ABSTRACT

The term Mutual Exclusion when applied to computer systems means a way to make sure that the shared variables are accessed by only one process at a time.

Our aim has been to study the fundamental concepts relating to the need for mutual exclusion and various techniques used to enforce it. The best way to understand these algorithms is to implement them. We have implemented solution to certain classical synchronization problems such as Producer Consumer and the Dining Philosopher problem. These problems are used to test any proposed techniques for enforcing mutual exclusion.

We have also made simulation programs in C++ to illustrate the working of Ricart and Agrawala algorithm which was the first optimal synchronization algorithm.We have implemented this algorithm for a totally connected network as well as a ring network.

ACKNOWLEDGEMENT

We feel honored in expressing our profound sense of gratitude and indebtedness to Mrs. Rajni Jindal, Assistant Professor & Project Head, Department of Computer Engineering, Delhi College of Engineering for providing us with the opportunity to work on such a prestigious project, under her expert guidance.

Her confidence in our capabilities and constant appreciation has been our strength throughout the project.

We would also like to thanks to Dr. D Roy Chaudhury, HOD, Department of Computer Engineering and Dr. Goldi Gabrani for their co-operation and help during the project.

Harmandeep Singh (2K1/COE/023)

Manan Chandra (2K1/COE/029)

Prasanjit Mandal (2K1/COE/040)

CERTIFICATE

This is to certify that project entitled

STUDY & IMPLEMENTATION OF MUTUAL EXCLUSION ALGORITHM

is a bonafide work of the following students of Delhi College of Engineering

Harmandeep Singh (2K1/COE/023)
Manan Chandra (2K1/COE/029)
Prasanjit Mandal (2K1/COE/040)

This project was completed under my direct supervision and Guidance and forms a part of their Bachelor of Engineering(B.E.) course curriculum.

They have completed their work with utmost sincerity, diligence and to my satisfaction.

I wish them best of luck for their future endeavours.

Mrs. RAJNI JINDAL

Assistant Professor & Project Head
Department of Computer Engineering
Delhi College of Engineering
Delhi University
Delhi-11042

INTRODUCTION

Sometimes processes have to interact sharing the common buffer area this interaction can lead to race conditions situations in which the exact timing determines the result. To avoid race conditions we need Mutual Exclusion- some way of making sure that if one process is using the shared variable or file the other process will be excluded from doing the same thing. That part of the program where the shared memory is accessed is called the Critical Section. Critical sections provide mutual exclusion.

Processes can communicate with each other using interprocess communication primitives. These primitives are used to ensure that no two processes are ever in their critical sections at the same time, that is to ensure mutual exclusion.

Various inter-process communication primitives are used among these are semaphores, monitors, event counters and message passing.

Monitors and semaphores are designed for solving mutual exclusion problem on one or more CPUs that all have access to a common memory. A number of classical problems have been solved using these and other primitives. The first test of any new proposed primitive is to see how well it solves the classical problems. These include the Producer-consumer, Dining Philosopher, Readers-Writers problems. Even with proper primitives care has to be taken to avoid errors and deadlocks.

For distributed systems, consisting of multiple CPUs each with its own private memory, connected by a local area network, message passing is used. Timestamps are attached to each message. They are used for ordering of events. The event with a lower timestamp occurs before an event with a higher timestamp.

INTERPROCESS COMMUNICATION

Processes frequently need to communicate with other processes. There is a need for communication between processes, preferably in a well-structured way not using interrupts. This situation calls for **IPC** or **Inter-Process Communication**.

There are basically three issues:

- How one process can pass information to another.
- Make sure two or more processes do not get into each other's way when engaging in critical activities(eg.- there are two processes, each try to grab the last 100K of memory)
- Proper sequencing must be done when dependencies are present: eg.- if process A produces data and process B prints it, B has to wait until A has produced some data before starting to print.

Race Conditions

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory or it may be a shared file. The location of shared memory does not change the nature of the communication or the problems that arise.

For example: **a print spooler**

When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are it prints them and then removes their names from the directory.

If a spooler directory has a large number of slots, 0..1..2..3…each one capable of holding a file name. Two shared variables are there:

out - which points to the next file to be printed

in - which points to the next free slot in the directory

Working:

At a certain instant, slots 0 to 3 are empty and slots 4 to 6 are full. More or less simultaneously, processes A & B decide they want to queue a file for printing. Process A reads IN and stores the value, 7, in a local variable called next_free_slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B. Process B also reads IN and also gets a 7, so it stores the name of its file in slot 7 and updates IN to be an 8. then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off. It looks at next_free_slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes (next_free_slot + 1), which is 8, and sets it to 8.

When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **Race Condition.**

CRITICAL SECTION

The question is how do we avoid race condition? The key to preventing trouble relating to shared memory & shared files is to find some way to prohibit more than one process from reading and writing the shared data at the same time. What we need is "**Mutual Exclusion**"-some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in any operating system, and a subject that we will examine.

The problem of avoiding "**race conditions**" can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that don not lead to race conditions. Sometimes a process may be accessing shared memory or files, or doing other critical things that can lead to races. The part of the program where the shared memory is accessed is called the "**Critical Section**". If we could arrange matters such that no two processes were ever in their critical sections at the same time, we could avoid race conditions.

Conditions to avoid **Race Conditions** are:

1. Ensure mutual exclusion between processes accessing the protected shared resource
2. Make no assumptions about relative speeds and priorities of contending processes
3. Guarantee that crashing or terminating of any process outside of its critical section does not affect the ability of other contending processes to access the shared resource
4. When more than one process wishes to enter the critical section, grant entrance to one of them in finite time.

The simplest way to ensure mutual exclusion is to outlaw concurrency. This approach is too drastic, as it also annihilates all performance improvements possible with concurrent execution of programs. What we really want to do is to temporarily grant a process that needs to complete a critical section exclusive access to a shared resource.

In many approaches to Mutual Exclusion, each process observes the following basic protocol:

(i)	Negotiation Protocol;	[winner proceeds]
(ii)	Critical Section;	[exclusive use of resources]
(iii)	Release Protocol;	[ownership relinquished]

Description:

- A process that wishes to enter a critical section first negotiates with all interested with all interested parties to make conflicting activity is in progress

- Concerned processes are aware of the imminent temporary unavailability of the resource.

- Once the consensus is reached, the winning process begins executing the critical section of code

- Upon completion, the process informs other contenders that the resource is available, and another round of negotiations may be started.

HARDWARE SOLUTIONS TO THE CRITICAL SECTION PROBLEM

1.DISABLING INTERRUPTS

The simplest way to achieve **Mutual Exclusion** is to have each process disable all interrupts just after entering its critical section, and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, and with interrupts turned off the CPU will not be switched to another process.
This process is unattractive because it is unwise to give user processes the power to turn off interrupts. Some common errors can be : if process does not re-enable it afterwards or the system is multiprocessor, with two or more CPUs, then disabling interrupts affects only the CPU that executed the disable instruction, so others access critical section and access shared memory.
It is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example-inconsistent state, race conditions could occur.

2.TEST AND SET LOCK (TSL)

Principle:
It reads the contents of the memory word into a register and then stores a nonzero value at that memory address. The operations of reading the word and storing into it are guaranteed to be indivisible-no other processor can access the memory word memory until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.
There is a shared variable, lock, to coordinate the access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read/write the shared memory. When it is done, the process sets lock back to 0 using an ordinary MOVE instruction.

```
enter_region:
        Tsl register, lock
        Cmp register, #0
        Jne enter_region
        Ret
leave_region:
        move lock, #0
        ret
```

Case 1: the first instruction copies the old value of lock to the register and then sets lock to 1. The old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again.

Case 2: sooner or later it will become 0 and the subroutine returns, with the lock set. When clearing the program just stores a 0 in lock

SOFTWARE SOLUIONS TO THE CRITICAL SECTION PROBLEM

TWO PROCESS SOLUTIONS:

ALGORITHM 1

```
Do {
        while (turn!=i);
                        CRITICAL SECTION
                turn = j;
                        REMAINDER SECTION
                }while(1);
```

Key Points:

- Our first approach is to let the processes share a **common integer variable turn initialized to 1 or 0**
- If **turn==i, then Pi executes in critical section**
- This **does not satisfy the progress requirement**, since it requires strict alternation of processes.
- For example: if turn==0 & P1 is ready to enter its critical section, P1 cannot do so, even though P0 may be in its remainder section.
- **Problem is that it does not retain information about state of each process**, remembers only which is to enter critical section
- This solution **ensures that only one process at a time can be in its critical section**

ALGORITHM 2

```
Do{     Flag[i]=true;
                While (flag[j]);
                        CRITICAL SECTION
                Flag[i]= false;
                        REMAINDER SECTION
   }while(1);
```

Key Points:

- In this algorithm process **Pi first sets flag[i] to True**, signaling that is ready to **enter its critical section**. Pi checks to verify that process Pj is not ready to enter critical section.If Pj were ready Pi would wait until flag[j] was false.
- At this point Pi would enter its critical section. **On exiting critical section Pi would set flag[i] to False.**
- **Each process updates its own flags** & strict **turn taking is removed**
- For example:

  An execution sequence

  T0: P0 sets flag[0] = TRUE

  T1: P1 sets flag[1] = TRUE

This algorithm is crucially dependent on the exact timing of the two processes. This situation could have been derived in an environment where there are several processors executing concurrently, or where an interrupt occurs immediately after step T0 is executed, and the CPU is switched from one process to another.

