International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 3– No.6, July 2012 – www.ijais.org

Efficient Mutual Exclusion Algorithm

Aasim khurshid
GPGC Mandian Abbotabad, Higher Education Department, KPK,Pakistan

Aamir Khan
COMSATS Institute of Information Technology, WahCantt, Pakistan

Farman Ullah
COMSATS Institute of Information Technology, WahCantt Pakistan

ABSTRACT

This paper presents an algorithm that can solve the problem in single processing, multiprocessing and distributed systems efficiently with minimal changes. For distributed systems we introduce message passing service while keeping rest of the mechanism same works faster than many other algorithms for distributed systems. Due to this multiple processes can execute in different critical sections concurrently. Performance of the algorithm is analyzed in terms of memory and time.

General Terms

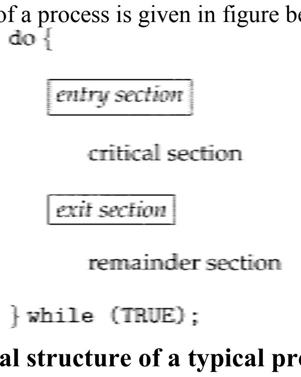
Operating Systems, Algorithms, Distributed systems.

Keywords

Mutual exclusion, Synchronization, Distributed systems, Operating systems, Algorithms, concurrency.

1. INTRODUCTION

A system having n numbers of processes say P0, P1, P2…… Pn be assumed. All processes are contending to access some shared data. A code segment called Critical Section is owned by each process, in which it can access and manipulate shared data e.g. changing common variables, update tables, writing to files etc. The significant attribute is that only one process be allowed to use its critical section at any point of time. The problem of critical section is to design a protocol that to cooperate these processes can use. The code segment that employs this request is called Entry section possibly followed by an exit section having code to exit from critical section may involve updating of some common data structure of the algorithm that can be used to allow other process to enter into their CS. The remaining code which is not associated to critical section of the process is organized in remainder Section.
The general structure of a process is given in figure below



```
do {
        entry section
            critical section
        exit section
            remainder section
} while (TRUE);
```

Figure 1: General structure of a typical process

Critical section problem solution must conform to these three basic requirements.

1.1 Mutual Exclusion:

Only one process at any instant of time can be allowed to execute in its critical section i.e. when one process say Pi is executing in its CS than other processes are to execute in their critical sections.

1.2 Progress:

When no process is executing in its critical section than only those processes which are not executing in their remainder section are allowed to compete in the decision to enter into their critical sections and this selection should not be delayed indefinitely.

1.3 Bounded waiting:

When a process made request to enter into its CS than there should be a limit on number of process that are allowed to enter into their critical section before this process's request is approved.
We assume that each process is executing at non zero speed. Yet no assumption regarding relative speed of the processes is possible.

2. RELATED WORK

2.1 Peterson's Solution:

Peterson Solution is one of the classic software based solution to critical problem for two processes. This may not work correctly on modern machine architecture because of the way they perform machine language instructions such as Test and Set. However, gist behind illustrating it here is that Peterson's solution is very handy to understand the complexities involved in designing software based solution to critical section problem.
Now for the solution consider the two processes share two variables:
int turn;
Boolean flag[1]
The integer variable turn specify whose turn it is to enter the critical section. The flag array is used to indicate if a process is ready to enter the critical section. flag[i]=true entails that process Pi is ready to enter into its critical section. Considering these the algorithm is as below

