\n");
    printf("TOKEN SEQ NO ARRAY: ");
    for (i = 0; i < PR; i++)
        printf("%d ", token.ln[i]);
    printf("\n");

    printf("SITES SEQ NO ARRAY: \n");
    for (i = 0; i < PR; i++)
    {
        printf(" S%d :", i);
        for (j = 0; j < PR; j++)
            printf(" %d ", site[i].rn[j]);
        printf("\n");
    }
}

int main()
{
    int i, j, time, pid;
    string str;
    initialize();
    token_holder = 0;
    site[0].hastoken = 1;

    time = 0;

    cout << "THE NO OF SITES IN THE DISTRIBUTED SYSTEM ARE " << PR << endl;
    cout << "INITIAL STATE\n"
         << endl;
    print();
    printf("\n");
    while (str != "OVER")
    {
```

```
        cin >> str;
        if (str == "REQ")
        {
            cin >> pid;
            cout << "EVENT :" << str << " " << pid << endl
                 << endl;
            request(pid);
            print();
            printf("\n");
        }
        else if (str == "REL")
        {
            cin >> pid;
            cout << "EVENT :" << str << " " << pid << endl
                 << endl;
            release(pid);
            print();
            printf("\n");
        }
    }
}
```

Output:







Conclusion:

1. We successfully implemented Token-Ring Mutual Exclusion.
2. This avoids Starvation
3. Lost Key is a major issue.