ALGORITHM 3

```
Do{
        Flag[I]=true;
        Turn=j;
            While(flag[j] && turn==j);
                CRITICAL SECTION
        Flag[I]=false;
                REMAINDER SECTION
    }while(1);
```

Key Points:

- The processes share 2 variables **'Boolean flag[2]' and 'int turn'.**
- **Initially flag[0]=flag[1]=false.Value of turn is immaterial (0** or **1).** To enter critical section process **Pi first sets flag[I] to be true** and **then sets turn to value j**. So that if other process wishes to enter critical section it can do so.
- If both process try to enter at same time, turn will be set to both i and j roughly the same time. Only one of the assignments will last.
- When P1 wishes to enter the critical section sets flag[i] but if P2 wishes to enter too then preempts P1 just before flag[i] is set, meanwhile P2 may set flag[j], start looping until flag[i] is false
- **So both loop forever waiting for other to go false**

MULTIPLE-PROCESS SOLUTIONS:

- Algorithm 3 solves the critical section problem for two processes, but for n processes we need a different algorithm. This algorithm is called "**Bakery Algorithm**" as it is based on the scheduling algorithm commonly used in bakeries.
- This algorithm was developed for a distributed environment

```
Do {
Choosing [i] = true;
Number[i]    = max(number [0], number[1],…, number[n-1]) +1;
Choosing[i]=false;

for (j=0;j<n;j++) {
while(choosing[j]);
while ( (number[j]!=0) && (number[j,j]<number[i,i]) );
}
        critical section;

number[i] = 0;

        remainder section

}while (1);
```

- On entering the store the customer receives a number, the customer with the lowest no is served next.
- The common data structure are
  - >Boolean choosing[n];
  - >int number[n];
- The bakery algorithm cannot guarantee that two processes do not receive the same number.
- In case of a tie, the process with the lowest name is served first.
- If Pi and Pj receive the same number and if I<j, then Pi is served first. Since process names are unique and totally ordered, our algorithm is completely deterministic.
- Initially, these data structures are initialized to false and 0, respectively.
  - (a,b) <(c,d) if a < c or if a==c & b<d
  - max(a0,…,an-1) is a number, k, such that k>=ai for I=0,….,n-1

SEMAPHORES

Semaphores is a variable type that ensure orderly access to shared regions of code. A semaphore mechanism basically consist of the two major primitive operations SIGNAL and WAIT(originally P and V) , which operate on semaphore variable s . The semaphore variable can assume integer values and, except possibly for initialisation, may be accessed and manipulated only by means of the SIGNAL and WAIT operations.
The two primitives take one argument each- the semaphore variable-may be defined as follows:

WAIT (s): Decrements the value of its argument semaphore, s, as soon as it would become nonnegative. Completion of the WAIT operation, once the decision is made to decrement its argument semaphore, must be indivisible.

```
while not (s>0) do {keeptesting};
s:= s-1;
```

SIGNAL (s): increments the value of its argument semaphore, s, as an indivisible operation.

```
s:= s+1;
```

Binary Semaphore:
A semaphore whose variable is allowed to take on only the values of 0 (BUSY) and 1 (FREE) is called a Binary Semaphore.

In this the logic of WAIT(s) is interpreted as waiting until semaphore variable s becomes equal to FREE, followed by its indivisible setting to BUSY before control is returned to mutual-exclusion protocol.

SIGNAL(s) sets the semaphore variable to FREE and thus represents the release phase of the mutual-exclusion sequence.
It is essential that signal(s) and wait(s) are implemented in an indivisible way. The normal way to implement Wait(s) and Signal(s) as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it and putting the process to sleep, if necessary. As all of these actions take only a few instructions no harm is done in disabling interrupts. If multiple CPUs are being used these semaphores should be protected by a “lock variable” with the TSL instruction used to make sure that only one CPU at a time examines the semaphore using TSL to prevent several CPUs from accessing the semaphore at the same time is quite different from

BUSY WAITING by the Producer-Consumer waiting for the other to empty or fill the buffer.

The semaphore operation will take only a few microseconds whereas the producer or consumer might take arbitrarily long. Following program contain code of three processes that share a resource accessed within a critical section. A binary semaphore MUTEX is used to protect the shared resource by enforcing its use in a mutually exclusive manner. Each process ensures the integrity of its critical section by opening it with a WAIT and closing it with a SIGNAL on the related semaphore. An arbitrary number of concurrent processes might join in.

```
Process p1;
 begin
        while true do
          begin
    wait(mutex);
                CRITICAL SECTION
                signal(mutex);
                other_p1
   end while
End p1;

Process p2;
 begin
        while true do
          begin
    wait(mutex);
                CRITICAL SECTION
                signal(mutex);
                other_p2
   end while
End p2;

Process p3;
 begin
        while true do
          begin
    wait(mutex);
                CRITICAL SECTION
                signal(mutex);
                other_p3
   end while
```

End p3;

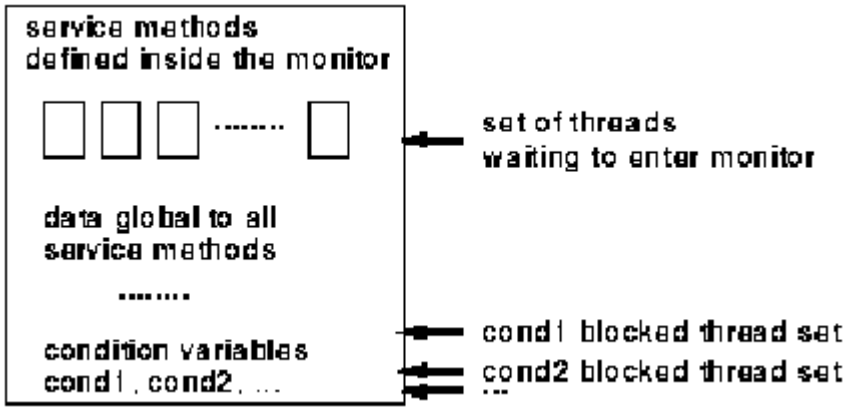
Table for above implementation:

Time	P1	P2	P3	Mutex	Processes
M1				1	
M2	Wait(mutex	Wait(mutex	Wait(mutex	0	-
M3	Critical	waiting	waiting	0	P1;P2,P3
M4	Signal(m)	Waiting	Waiting	1	-;P2,P3
M5	Other_P1	Critical	waiting	0	P2;P3
M6	Wait(mutex	Critical	waiting	0	P2;P3,P1
M7	waiting	Signal(m)	waiting	1	-;P3,P1
M8	Critical	Other_P2	waiting	0	P1,P3

MONITORS

A *monitor* is an object with some built-in mutual exclusion and thread synchronization capabilities. They are an integral part of the programming language so the compiler can generate the correct code to implement the monitor. Only one thread can be active at a time in the monitor where ``active" means executing a method of the monitor. Although monitors provide an easy way to achieve mutual exclusion , but that is not enough. We also need a way for the processes to block when they cannot proceed . IThe solution lies in the intoducton of control variables.

Monitors have *condition variables* on which a thread can *wait* if conditions are not right for it to continue executing in the monitor. Some other thread can then get in the monitor and perhaps change the state of the monitor. If conditions are now right that thread can *signal* a waiting thread moving the latter to the ready queue to get back into the monitor when it becomes free.



Condition variables are not counters. They donot accumulate signals for later use the way semaphores do . Thus if a conditional variable is signaled with no one waiting on it, the signal is lost. The Wait must come before the Signal. This rule makes the implementation much easier. In practice it is not a problem because it is easy to keep track of the state of each process with variables , if need be. A process that might otherwise do a signal can see that this operation is not necessary by looking at the variables.

It is up to the compiler to implement the mutual exclusion on monitor entries, but a common way is to use a binary semaphore. Because the compiler , not the programmer is arranging for the mutual exclusion , it is much less likely that something will go wrong . In any event the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical region into monitor procedures. No process will ever execute their critical regions at the same time.

Drawbacks of Monitors

By making the mutual exclusion of crtical regions automatic , monitors make parallel programming much ,less error-prone than semaphores. Still they do have some drawbacks. Monitors are programming language construct . The compliler must recognize them and arrange for mutual exclusion somehow. C.Pascal and most other languages do not have monitors , so it is unreasonable to expect their compilers to enforce any mutual exclusion rules.

Another problem with monitors is that they were designed to mutual exclsion problem on one or more CPU's that all have access to a common memory. By putting the semaphores in the shared memory and protecting them with TSl instruction we can avoid races. When we go to distributed system consisting of multiple CPUs each having it's own memory, these monitors become inapplicable .

The conclusion is that semaphores are not usable except in few programming languages.

Using Semaphore to implement Monitors

If the operating system provides semaphores as a basic feature, any compliler writer can easily implement monitors in his language.