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
    Critical Section
        flag[i] = FALSE;
        Remainder Section
            } while (TRUE);
```

2.1.1 Proof of Correctness:

Now we prove that algorithm satisfies all the three requirements of CS problem solution namely mutual exclusion, progress and bounded waiting.
To prove mutual exclusion, note that a process Pi can enter into its CS if flag[i]=true and turn=0 , and process Pj can enter into its CS if flag[j]=true and turn=1, and both Pi and Pj can enter into their CS at same time when Pi=Pj=true and turn to be 0 and 1 at the same time, as one variable could not have

two values at the same time so there is no possibility that both Pi and Pj can enter into their CS simultaneously, which refers Mutual exclusion is preserved.

To prove progress and bounded waiting, note that Pi can be prevented only if it stuck in the while loop with the condition flag[j]=true and turn=j; this loop is the only one possible. If Pj is not ready to enter into its CS then flag[j]=false, then Pi enters its critical section. If Pj has its flag true and turn=i or turn=j, Pi will enter if turn=i and Pj will enter if turn=j, into their CSs. However once Pj exits critical section it will reset flag[j]=false, allowing Pi to enter into its CS. If Pj sets flag[j] to true than it must set turn=i, since Pi does not change the value of turn while executing in while loop, so Pi will enter into its critical section which shows progress after at most one entry of Pj which proves Bounded waiting.[6]

2.2 Synchronization Hardware:

Hardware instructions can be the choice to effectively solve the critical section problem which makes the task of programming easier as well as improves system efficiency. Uni-processor environment is the one in which disabling interrupts while a shared variable is being modified solve the CS problem. Disabling interrupts on multiprocessor systems can be time-consuming due to the need of message passing to all the processors which ultimately delays CS entry and decreases system efficiency. Therefore many machines come with special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, atomically. These special instructions can be used in a relatively simple manner to solve the CS problem. This instruction is executed atomically.

TestAndSet instruction is executed atomically and its definition is as below.

```
boolean TestAndSet (boolean &target)
{
boolean rv = target;
target = true;
return rv;
}
```

Shared data in this algorithm is

Boolean lock (initialize to false)

And structure for process should be like this as below.

```
do
 {
while (TestAndSet(lock)) ;
critical section
lock = false;
remainder section
}
```

This solution for critical section meets all the three requirements of critical section.

2.3 Semaphores:

Semaphores are used for the solution of problems of more complexity. A Semaphore S is an integer variable and two atomic operations:

- **Wait**
- **Signal**

can be used to access semaphore.

The conventional definition of wait and signal operations are:-

```
wait (S) {
while S≤ 0 do no-op;
S--;
}
signal (S) {
S++;
}
```

The value of S can be modified in the operations wait and signal must be executed inextricably.

Semaphores can be used to solve the CS problem for n processes.

Shared data

Semaphore mutex; (integer variable initially mutex=1)

2.3.1 Process Structure:

For this algorithm each process has the following structure

```
do {
wait(mutex);
critical section
signal (mutex);
remainder section
} while (1);
```

2.3.2 Implementation:

The wait operation of semaphore is described as

```
void wait (semaphore S) {
S.value--;
if (S.value < 0) {
add this process to Semaphore List;
block();
}//end of If Block
}//End of Wait Function
```

The signal operation of semaphore can now be described as:

```
void signal (semaphore S) {
S.value++;
if (S.value <= 0) {
remove a process P from S.L;
wakeup (P);
} // End of IF Block
} // End of Signal Function
```

The significant characteristic of the semaphore is that they execute atomically. The CS problem is to make sure that no two or more processes can execute wait and signal operations at the same time at the same semaphore. Existing solutions for single processing environment is to disable interrupts. By disabling interrupts it is sure that instructions from different processes are not interleaved and only the current process will be running until scheduler gets control back.

This solution can be implemented in multiprocessing environment as to disable interrupt on every processor otherwise instructions from different processes running on different processors may get interleaved. But this solution is not feasible as it is difficult to disable interrupts on different processors and doing this can diminish the performance of the system as whole. Therefore multiprocessing systems should implement some other mechanism to perform these operations atomically such as spin lock [5].

Problems with semaphores:

Semaphore implementation using waiting queue experiences sometime a problem of deadlock and starvation. For instance how it suffers from deadlock consider two semaphores X and Y and two processes Pi and Pj, now

They have instructions like.

Pi	**Pj**
wait(X)	**wait(Y)**
wait(Y)	**wait(X)**
.	.
.	.
.	.
Signal(X)	**Signal(Y)**
Signal(Y)	**Signal(X)**

Suppose that Pi executes wait(X) and then Pj executes wait(Y), when Pi wait(Y) it must wait Pj to execute signal(Y) and conversely when Pj executes wait(X) it must wait for Pi to execute signal(X). When such situation is reached that both processes are waiting for each other to complete which they can complete only when other completes first is the deadlock state.

Starvation is indefinite blocking of set of processes. When semaphores adds and removes processes in LIFO order than there of possibility of starvation of some of process and these process may not manage to get permission to get into their critical sections.

2.4 Bakery Algorithm:

Leslie Lamport a computer scientist proposed a computer algorithm named Lamport's bakery algorithm, which provides very simple solution to critical section problem. In computers multiple threads may try to access simultaneously the same resources. If two or more threads try to write into the same memory location at the same time than there is fair chance of integrity violation to occur, or before one thread finished writing into some memory location another thread reads that memory location. Lamport's bakery algorithm is a mutual exclusion algorithms proposed to eliminate the concurrent access of multiple threads to enter into their critical section at same instant of time in order to prevent data corruption. Bakery algorithm is use to solve critical section problem for n processes. Before entering into critical section each process receives a number in increasing order. Process with the smallest number enters the critical section. If two processes Pi and Pj receives the same number than process names will be used to serve the request, i.e. if i<j than Pi will be served first otherwise Pj.

The algorithm uses the following shared data:

