Name : Ajay Shitkar
Roll No : 58
Experiment No. 06
Implement Mutual Exclusion Algorithm
Date of Performance: 18/03/24
Date of Submission: 01/04/24

Aim: To implement mutual exclusion algorithm

Objective: Develop a program to implement mutual exclusion algorithm

Theory:

Mutual exclusion : makes sure that concurrent process access shared resources or data in a serialized way. If a process , say Pi , is executing in its critical section, then no other processes can be executing in their critical sections

Token ring is one among mutual exclusion algorithm in distributed systems that does not involve with a server or coordinator.

Working

A token is passed from one node to another node in a logical order. A node received token has a proper right to enter the critical section. If a node being holding token does not require to enter critical section or access a shared resource, it just simply passes the token to the next node.

Advantages

- Process suffer from no starvation. Before entering a critical section, a process waits at most for the duration that all other processes enter and exit the critical section
- No hand shake messages are required to get the token. Then less overhead

Disadvantages

- If a token is lost for some reasons, another token must be regenerated
- Detecting lost token is difficult since there is no upper bound in the time a token takes to rotate the ring (no particular timeout)
- The ring must be reconstructed when a process crashes

Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
int ns,ncs,timestamp,site;
cout<<"Enter number of sites :";
cin>>ns;
cout<<"Enter number of sites which want to enter critical section:";
cin>>ncs;
vector<int> ts(ns,0);
vector<int> request;
map <int,int> mp;
for(int i=0;i<ncs;i++)
{
cout<<"\nEnter timestamp:";
cin>>timestamp;
cout<<"Enter site number:";
cin>>site;
ts[site-1]=timestamp;
request.push_back(site);
mp[timestamp]=site;
```

```
}
cout<<"\nSites and Timestamp:\n";
for(int i=0;i<ts.size();i++)
{
cout<<i+1<<" "<<ts[i]<<endl;
}
for(int i=0;i<request.size();i++)
{
cout<<"\n Request from site:"<<request[i]<<endl;
for(int j=0;j<ts.size();j++)
{
if(request[i]!=(j+1))
{
if(ts[j]>ts[request[i]-1] || ts[j]==0)
cout<<j+1<<" Replied\n";
else
cout<<j+1<<" Deferred\n";
}
}
}
cout<<endl;
map<int,int>:: iterator it;
```

```
it=mp.begin();
int c=0;
for(it=mp.begin();it!=mp.end();it++)
{
cout<<"Site "<<it->second<<" entered Critical Section \n";
if(c==0)
cout<<"Similarly,\\n";
c++;
}
return 0;
}
```

Output:

```
Enter number of sites :5
Enter number of sites which want to enter critical section:3

Enter timestamp:2
Enter site number:2

Enter timestamp:3
Enter site number:3
```

Enter timestamp:1

Enter site number:4

Sites and Timestamp:

1 0

2 2

3 3

4 1

5 0

Request from site:2

1 Replied

3 Replied

4 Deferred

5 Replied

Request from site:3

1 Replied

2 Deferred

4 Deferred

5 Replied

Request from site:4

1 Replied

2 Replied

3 Replied

5 Replied

Site 4 entered Critical Section

Similarly,

Site 2 entered Critical Section

Site 3 entered Critical Section

Conclusion: The implementation of a mutual exclusion algorithm represents a fundamental aspect of concurrent and distributed systems design. Through the development and testing of the algorithm, this experiment aimed to ensure that only one process accesses a critical section at a time, thus preventing race conditions and maintaining data integrity. By employing techniques such as locks, semaphores, or distributed coordination mechanisms like Ricart-Agrawala or Lamport's algorithm, the experiment showcased different strategies for achieving mutual exclusion across various system architectures. This practical exploration not only deepened understanding of concurrency control but also provided hands-on experience in mitigating concurrency issues, essential for building reliable and scalable distributed applications.