First a small runtime collection of procedures for managing monitors is constructed and put in the library . They are shown in Fig.

```
Typedef int semaphore;

Semaphore mutex=1;                  // to control access to the monitor

Void enter_monitor(void)            // cod eto execute upon entry to monitor
{
 down(mutex);
}

void leave_normally(void)           // leave monitor without signaling
{
 up(mutex);                         // allow other processes to enter
}

void leave_with_signal(semaphore c)     //signal on c and leave monitor
{
 up();                                  // release one process waiting on c
}

void wait(semaphore c)                  // go to sleep
{
 up(mutex);                             // allow another process to enter
down();                                 // go to sleep on condition
}
```

Whenever generating code involving monitors, calls are made to the appropriate runtime procedure to perform the necessary function.

Associated with each monitor is a binary semaphore , mutex , initially 1 , to control entry to the monitor and additional semaphore,initially 0 per condition variable. When a process eneters the monitor , the compliler generates a call to the runtime procedure enter_monitor which does a down on the mutex associated with the monitor being entered . If the monitor is currently in use , the process will block.

It might seem logical that the code for exiting a monitor simply do an Up on mutex, this simple solution does not work . When the process has not signaled any other processes , then it can , just Up mutex and exit the monitor . This case is shown as leave normally.

This complication comes from the condition variables, Wait on a condition variable c is carried out as sequence of two semaphore operations . first comes an operation UP on mutex, to allow other processes to enter the monitor. Then comes a down on the condition variable.

Signal must always be done as the last operation before leaving the monitor . This rule is needed to be able to to combine signaling and exiting into library procedure leave_with_signal. All it does is an Up on the condition variable.

Java Monitors

Java uses the *synchronized* keyword to indicate that only one thread at a time can be executing in this or any other synchronized method of the object representing the monitor. A thread can call wait() to block and leave the monitor until a notify() or notifyAll() places the thread back in the ready queue to resume execution inside the monitor when scheduled. A thread that has been sent a signal is **not** guaranteed to be the next thread executing inside the monitor compared to one that is blocked on a call to one of the monitor's synchronized methods. Also it is **not** guaranteed that the thread that has been waiting the longest is the one woken up with a notify(); an arbitrary thead is chosen by the JVM. Finally when a notifyAll() is called to move all waiting threads back into the ready queue the first thread to get back into the monitor is **not** necessarily the one that has been waiting the longest.

Each Java monitor has a single nameless anonymous condition variable on which a thread can wait() or signal one waiting thread with notify() or signal all waiting threads with notifyAll(). This nameless condition variable corresponds to a lock on the object that must be obtained whenever a thread calls a synchronized method in the object. Only inside a synchronized method may wait() notify() and notifyAll() be called.

Methods that are static can also be synchronized. There is a lock associated with the class that must be obtained when a static synchronized method is called.

Usually all the publicly accessible methods the service or access methods are synchronized. But a Java monitor may be designed with some methods synchronized and some not. The non-synchronized methods may form the public access and call the synchronized methods which are private.

DISTRIBUTED MUTUAL EXCLUSION

Assumptions
1.The system consists of *n* processes; each process *Pi* resides at a different processor.
2. Each process has a critical section that requires mutual exclusion.

Basic Requirement
If *Pi* is executing in its critical section, then no other process *Pj* is executing in its critical section.

General Requirements

- Mutual exclusion must be enforced: only one process at a time is allowed in its critical section.
- A process that hales in its non-critical section must do so without interfering with other processes.
- It must not be possible for a process requiring access to a critical section to be delayed indefinitely: No deadlock or starvation.
- When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
- No assumptions are made about relative process speeds or number of processors.
- A process remains inside its critical section for a finite time only.

Centralized Approach

- This scheme requires three messages per critical section entry: Request, reply, release.
- One of the processes in the system is chosen to coordinate the entry to the critical section (control access to shared objects).
- A process that wants to enter its critical section sends a *request* message to the coordinator.
- The coordinator decides which process can enter the critical section next, and its sends that process a *reply* message.
- When the process receives a *reply* message from the coordinator, it enters its critical section.
- After exiting its critical section, the process sends a *release* message to the coordinator and proceeds with its execution.

Limits of the Centralized approach
If the coordinator process crashes, the system becomes useless.
The several messages towards the coordinator process can create a bottleneck.

Benefits of the Centralized approach
- Ensures mutual exclusion.
- Ensures ordering of the access requests.
- No starvation possible for any process.
- Use of only three messages.

General requirements of distributed algorithms

- All nodes have equal amount of information.
- Each node has only a partial picture of the total system and must make decisions based on this information.
- All nodes bear equal responsibility for the final decision.
- All nodes expend equal effort, on average, in effecting a final decision.
- Failure of a node, in general, does not result in a total system collapse.
- There exits no system wide common clock with which to regulate the time of events.

Time Stamping
Since in a distributed system the clocks are not synchronized, we have to be able to order events.

Implementation of happened-before relation
Associate a timestamp with each system event. Require that for every pair of events *A* and *B*, if *A is happened before B*, then the timestamp of *A* is less than the timestamp of *B*. Within *each* process *Pi* a *logical clock, LCi* is associated. The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process.

A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock (when a message is received, the receiving system sets its counter to one more than the maximum of its current value and the incoming time-stamp (counter)).

If the timestamps of two events *A* and *B* are the same, then the events are concurrent. We may use the process identity numbers to break ties and to create a total ordering. For this method to work, each message is sent from one process to all other processes.

Fully Distributed Approach

1 When process *Pi* wants to enter its critical section, it generates a new timestamp, *TS*, and sends the message *request* (*Pi, TS*) to all other processes in the system.

2 When process *Pj* receives a *request* message, it may reply immediately or it may defer sending a reply back

3 When process *Pi* receives a *reply* message from all other processes in the system, it can enter its critical section

4 After exiting its critical section, the process sends *reply* messages to all its deferred requests

The decision whether process *Pj* replies immediately to a *request*(*Pi, TS*) message or defers its reply is based on three factors:

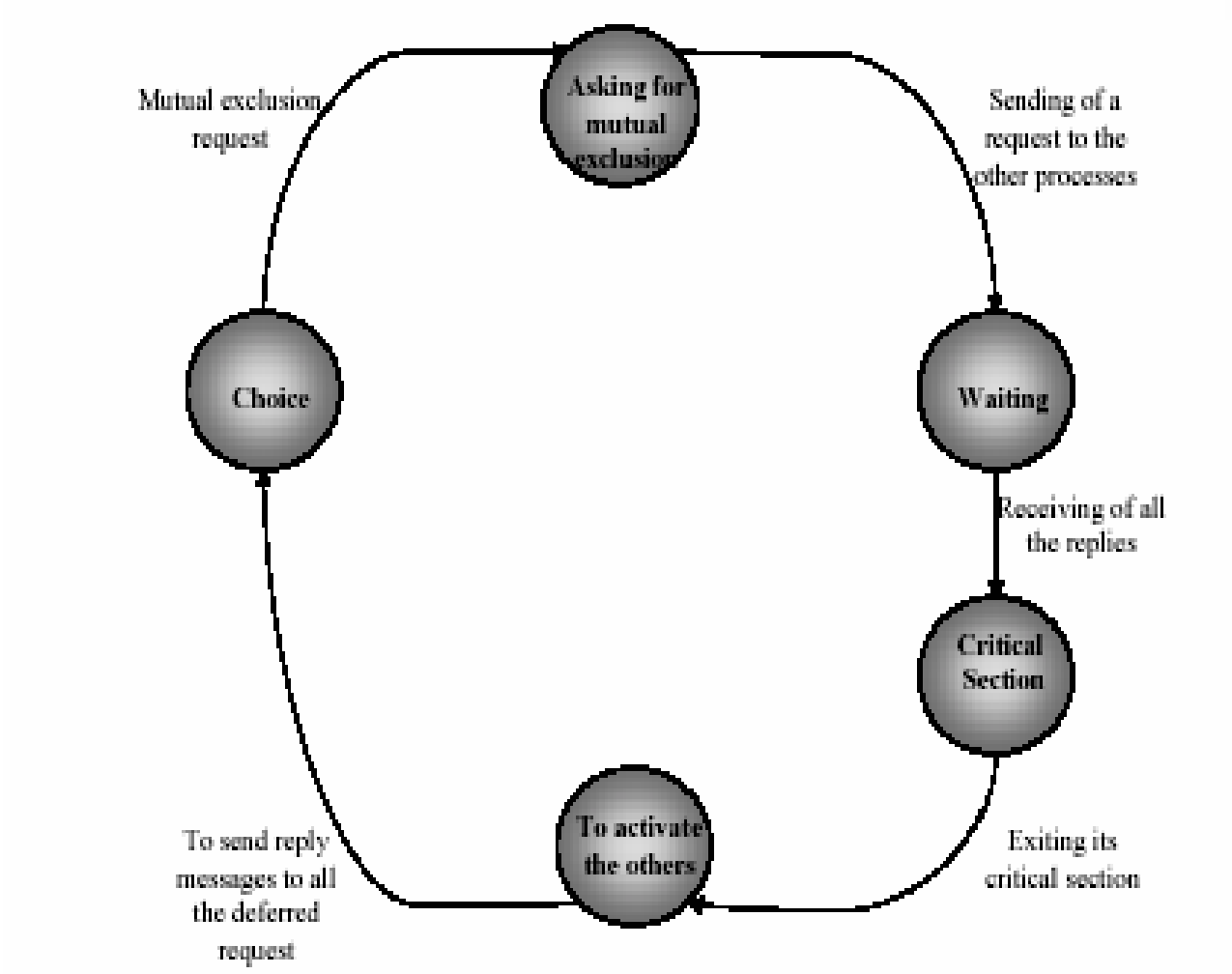
If *Pj* is in its critical section, then it defers its reply to *Pi*

If *Pj* does *not* want to enter its critical section, then it sends a *reply* immediately to *Pi*

If *Pj* wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp *TS*, If its own request timestamp is greater than *TS*, then it sends a *reply* immediately to *Pi* (*Pi* asked first) Otherwise, the reply is deferred

Advantages

- Freedom from Deadlock is ensured.
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering.
- The timestamp ordering ensures that processes are served in a first-come, first served order
- The number of messages per critical-section entry is 2 x (n – 1) , (n - 1) requests and (n- 1) replies.
- This is the number of required messages per critical section entry when processes act independently and concurrently



STATE TRANSITION DIAGRAM FOR THE FULLY DISTRIBUTED APPROACH

Undesirable Consequences of Fully Distributed Approach

- The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex
- If one of the processes fails, then the entire scheme collapses. This can be dealt with by continuously monitoring the state of all the processes in the system
- Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section.
- This protocol is therefore suited for small, stable sets of cooperating processes.