```
boolean choosing[n];
int number[n];
```

number[n] is the integer array of n length that stores the identification number given by the algorithm when process wants to enter into critical section initialized to 0. Choosing[n] is the Boolean array of length n initialized to false.

Considering this shared data the algorithm is as below.

```
do {
1. choosing[i] = true;
2. number[i] = max(number[0], number[1], ..., number [n – 1])+1;
3. choosing[i] = false;
for (j = 0; j < n; j++) {
4. while (choosing[j]) ;
5. while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
}
Critical Section
6. number[i] = 0;
Remainder section
} while (1);
```

The algorithm satisfies all the three properties of critical section problem. Proof is as below.

2.4.1 Safety:

Safety or mutual exclusion is that only one process will enter into its critical section at one point of time. Process Pi in its start will turn choosing[i]=true, means it wants to enter into its critical section. It can enter into its critical section when 4 and 5 conditions get false. Which states that it has to wait until all other processes having choosing[j]=true and number[j] not equal to zero and this number[j] either less than number[i] or if this identification number equal than check process names. There is a chance that two processes Pi and Pj receives the same ticket number than it will check the process id which definitely could not be the same for two processes, which means no two processes can get into their critical section at one point of time. So safety is preserved.

2.4.2 Progress:

After execution in critical section process Pi sets its number [0] and choosing[i]=false to allow other processes to critical section. When no process is in its critical section and some processes wants to enter into its critical section than clearly a process having the smallest ticket number will be allowed to enter into its critical section which implies progress is preserved.

2.4.3 Bounded waiting:

Processes which want to enter into their critical sections .i.e. desire to access some shared data receives ticket number in increasing order. A bound exists which is it will be allowed to enter its critical section after at maximum the number of processes which made request before this process. The algorithm follows first come first serve order which preserves the bounded waiting requirement of the critical section problem**.**

3. Proposed Algorithm

The core algorithm receives requests from processes and gives them mutual exclusive access to some shared data. To achieve this job processes need to be organized in some identical fashion as they have to perform some tasks in common, for instance request CS in the start and while exiting notify the main algorithm by some mean that it made its way out of its CS. For this process organization is as below.

3.1 Process organization

There can be number of processes in the system. The general structure of the process is as

```
do {
start section
critical section
exit section
remainder section
} while(true)
```

In the start section process will have a code that makes request to enter into critical section and to do code which is required to enter into its critical section. In our case the start section of the code is just to make a request for its CS. No shared variables are needed to set in the start section.

In the critical section when the algorithm allows this process to execute in its CS. In this section the process may access and manipulate the shared data.

When a process makes its exit from critical section it has to reset a shared variable flag=true; which means other process may enter into its own CS now on. Queuing module is continuously watching this variable so that it can allow other process to make progress.

Process executing in remainder section is either done up with its critical section or it don't want to enter into its CS. A process executing in this section is not allowed to make request for its CS.

3.2 Algorithm Description

The algorithm is designed to solve the critical section problem for n processes. When a process enters the system it receives an identity number from identity generation module. When a process make request to enter into its critical section it will wait for 5 milliseconds to get response from the main module if it didn't get any it will change its state to waiting and will wait in the waiting queue. Through this we can allow the scheduler to take another process from the ready queue to use CPU time. In contrast if it gets response within this time (called GRACE period) it will make its entry to critical section. This grace period can be changed considering on the process flow and average context switching time overhead.

The core algorithm is continuously receiving processes who want to enter into their critical sections, storing them in the Queue. In parallel Queuing module removing a process from the top of the queue and allowing it access to enter into its critical section to use shared data.

When a process completes its execution in the critical section it reset a shared Boolean variable to False and when the value of this variable get false queuing module allows another process to enter into its critical section.

The general form of algorithm is like

Shared data:

Boolean flag (= false initially)
int ID[ ] (array of length n)

Algorithm Program