Token-Passing Approach

It exists a logical ring where each process has an assigned position and know who is next The token is an entity that at any time is held by one process. The process holding the token may enter its critical section without asking permission. Only this which has got the token can enter its critical section. When a process leaves its critical section, it passes the token to another process. When a process wants enter its critical section and it hasn't got the token, it sends request messages to all the other processes, waiting since the arrival of the token.

How to manage the use of the critical section

if not token_present then begin clock := clock + 1;
broadcast(Request, clock I);
wait(access, token);
token_present := True;
end
endif;
token_held := True:
<critical section>
token(i) := clock;
token_held := False;
for j := i + 1 to n, 1 to i - 1 do
if (request(j) > token(J)) [Symbol]^token_present
then begin
token_present := False;
send(access, token(j))
end

endif;

How to do when a process receives a request

when received (Request, k, j) do
request(j) := max(request(j), k);
if token_present[Symbol] ^not token_held then
token(i) := clock;
token_held := False;
for j := i + 1 to n, 1 to i - 1 do
if (request(j) > token(J)) [Symbol]^token_present
then begin
token_present := False;
send(access, token(j))
end
endif;
endif
enddo;

Notation :

send(j, access, token) send message of type access, with token, by process j
broadcast(request, clock, i) send message from process i of type request, with timestamp clock, to all other processes
received(request, t, j) receive message from process j of type request,with timestamp t

Troubles

- Difficult detection/regeneration of lost token
- Process crash
  - Difficult detection
  - Easy recover: if required acknowledgement from process when the token is received, dead processes are detected by neighbor which can skip it over (need to know the entire ring configuration).

Benefits

- Mutual exclusion is achieved, there is no starvation.
- At worst a process have to wait for every other process to enter and leave one critical region

COMPARISON

Algorithm	Messages per entry/exit	Delay before entry (in msg times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n-1)	2 (n-1)	Crash of any process
Token ring	1 to ∞	0 to (n-1)	Lost token, process crash

NOTE: None of the approaches is robust to system crashes

The Ricart and Agrawala algorithm

This algorithm creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. The algorithm sends only 2*(N - 1) messages, where N is the number of nodes in the network per critical section invocation. This number of messages is at a minimum if parallel, distributed, symmetric control is used; hence, the algorithm is optimal in this respect. The time needed to achieve mutual exclusion is also minimal under some general assumptions.
As in s "bakery algorithm," unbounded sequence numbers are used to provide first-come first-served priority into the critical section. It is shown that the number can be contained in a fixed amount of memory by storing it as the residue of a modulus. The number of messages required to implement the exclusion can be reduced by using sequential node-by-node processing, by using broadcast message techniques, or by sending information through timing channels. The "readers and writers" problem is solved by a simple modification of the algorithm and the modifications necessary to make the algorithm robust are described.

Key Words and Phrases: concurrent programming, critical section, distributed algorithm, mutual exclusion, network, synchronization.

1. Introduction

An algorithm is proposed that creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. It is assumed that there is an error-free underlying communications network in which transit times may vary and messages may not be delivered in the order sent. Nodes are assumed to operate correctly; the consequences of node failure are discussed later. The algorithm is symmetrical, exhibits fully distributed control, and is insensitive to the relative speeds of nodes and communication links.
The algorithm uses only 2* (N - 1) messages between nodes, where N is the number of nodes and is optimal in the sense that a symmetrical, distributed algorithm cannot use fewer messages if requests are processed by each node concurrently. In addition, the time required to obtain the mutual exclusion is minimal if it is assumed that the nodes do not have access to timing-derived information and that they act symmetrically.
While many writers have considered implementation of mutual exclusion [2,3,4,5,6,7,8,9], the only earlier algorithm for mutual exclusion in a computer network was proposed by Lamport [10,11]. It requires approximately 3* (N - 1) messages to be exchanged per critical section invocation. The algorithm presented here requires fewer messages (2* (N - 1)).

2. Algorithm

2.1 Description

A node enters its critical section after all other nodes have been notified of the request and have sent a reply granting their permission. A node making an attempt to invoke mutual exclusion sends a REQUEST message to all other nodes. Upon receipt of the REQUEST message, the other node either sends a REPLY immediately or defers a response until after it leaves its own critical section.

The algorithm is based on the fact that a node receiving a REQUEST message can immediately determine whether the requesting node or itself should be allowed to enter its critical section first. The node origin- nating the REQUEST message is never told the result of the comparison.

A REPLY message is returned immediately if the originator of the REQUEST message has priority; otherwise, the REPLY is delayed.

The priority order decision is made by comparing a sequence number present in each REQUEST message. If the sequence numbers are equal, the node numbers are compared to determine which will enter first.

2.2 Specification

The network consists of N nodes. Each node executes an identical algorithm but refers to its own unique node number as ME. ME is a pun on "mutual exclusion."

Each node has three processes to implement the mutual exclusion:

(1)One is awakened when mutual exclusion is invoked on behalf of this node.

(2)Another receives and processes REQUEST messages.

(3)The last receives and processes REPLY messages.

The three processes run asynchronously but operate on a set of common variables. A semaphore is used to serialize access to the common variables when necessary.

If a node can generate multiple internal requests for mutual exclusion, it must have a method for serializing those requests. The algorithm is expressed below in an Algol-like language.

SHARED *DATABASE*
CONSTANT
me, ! This node's unique number

N; ! The number of nodes in the network

INTEGER
Our_SequenceNumber,

! The sequence number chosen by a request
! originating at this node

HighestSequenceNumber initial (0),
! The highest sequence number seen in any
! REQUEST message sent or received

Outstanding_Reply_Count;
! The number of REPLY messages still
! expected

BOOLEAN
Requesting Critical_Section initial (FALSE),
! TRUE when this node is requesting access
! to its critical section

Reply_Deferred [I:N] initial (FALSE);
! Reply_Deferred [j] is TRUE when this node
! is deferring a REPLY toj's REQUEST message

BINARY SEMAPHORE
Shared vars initial (1);
! Interlock access to the above shared
! variables when necessary

PROCESS WHICH INVOKES MUTUAL EXCLUSION FOR THIS NODE
Comment Request Entry to our Critical Section;
P (Shared_vats)
Comment Choose a sequence number;
RequestingCritical_Section := TRUE;
Our_Sequence_Number := Highest_Sequence_Number + l;
V (Shared_vars);
Outstanding_ReplyCount := N - l;
FORj := I STEP l UNTIL N DO IFj # me THEN
Send_Message(REQUEST(Our_Sequence_Number, me),j);
Comment sent a REQUEST message containing our sequence number
and our node number to all other nodes;
Comment Now wait for a REPLY from each of the other nodes;
WAITFOR (Outstanding_Reply_Count = 0);
Comment Critical Section Processing can be performed at this point;
Comment Release the Critical Section;
RequestingCritical_Section := FALSE;
FOR j := l STEP 1 UNTIL N DO
IF Reply_Deferred[j] THEN
BEGIN
Reply_Deferred[j] := FALSE;
Send_Message (REPLY, j);
Comment send a REPLY to node j;

END;

PROCESS WHICH RECEIVES REQUEST (k, j) MESSAGES
Comment k is the sequence number begin requested,
j is the node number making the request;

BOOLEAN Defer it ;
! TRUE when we cannot reply immediately
Highest_Sequence_Number :~
Maximum (Highest_Sequence_Number, k);
P (Shared_vars);
Defer it :=
Requesting_Critical_Section
AND ((k > Our_sequence_Number)
OR (k = Our_Sequence_Number *ANDj* > me));
V (Shared_vars);
Comment Defer_it will be TRUE if we have priority over
node j's request;
IF Defer it THEN Reply_Deferred[j] := TRUE ELSE
Send_Message (REPLY, j);

PROCESS WHICH RECEIVES REPLY MESSAGES
Outstanding_Reply_Count := Outstanding_Reply_Count - 1;