```
/********************* Thread1****************/
        Add-To_Waiting-Queue()
        {
        While(Request)
        {
        Queue.Add-this-ID;
        }//end of loop
        }// end of function
/********************* Thread2****************/
        Allow-To-CS()
        {
        While(Queue not Empty){
        Flag=false;
        Queue.allow(process on top);
        while(flag==false);
        }//end of loop
        }//end of function
/**************Process exit section****************/
Critical section;
        Flag=true; //exit section
        Remainder section;
```

3.3 Correctness Proof

Correctness proof for n processes solution is organized as Mutual exclusion, Progress and bounded waiting.

3.3.1 Mutual exclusion

Clearly algorithm takes a process from the queue and allows it to enter into its critical section. When a process is allowed to enter into its CS algorithm resets the shared variable falg=false. Which means processes should have to wait outside their critical section as a process is already executing in its CS. Through this only one process at a time is allowed to enter into its CS at any point of time. If time expires or process exits its critical section it updates a shared variable flag to True. Only than a new process is allowed to enter when the value of the flag gets false.

3.3.2 Progress

Algorithm continuously investigates the shared variable flag in Thread2 whenever it resets by any means the new process is allowed which makes progress. While flag=false this thread is in busy waiting in the same statement whenever the flag variable is rest to true in the process's exit section this condition gets false and the algorithm moves to the next iteration in the loop and if queue not empty it allows another process to its CS which illustrate progress.

3.3.3 Bounded waiting

Algorithm is fair it works on first come first serve basis so the bound is the process Pi is allowed to enter into its CS on its turn i.e. after the total number of processes which made request before it. So bounded waiting is preserved. When a new process wants to enter into its critical section it sends request and when a request is received it adds this request to the end of the queue. When all the other processes that made request before this process are done up with their critical section it gets permission to enter into its CS.

3.4 Block Diagram:

The block diagram of the system is presented in this section. The block diagram demonstrates the major components of the system and the flow of the system. It also shows how processes are coming and receiving identities as well as how they get permission when they make request for the critical section.

The block diagram of the complete process is as below.

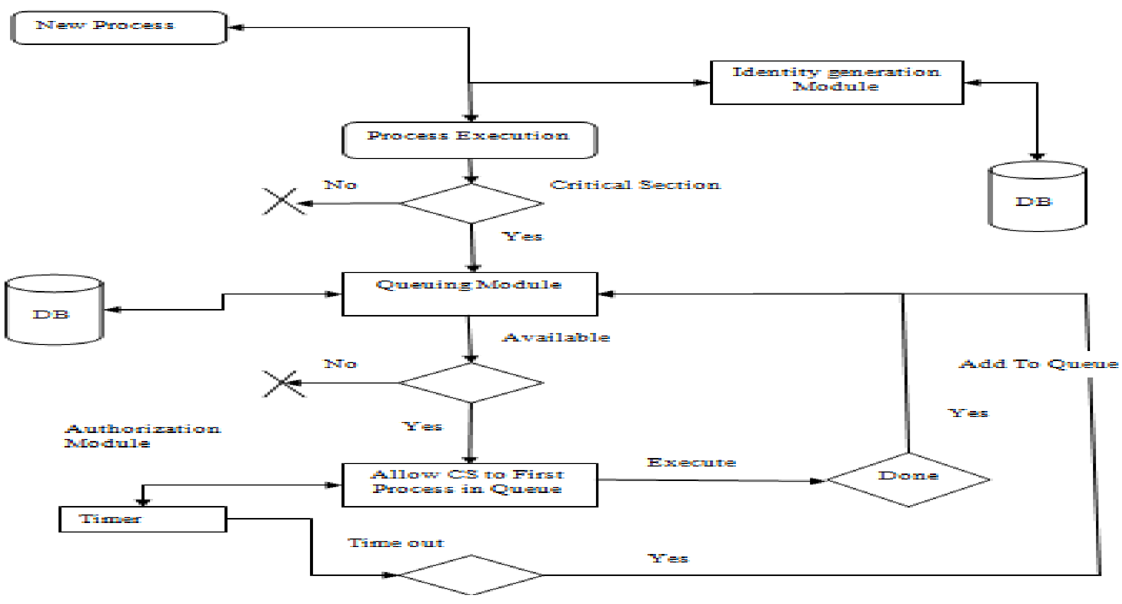


Figure 2: Block Diagram

4. RESULTS AND DISCUSSION

4.1 NOPR Vs Time

Average results for the comparison of Bakery and the proposed algorithm are simulated in figure 5.6. This shows the overall comparison of time taken for both the algorithms at different loads starting from number of processes to be 10 to 100000. Graph illustrates the comparison at many different points which evidence the improved efficiency of the proposed algorithm.

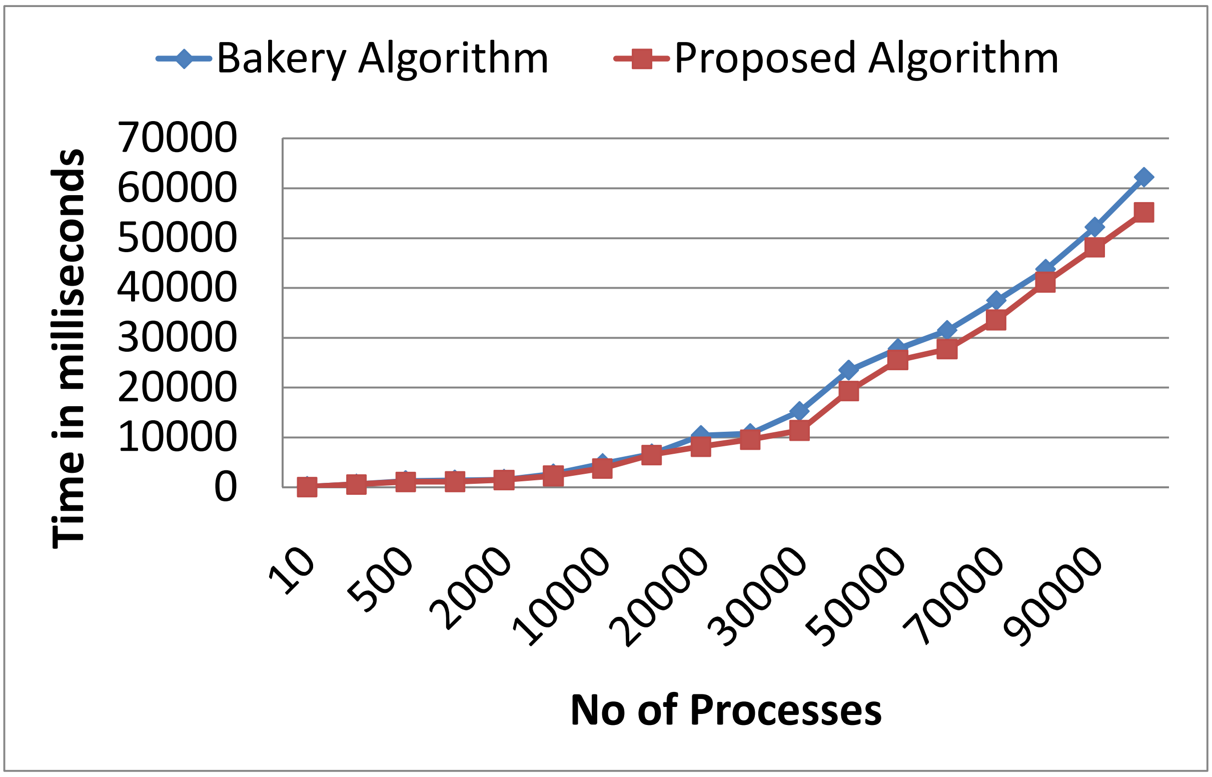


Figure 3: Comparison graph

4.2 Space Complexity

Space cannot be ignored while evaluating an algorithm; therefore it is considered important for the algorithm to utilize space well and does not burden the system by the algorithm's data structure.

The algorithm is using two shared variables, i.e. an array of integer type called ID[]and a Boolean variable flag. ID[] is used to allocate identification to all the processes in the system making request for critical sections, as all the processes needs identification to be unique so the size of this array is equal to the number of processes requesting CSs.

The variable flag is a single variable which is used for checking that if some process is in its critical section or not. If some process is not in critical section it turns to true and if some is in it is false, which forces mutual exclusive access. The proposed algorithm is using less space in this regard as many others for instance bakery algorithm is using a Boolean array for this purpose which increases the need of space for the algorithm's data structure specially when the number of process requests for CSs are extensively large in number. Proposed algorithm uses almost half space than bakery algorithm as whole.

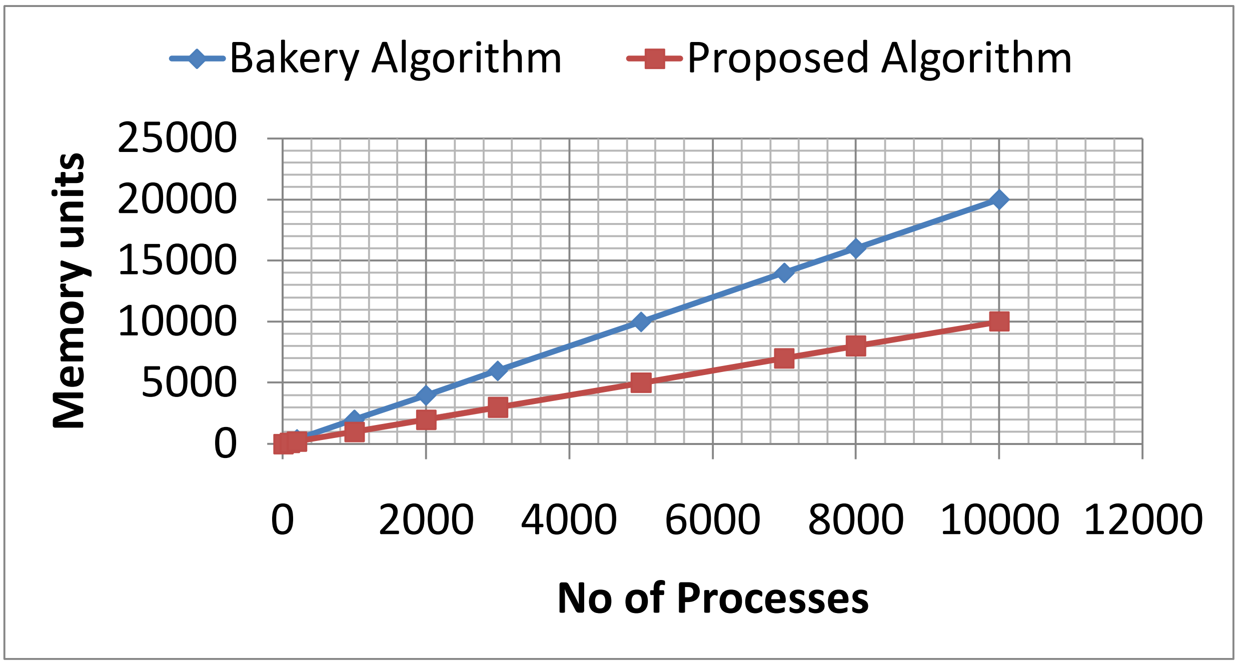


Figure 4 Memory Comparison

Space complexity for the algorithms is given below for number of processes to be N.
Bakery algorithm= 2N;
Proposed Algorithm=N+1;
Where
N is positive integer representing number of processes requesting CS at any point of time.

Table 5.1 show the number of memory units required for the number of processes making request for CSs mentioned in the first column. Second and third column shows the memory required for the bakery algorithm and proposed algorithm respectively.