Condition and action entries	Rule number				
	1	2	3	4	5
Receiving node is also requesting the resource	*N*	*Y*	*Y*	*Y*	*Y*
Received message's sequence number compared to ours	–	<	>	=	=
Received message's node number compared to ours	–	–	–	<	>
Send REPLY back	*X*	*X*		*X*	
Defer the REQUEST			*X*		*X*

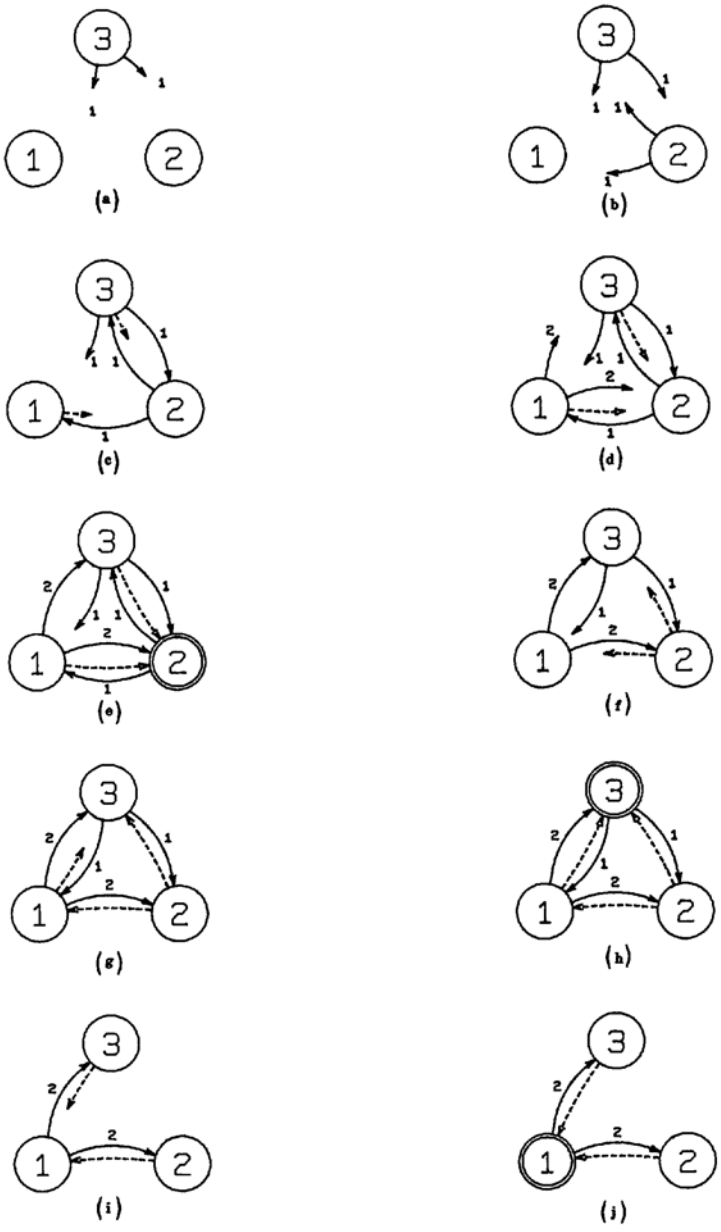
DECISION TABLE

2.3 Example

Imagine a three-node network using this algorithm.Initially the highest sequence number at each node is zero. Solid lines show REQUEST messages; the number is the sequence number of the request. The dashed lines show REPLY messages. In Figure 1 (a), node 3 is the first to attempt to invoke mutual exclusion. It chooses sequence number l and sends REQUEST messages to nodes 1 and 2. Before either message can arrive, node 2 wishes to enter its critical section. It also chooses sequence number 1 and sends REQUEST messages to the other nodes (Figure l(b)). In Figure l(c) node 2's messages have arrived. At node l, which has not yet made a request itself, a REPLY is immediately generated. At node 3, 2's request is found to have an identical sequence number to 3's request; node 2 wins on the node number tie-breaking rule. A REPLY is sent. But at node 2, 3's request is found to have an identical sequence number but loses the tiebreaker.A reply is deferred. Figure l(d) shows node l making a request to enter its critical section. It uses sequence number 2 since it has received a REQUEST message with a sequence number of 1 (from node 2). Owing to an anomaly in the communications system, the REQUEST message to node 2 overtakes the REPLY that is on its way there. No reply message is sent since the message's sequence number is higher than node 2's sequence number. In Figure l(e), node 2 can now enter its critical section since it has received both of the necessary replies.

Node l's REQUEST has also arrived at node 3 but has been deferred since the request's sequence number is higher than that selected by node 3. When node 2 has finished its critical section processing, it sends REPLY messages back to both nodes 1 and 3 (Figure l(f)). In Figure l(g), nodes 1 and 3 have received their REPLY messages from node 2 but not yet from each other. Node 3's request has arrived at node 1. Since it bears a smaller sequence number, a REPLY is immediately generated. Figure l(h) shows node 3 entering its critical section after it received both replies. In Figure l(i), node 3 has finished its critical section processing and is returning the deferred REPLY message to node 1. Finally in Figure l(j), node 1 begins critical section processing. At the conclusion of its critical section, node 1 does nothing since it knows of no other node wishing to invoke mutual exclusion.

Fig. 1. Three-node mutual exclusion example. (See Sect. 2.3.)



2.4 Discussion

The sequence numbers are similar to the numbers used by Lamport's "bakery algorithm." The node with the lowest number is the next one to enter the critical section. Ties are broken by comparing node numbers. A REPLY is generated when its sender agrees to allow the node sending a REQUEST to enter its critical section first. The sequence numbers prevent high numbered nodes from being "shut-out" by lower numbered nodes. Once node A's REQUEST messages have been processed by all other nodes, no other node may enter its critical section twice before node A has entered its critical section. The sequence numbers and node numbers form a virtual ordering among requesting nodes. No one of the nodes has any more information than a list of some or all of the other nodes following it in the virtual order. Yet the system as a whole defines a unique virtual ordering based on a first-come-first-served discipline.

3. Assertions

3.1 Mutual Exclusion

Mutual exclusion is achieved when no pair of nodes is ever simultaneously in its critical section. For any pair of nodes, one must leave its critical section before the other may enter.

ASSERTION. *Mutual exclusion is achieved.*

PROOF. Assume the contrary, that at some time two nodes (A and B) are both in their critical sections at the same time. Examine the message traffic associated with the current cycle of the algorithm that occurred in each node just prior to this condition. Each node sent a REQUEST to the other and received a REPLY.

CASE 1: Node A sent a REPLY to Node B's REQUEST before choosing its own sequence number. Therefore A will choose a sequence number higher than B's sequence number. When B received A's REQUEST with a higher number, it must have found its own Requesting_Critical_Section = TRUE since this is set to be TRUE before sending REQUEST and A had received this request before sending its own REQUEST. The algorithm then directs B to defer the REQUEST and not reply until it has left its critical section. Then node A could not yet be in its critical section contrary to assumption.

CASE 2: Node B sent a REPLY to A's REQUEST before choosing its own sequence number. This is the mirror image of Case 1.

CASE 3: Both nodes sent a REPLY to the other's REQUEST after choosing their own sequence numbers. Both nodes must have found their own Requesting_Critical_Section to be TRUE when receiving the other's REQUEST message. Both nodes will compare the sequence number and node number in the REQUEST message to their own sequence and node numbers. The comparisons will develop opposite senses at each node and exactly

one will defer the REQUEST until it has left its own critical section contradicting the assumption.

Therefore, in all cases the algorithm will prevent both nodes from entering their critical sections simultaneously and mutual exclusion is achieved.

3.2 Deadlock

The system of nodes is said to be deadlocked when no node is in its critical section and no requesting node can ever proceed to its own critical section.

ASSERTION. *Deadlock is impossible.*

PROOF. Assume the contrary, that deadlock is possible. Then all requesting nodes must be unable to proceed to their critical sections because one or more REPLYs are outstanding. After a sufficient period of time, the only reason that the REPLY could not have been received is that the REQUEST is deferred by another node which itself is waiting for REPLYs and cannot proceed. Therefore, there must exist a circuit of nodes, each of which has sent a REQUEST to its successor but has not received a REPLY. Since each node in the loop has deferred the REQUEST sent to it, it must be requesting the critical section itself and have found that the sequence number node number pair in that REQUEST was greater than its own. However, this cannot hold for all nodes in the supposed circuit, and thus the assertion must be true.

3.3 Starvation

Starvation occurs when one node must wait indefinitely to enter its critical section even though other nodes
are entering and exiting their own critical sections.

ASSERTION. *Starvation is impossible.*

PROOF. Assume the contrary, that starvation is possible. Nodes receiving REQUEST messages will process them within finite time since the process which handles them does not block. After processing the REQUEST sent by the starving node, a receiving node cannot issue any new requests of its own with the same or lower sequence number. After some period of time the sequence number of the starving node will be the lowest of any requesting node. Any REQUESTs received by the starving node will be deferred, preventing any other node from entering its critical section. By the previous assertion, deadlock cannot occur and some process must be able to enter its critical section.
Since it cannot be any other process, the starving process must be the one to enter its critical section.