Table 1: Memory Utilization

No of Process(es)	Bakery Algorithm (Memory Units)	Proposed Algorithm (Memory Units)
1	2	2
10	20	11
20	40	21
50	100	51
100	200	101
500	1000	501
1000	2000	1001
5000	10000	5001
10000	20000	10001
50000	100000	50001
100000	200000	100001

This clears the fact that proposed algorithm is taking almost the same memory units as the number of processes making requests for the critical section.

5. Distributed Systems Algorithm

In distributed systems, the proposed algorithm serves as centralized algorithm. The algorithm works in central and all the sites sends request to this central site through message passing. Assumptions are that for any two processes pi and pj, the messages sent from pi to pj are received in the same order in which they are sent. Furthermore, we assume that every message is eventually received. We assume that every process can send message directly to every other process. For generality we consider every site has one process.

5.1 Working

All the processes which want to enter into their CS sends request to the central site which they know by sending a message. Message contains the site ID. The central site receives that request and adds this to the end of the end of the queue. Whereas in parallel it removes a process ID from the queue and allows the process on top the entry authority to its critical section.

When site completes its execution in critical section it sends finish message to the central site which means it is no more in critical section. Than the algorithm removes another process ID from the queue and sends it ok message mean that it can enter into its CS.

The core algorithm works in similar for distributed systems while message passing system is introduced as in distributed systems sites can be at remote locations.

6. CONCLUSION

In this paper the performance of mutual exclusion algorithms is investigated. An algorithm is proposed that performs efficiently while satisfying all the necessary requirements of mutual exclusion algorithms.

The proposed algorithm is evaluated in consideration of time and memory. It is clear from the results that isolating the receiving request module from the authorization module which allows the processes to execute in their critical sections, and parallel running of these two modules increases the efficiency of the algorithm.
The busy waiting time variable that makes the algorithm efficient in terms of time. Keeping in focus on memory it is clear from the results that the algorithm limits the shared data structure used by the algorithm and taking almost the memory than its antecedent. The queuing implementation allows the algorithm to be used in single processing, multi-processing and distributed environment with minimal change.

7. REFERENCES

[1] E.W. Dijkstra; Solution of a Problem in Concurrent Programming Control, Communication ACM, vol. 8, no. 9, Sept. 1965

[2] Ricart G. and Agrawala A.: An Optimal Algorithm for Mutual Exclusion in Computer Networks. Communications of the ACM, vol. 24, no. 1,pp. 9-17, Jan. 1981

[3] D . Agrawal , A . El Abbadi , " An efficient solution to the distributed mutual exclusion problem " , in proc. 8th ACM Symposium on Principles of Distributed Computing , pp. 193-200 , 1989 .

Leslie Lamport; Time , Clocks and Ordering of Events in a Distributed System. Communications of the ACM, vol. 21, no. 1,pp. 558-565, July. 1978

[5] Md. Abdur Razzaque Choong Seon Hong, “Multi-Token Distributed Mutual Exclusion Algorithm,” in 22nd IEEE International Conference on Advanced Information Networking and Applications, 1550-445X/08, AINA, 2008, pp. 963–970.

[6] A S Silberschatz, P.B.Galvin, G Gangne, Operating System Concepts, USA, John Wiley & Sons, 2005, ISBN: 0-471-69466-5.

[7] Sandeep Lodha and Ajay Kshemkalyani, “A Fair Distributed Mutual Exclusion Algorithm,” IEEE Transactions On Parallel And Distributed Systems, Vol. 11, No. 6, June 2000,pp. 537-549

[8] Y.-I. Chang, “A Simulation Study on Distributed Mutual Exclusion,”. J. Parallel and Distributed Computing, vol. 33, pp. 107-121,1996

[9] O. Carvalho and G. Roucairol, ªOn Mutual Exclusion in Computer Networks, Technical Correspondence,º Comm. ACM, vol. 26, no. 2, pp. 146-147, Feb. 1983.

[10] Ricciuti, Mike (July 20, 2007). "Next version of Windows: Call it 7". CNET News. Available online at http://www.news.com/2100-1016_3-6197943.html.

[11] Nash, Mike (October 28, 2008). "Windows 7 Unveiled Today at PDC 2008". Windows Team Blog. Microsoft. Available online at http://windowsteamblog.com/blogs/windows7/archive/2008/10/28/windows-7-unveiled-today-at-pdc-2008.aspx. Retrieved November 11, 2008.

[12] Sadegh Firoozandeh and Abolfazl Toroghi Haghighat,” Reducing Coordinator Failures in Centralized Algorithm to Guarantee Mutual Exclusion Using a Backup Site” in Second IEEE International Conference on Future Networks, 2010,pp. 124-128

[13] J. M. Helary , N. Plouzeau , M. Raynal , " A distributed algorithm for mutual exclusion in an arbitrary network, volume 31 of Computer Journal,pp. 289-295, 1988.