4.Message Traffic

This algorithm requires one message to (REQUEST) and one message from (REPLY) each other node for each entry to a critical section. If the network consists of N nodes, 2*(N - l) messages are exchanged. It will be shown that this number is the minimum required when nodes act independently and concurrently. Hence, the algorithm is optimal with regard to the number of messages exchanged.

4.1 Concurrent Processing

For a symmetrical, fully distributed algorithm there must be at least one message into and one message out of each node. If no message enters/leaves some node, that node must not have been necessary to the algorithm; then the algorithm is not symmetrical or is not fully distributed. Furthermore, to allow the algorithm to operate concurrently at all nodes, the messages entering nodes must not wait for the message generated at the conclusion of processing at other nodes. This would indicate that two separate messages per node are required. The requesting node does not need to send and receive messages to itself, however, and so a total of 2*(N - 1) messages are needed. This number must be a minimum for any parallel, symmetric, distributed algorithm.

4.2 Serial Processing

If the nodes do not act independently of each other, it is possible to reduce the number of messages by using serial node-by-node processing. The first condition discussed earlier (one message into and out of each node) still holds so a minimum of N messages are required. No parallelism can exist in such a structure since a message out of a node must double as the message into some other node. If the algorithm presented here is modified so that messages are sent from node to node sequentially, it achieves the theoretical minimum number of messages in this case also. Parallel operation is necessarily sacrificed.

The modifications required are considered in Section 6.3.

5. Delay in Granting Critical Sections

The algorithm also grants mutual exclusion with minimum delay if some general assumptions are made.

5.1 Definition of Delay

The delay involved in granting the critical section resource is the stretch of time beginning with the requesting node asking for the critical section and ending when that node enters its critical section. The execution time of the instructions in the algorithm is assumed to be negligible compared to the message transmission times.

5.2 Assumptions

The following assumptions prevent the use of central control or extra information derived from timing:

Assumption 1. No information is available bounding transmission time delays or giving actual transit times. Because of this assumption, it takes one round-trip time to determine the state of another node. By adopting this assumption, sending information through timing channels becomes impossible.
Assumption 2. No node possesses the critical section resource when it has not been requested. This assumption prevents a node or series of nodes from acting as a central control because it retained the critical section resource.
Assumption 3. Nodes do not anticipate requests.

5.3 Bounds
Three conditions that put a lower bound on delay times are developed and the mutual exclusion algorithm is shown to achieve these bounds.

5.3.1 Bound 1: Minimum delay time per request.
Before a node enters its critical section, it must make sure that no other node is entering. To do this it must determine the current status of any other node that could take precedence if there is a time overlap and both nodes are said to be requesting concurrently . By assumption 1 this will take at least one round-trip transmission time. By assumptions 2 and 3 this process cannot start before the request arrives. Therefore, no request can be serviced in less than one round-trip time.

5.3.2 Bound 2: Minimum delay time with conflict.
When two nodes are requesting concurrently, they do not know which of them made their request first because of the absence of timing information. A tie-breaking scheme, representing a total ordering among requesting nodes, must be used. Since the tie-breaking rule does not know which node actually made the earlier request, half of the time a critical section grant cannot be made until after the node making the later request has received its round-trip replies. Conflict may also occur with more than two nodes. One of them must be selected by the tiebreaker to be granted access to its critical section first. 5.3.3 Bound 3: System throughput. Once a node has released the critical section resource, no other node can enter its critical section in less than a one-way trip
transmission time. This is the minimum amount of time needed to notify other nodes that critical section processing has been completed and to transmit the new values of network-wide information.

5.4 Compliance
The algorithm achieves these bounds:
CASE A: If when a critical section is released at least one node is eligible to enter its critical section based on Bounds 1 and 2 within a one-way trip time in the future, the algorithm will achieve the more ambitious Bound 3. If the next node to enter its critical section is eligible under Bounds 1 and 2 within a one-way trip time in the

future, then at least one one-way trip time has elapsed already since that node made its

---

request. Since it is next, only the node currently releasing the critical section could be delaying a REPLY message and this REPLY will be triggered by the release of the critical section. This final reply will reach the next node in a one-way
trip time satisfying Bound 3.
CASE B: Case A does not hold. The algorithm achieves Bound 1 or Bound 2 depending upon interference. The node with lowest sequence number/node number pair among requesting nodes will have none of its requests queued by other nodes and, hence, will enter its critical section in the minimum amount of time given by Bounds 1 and 2. In short, the algorithm achieves Bound B whenever it can do so without violating Bounds 1 and 2. The algorithm therefore has minimal delay times under assumptions 1, 2, and 3. The delay time envelope when plotted against arrival rate is discussed further in .When a particular network has closely bounded message delay times and either synchronized docks or knowledge of transit times, this timing information can be used to reduce delay times still further.

6. Modifications

Several interesting modifications can be made to the algorithm to take advantage of different environments.

6.1 Implicit Reply

The REPLY message carries only a single bit of information. When the message transmission time between nodes has an upper bound, the sense of the response can be changed so that no reply within that time period indicates an implicit reply. An explicit message, called "DEFERRED", is sent when REPLY would ordinarily not be sere. The number of messages required by the implicit reply scheme varies between l*(N - l) and 3*(N - l) depending on the number of DEFERRED messages sent. When there is little contention for the critical section resource, the number of messages approaches I*(N- 1). Since a requesting node must usually wait for the maximum round-trip time before entering its critical section, the usefulness of this modification depends on an upper bound for transmission time which is not much larger than the average.

6.2 Broadcast Messages

When the communications structure between nodes permits broadcast messages, the initial REQUEST message can be sent using that mechanism. The message traffic is reduced to N messages, one broadcast REQUEST and (N - 1) REPLYs. If combined with the implicit reply modification discussed above, the message count can be as low as one.

6.2.1 Communications medium sequencing.

Broadcast REQUEST messages need not contain the usual sequence number if their time of successful transmission can be monitored. The broadcast medium enforces serialization of the REQUESTs and a queueing order equivalent to the sequence numbers may be obtained by observing the order of REQUEST messages appearing on the broadcast medium. The REPLY messages can also be broadcast, and only two messages

per critical section invocation are required. REPLYs are only needed from those other nodes which have themselves successfully broadcast a prior REQUEST but received no corresponding REPLY.

6.2.2 No communications medium sequencing.

Even if the order of successful REQUEST broadcasts cannot be monitored, it is useful to broadcast the REPLY messages following critical section processing. The size of the audience depends on the degree of contention. A broadcast REPLY message must contain a list of intended recipients because it is not sufficient for nodes waiting for a REPLY to assume it applies to them. 2 2 Example: While node 1 is performing critical section processing related to its request with sequence number 1, node 2 decides to issue a REQUEST message with sequence number 2. Before the REQUEST message arrives at node 1, node I completes its critical section processing and broadcasts the REPLY it owes some other node(s). Without a list of intended recipients, node 2 might think that the REPLY applies to its REQUEST message and continue. In fact, node 1 may make a new request with sequence number 2 and be entitled to enter its critical section first due to the tie-breaking rule.

6.3 Ring Structure

The number of messages can be cut to N by processing the requests serially through a logical circuit consisting of all nodes instead of allowing processing to proceed concurrently. N is the minimum number of messages required for any distributed symmetric algorithm when broadcasting is not available and information is not sent via timing channels. 3 The algorithm must be modified by replacing the REPLY message with an echo of the REQUEST message. As the REQUEST message travels around the circuit of nodes, it may be deferred at several stops. When it is received at the initiating node, mutual exclusion has been achieved and critical section processing may begin. A further possible modification sends the REQUEST message from node to node around the circuit without pause but the notation "DEFERRED by node j" is added by each node j that is copying and deferring the request. The Outstanding_Reply_Count is then set according to the notations when it arrives back at the initiating node. The nodes which have marked the REQUEST as deferred generate individual REPLYs in the usual way. This technique comes close to N messages while eliminating the cumulative delays at each stop.

6.4 Bounding Sequence Numbers

The sequence numbers in the algorithm increase at each critical section invocation and are theoretically unbounded. The ticket numbers of the "bakery algorithm" suffer from the same problem. A technique for limiting the amount of storage necessary to hold these unbounded numbers can be borrowed from computer communications protocols. Although the numbers themselves are unbounded, their range is bounded. The sequence numbers increase by no more than one each time a node requests entry to its critical section. That request cannot be granted as long as a lower sequence number request is outstanding. Therefore the numbers must fall within the range from *X to X + N--1*.

The sequence numbers can be stored modulo M where M _> 2N - 1. When making a comparison, the smaller number should be increased by M if the difference is N or more. Thus only *log2(2N* - 1) bits of storage are needed regardless of the number of times the critical section is entered.

6.5 Sequence Number Incrementation

Aside from this method for limiting the storage required to hold sequence numbers, there is no reason for incrementing sequence numbers in unit steps. Two situations make larger increments attractive:

(1) The algorithm tends to favor lower numbered nodes slightly, owing to the tie-breaking rule. This favoritism can be reduced by incrementing the sequence number by a random integer. The tie-breaking node number is still required in case the random integers used were equal.

(2) Deliberate priority can be introduced by instructing high priority nodes to use small increments and low priority nodes to use large increments. In addition, high priority nodes may be allowed to monopolize critical section processing until forced to increment their sequence numbers past the one chosen by a lower priority node. In doing so, the process at a high priority node which receives and handles messages may choose to delay acting on those received from low priority nodes in order to keep the Highest_Sequence_Number from being prematurely incremented past the one chosen by the low priority node.

a To involve all nodes, at least one message must be received and one sent per node. The minimum number of messages that meet this requirement is N.

6.6 Readers and Writers

The algorithm is easily modified to solve the "Readers and Writers" problem where writers are given priority. The modification is simply that "readers" never defer a REQUEST for another "reader"; instead they always REPLY immediately. "Writers" follow the original algorithm.

7. Considerations for Practical Networks

7.1 Node Numbers

It is more convenient to draw node numbers from a larger range than 1... N. The algorithm may be changed to map the integers 1... N into the actual node numbers by indexing a table NAMES [1... N]. The comparison of node numbers should then be performed by comparing the values contained in NAMES.

7.2 Insertion of New Nodes

New nodes may be added to the group participating in the mutual exclusion algorithm. They must be assigned unique node numbers, obtain a list of participating nodes, be

placed on every other node's list of participants, and acquire an appropriate value for their Highest_Sequence_Number variable.

7.2.1 Restart interval.

If the node could have been previously operational in the group (e.g., it failed and is now restarting), it should first notify other nodes that it failed and then wait long enough to be sure its old messages were delivered and the network processed its removal. Usually the network will already be aware of the node's failure, but this cannot be assumed. If this step was not followed, the failure may be detected at approximately the same time as the node rejoins the group. This would result in conflicting bookkeeping at different nodes.

7.2.2 Reconcile participant lists.

A new node must obtain a list of other participating nodes and have itself added to the others' lists. A new node should contact a "sponsor" node which is already participating in the group. The sponsor should then invoke mutual exclusion, initialize the new node's participant list from its own, and broadcast the new node's identity before releasing mutual exclusion. Each node receiving this notification adds the new node number to its NAMES array and increments N, the number of active nodes. An alternative is possible if the communications network can deliver a message to all other nodes without the sender naming all the other nodes in the network. In this case a new node obtains a list of participants from a nearby node and then sends a broadcast message asking all other nodes to include it on their list of participating nodes.

7.2.3 Set highest sequence number.

The Highest_Sequence_Number variable of a new node must not be set to any value lower than the sequence number of any REQUEST message which would already have been received had the new node been continuously active. Until an appropriate value of Highest_Sequence_Number is obtained, mutual exclusion cannot be requested and incoming REQUEST messages are processed normally. A new node can determine that its Highest_Sequence_Number is high enough by several methods.

(1) Ask all other nodes for their Highest_Sequence_Number and use the largest.

(2) Wait until one REQUEST message has been received from every other node.

(3) Wait until the sequence numbers on REQUEST messages have increased by N - 1.

(4) Wait until all (N - 1) nodes would have time to enter and leave their critical sections even if they all had outstanding requests. This requires the ability to bound message transmission times and critical section times. If no REQUEST message is received during this time, the value of Highest_Sequence_Number from any nearby node can be used.

(5) Wait until the fourth REQUEST message is received from a single node. This method requires that messages are sent and delivered in the same order. The new node may request access to its critical section after any of the above methods has been used to verify that its Highest_Sequence_Number variable is sufficiently high.

7.3 Removal of Nodes

A node wishing to leave the group may do so by notifying all other nodes of its intention. The other nodes should acknowledge this message. While waiting for acknowledgement,

---

the departing node may not request mutual exclusion and must continue to send REPLY messages to any REQUEST messages it receives. Each node checks to see if the departing node is listed in its NAMES array, and if so, removes it and decrements the value of N, the number of active nodes, by one. If messages may be delivered out of order, a node awaiting a REPLY message from a departing node should pretend the REPLY was received.

7.4 Node Failures

In practice some nodes fail and will not respond to messages directed at them. To prevent this situation from stopping the proposed mutual exclusion algorithm, a timeout-recovery mechanism may be added. The timeout detection of a failed node relies on knowledge of an upper bound on the time which may elapse before a working node responds to a message and an estimate of the maximum processing time within a critical section. The only message in the original algorithm which demands a response is the REQUEST message. A requesting node should start a timer when the REQUEST messages are sent. The timer should be restarted when a REPLY is received and cancelled when the critical section processing begins.

A bit map, Awaiting_Reply [1... N], can be used to identify which nodes have not yet sent a REPLY message. The Awaiting_Reply array is set to all TRUE values before a REQUEST message is issued. Individual bits are turned off when REPLY messages are received. If the timer expires, 4 all nodes for which Awaiting_Reply is TRUE are suspected of having failed. A probing message, ARE_YOU_THERE(me), should be sent to each suspect node. If no answer is received during a second timeout period, ~ the suspect node has failed. When an ARE_YOU_THERE(j) message is received,

Reply_Deferred[j] should be examined. If it is FALSE, it must be that the REQUEST was not received, the REPLY was lost, or the node has restarted; the correct response is REPLY(me). If Reply_Deferred[j] is TRUE, a YES_I_AM_HERE message should be sent to confirm that the node is alive. The timeout does not impose an upper limit on the duration of a critical section. If critical section processing exceeds the timeout, all nodes will respond with YES I AM_HERE messages and a new timeout period may

begin. When it has been determined that node j has failed, this can be broadcast by the node detecting the failure. Any node which is awaiting a REPLY message from the failed node should pretend that a REPLY was received. In addition the node should be erased from the NAMES array if present and N, the number of active nodes, decremented by one. If the failed node recognizes that it has failed and **has** been restarted, it may return to the group through the mechanism for adding a new node. If it does not know that it has failed and issues new REQUEST messages, any node which receives the REQUEST message and does not fred the node's name in its NAMES array may return a special message notifying the node that it should restart itself and use the insertion protocol.

4 The appropriate value is worst-case round-trip message transmission
time plus worst-case processing time at the distant node plus
a reasonable estimate of maximum critical section time.
In this case just round-trip message time plus worst-case processing time at the distant node.

The Effect of Message Ordering

The algorithm presented in this paper does not depend on messages being delivered or acted upon in the order in which they are sent. If such a condition does exist, there is a stronger limit to the number of times other nodes can enter their critical sections before a requesting node A can. Without delivery in order of transmission, the worst case anMysis shows that $N(N + 1)/2 - 1$ nodes can enter their critical section before Node A may.

To determine this bound, assume that A has the highest node number and therefore the least priority in breaking ties. A's sequence number may be (N - 1) higher than the lowest outstanding sequence number. (See Section 6.4.) It is possible, by judiciously ordering the delivery of messages, for each other node to enter its critical section with its sequence number taking on each value between its current value and A's value. To get the worst case, assume that all nodes have chosen a distinct sequence number with A's number the highest. Therefore, one node can enter its critical section N times before A may, another (N - 1), another (N - 2) and so on down to the node whose REQUEST message caused A's sequence number selection. This takes two critical section entries at most. This sum, N + (N - 1) + (N - 2) +... + 3 + 2, is the number of times other nodes may enter their critical section after A has made a request in the worst case.

If delivery is guaranteed to be in the order of transmission, no other node may enter its critical section more
than twice between the time that A selects a sequence number and A is permitted to enter its critical section. No more than 2*(N - 1) critical sections are possible before A may enter.

To get this bound observe that after node .4 has done its "Node Requests Critical Section" processing, it cannot receive more than one
REQUEST from another node (j) which contains a lower or equal sequence number. By the time it gets the REPLY from this REQUEST, it must also have received A's REQUEST; it cannot thereafter select a lower or equal sequence number. Each other nodej can enter its critical section at most once because of an already approved REQUEST and once with the one REQUEST which contains a lower or equal sequence number. If every other node follows this worst case pattern, at most 2*(N - 1) critical section entries may preceed A's .

When delivery in order is used, a new node may assume its Highest_Sequence_Number is synchronized when it has heard the fourth REQUEST message from the same node.

Assume that a nodej sent its REQUEST messages before the new node came on-line. The new node is not synchronized until it holds a higher number in Highest_SequenceNumber than the sequence number used by j. The reference node B (which is generating the four requests) can enter its critical section at most twice before node j enters its critical section. Therefore, by the time B enters its critical section the third time, no nodes like j exist which did not know about the

new node when they made their requests. Reference node B may have issued three REQUEST messages seen by the new node before entering its critical section for the third time. The fourth REQUEST message guarantees that the critical section was entered for the third time.

Program no. 1

AIM
This program seeks to illustrate the working of Ricart and Agrawala algorithm in a totally connected network i.e. where every node is directly connected to every other node.

PROPERTIES AND ASSUMPTIONS

- There are 3 independent processes a, b and c.
- A process can be in 3 possible states Idle, Requesting and Critical.
- A process has 2 queues- one is a list of processes which are waiting for an acknowledgement and the other is a list of acknowledgements the process has received
- A process cannot make a new request until its previous request has been serviced.
- A process requires 1 complete clock cycle to complete its critical section.
- A message generated by a source during clock cycle is delivered to the source at the end of the same cycle i.e. communication delays are considered negligible.

INPUT
As input it accepts an ordered list of process names along with their time stamps.

OUTPUT
The output is a snapshot of the system at intervals of 1 clock period.

//Program showing the working of Ricart and Agrawala algorithm

```
#include<iostream.h>
#include<conio.h>
#include<process.h>

struct req
{
  int t;
  char name;
}
R[5];

class proc
{
   public:
  char st,name,curr;
  req reqs[3];
  int t,j,f,v,acks[3];
  proc()
         {
           j=-1;
           t=55;
          st='I';
          curr='I';
 }
  void rqst(req c){ reqs[++j]=c;}
  void acksend();
  void ckcl()
      { if((curr=='R')&&(acks[0]==1)&&(acks[1]==1)&&(acks[2]==1))
             { st='C';
               acks[0]=acks[1]=acks[2]=0;
             }
      }
```

```
}
P[3];
```

```
void main()
{
 int q=0;
 P[0].name='x';
 P[1].name='y';
 P[2].name='z';
 clrscr();
 cout<<"Enter 5 processes along with time stamps \n";
 for(int b=0;b<5;b++) cin>>R[b].name>>R[b].t;
 cout<<endl;
 int y=1;
 cout<<"Time       Process x          Process y          Process z \n"
    <<"       state  reqs  acks   state  reqs  acks   state  reqs  acks \n";
 do
  {
  //first check for critical section condition
     for(b=0;b<3;b++) if(P[b].curr=='C') P[b].t=50;
     P[0].ckcl();
     P[1].ckcl();
     P[2].ckcl();

 //secondly check if acknowledgements need to be sent
     P[0].acksend();
     P[1].acksend();
     P[2].acksend();

//thirdly send requests
for(int q=0;q<5;q++)
 { if(R[q].t==y)
  {
   switch(R[q].name)
    {
      case 'x': if(P[0].curr!='R')
                                { P[0].st='R';
                                  P[0].t=y; }
                 else{  cout<<"Process  x  cannot  request  critical  section  again  untill  its
previous request has been serviced";
                                 getch();
                                 exit(0);
                               }
                     break;
```

```
    case 'y': if(P[1].curr!='R')
                        { P[1].st='R';
```

```
                          P[1].t=y;
                        }
                else{ cout<<"Process y cannot request critical section again untill its
previous request has been serviced";
                     getch();
                     exit(0);
                    }
                break;
    case 'z': if(P[2].curr!='R')
                        { P[2].st='R';
                          P[2].t=y;
                        }
                else{ cout<<"Process z cannot request critical section again untill its
previous request has been serviced";
                     getch();
                     exit(0);
                    }
                break;
    }
    for(int u=0;u<3;u++) P[u].rqst(R[q]);
  }  //end if
 } //end for
  for(b=0;b<3;b++) if((P[b].curr==P[b].st)&&(P[b].st=='C')) P[b].st='I';
  P[0].curr=P[0].st ;
  P[1].curr=P[1].st ;
  P[2].curr=P[2].st ;
if(y<10)
{
cout<<"          "<<y<<"                                                    "<<P[0].st<<"
"<<P[0].reqs[0].name<<P[0].reqs[1].name<<P[0].reqs[2].name<<"
"<<P[0].acks[0]<<P[0].acks[1]<<P[0].acks[2];
        cout<<"                                                             "<<P[1].st<<"
"<<P[1].reqs[0].name<<P[1].reqs[1].name<<P[1].reqs[2].name<<"
"<<P[1].acks[0]<<P[1].acks[1]<<P[1].acks[2];
        cout<<"                                                             "<<P[2].st<<"
"<<P[2].reqs[0].name<<P[2].reqs[1].name<<P[2].reqs[2].name<<"
"<<P[2].acks[0]<<P[2].acks[1]<<P[2].acks[2];
cout<<endl;
getch();
}
else
```

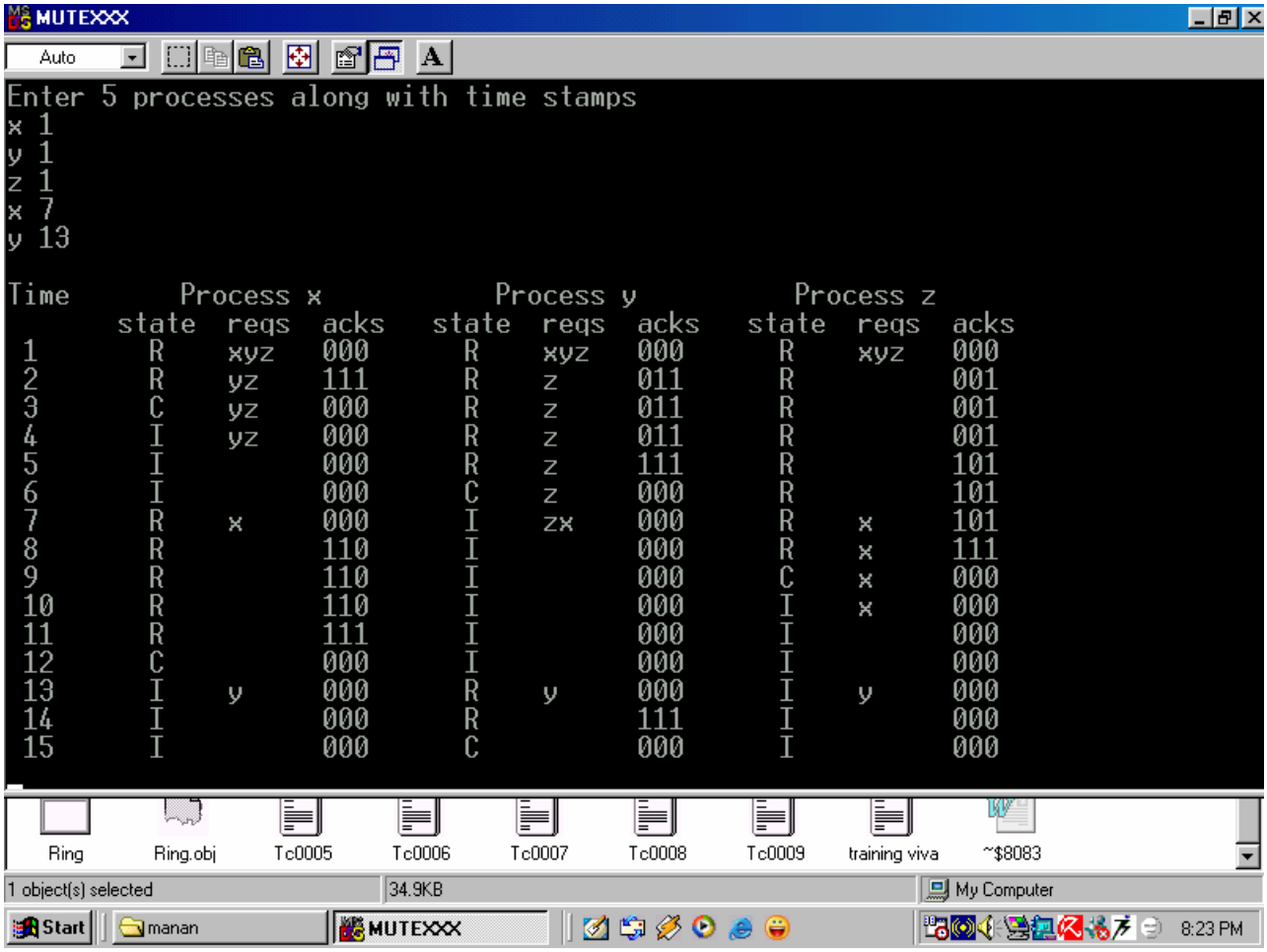
```
{
```

```
cout<<"            "<<y<<"                                                    "<<P[0].st<<"
"<<P[0].reqs[0].name<<P[0].reqs[1].name<<P[0].reqs[2].name<<"
"<<P[0].acks[0]<<P[0].acks[1]<<P[0].acks[2];
        cout<<"                                                    "<<P[1].st<<"
"<<P[1].reqs[0].name<<P[1].reqs[1].name<<P[1].reqs[2].name<<"
"<<P[1].acks[0]<<P[1].acks[1]<<P[1].acks[2];
        cout<<"                                                    "<<P[2].st<<"
"<<P[2].reqs[0].name<<P[2].reqs[1].name<<P[2].reqs[2].name<<"
"<<P[2].acks[0]<<P[2].acks[1]<<P[2].acks[2];
cout<<endl;
getch();
}
y++;
}while((y<=R[4].t)||!((P[0].st=='I')&&(P[1].st=='I')&&(P[2].st=='I')));
getch();
}  //end of main

 void sendack( char rcvr,int sndr)
 { switch(rcvr)
   {
    case'x':P[0].acks[sndr]=1;break;
    case'y':P[1].acks[sndr]=1;break;
    case'z':P[2].acks[sndr]=1;break;
    default:cout<<"wrong process name"<<rcvr<<sndr;
            getch();
            exit(0);
            break;
    }
 }

 void proc::acksend()
 {  if(curr=='I') f=55;
    else if(curr=='R') f=t;
    else f=-1;
    int shift=0;
    for(int g=0;g<=j;g++)
        { if((reqs[g].t<f)||((reqs[g].t==f)&&(name>=reqs[g].name)))
           { switch(name)
                  { case'x':v=0;break;
                    case'y':v=1;break;
                    case'z':v=2;break;
```

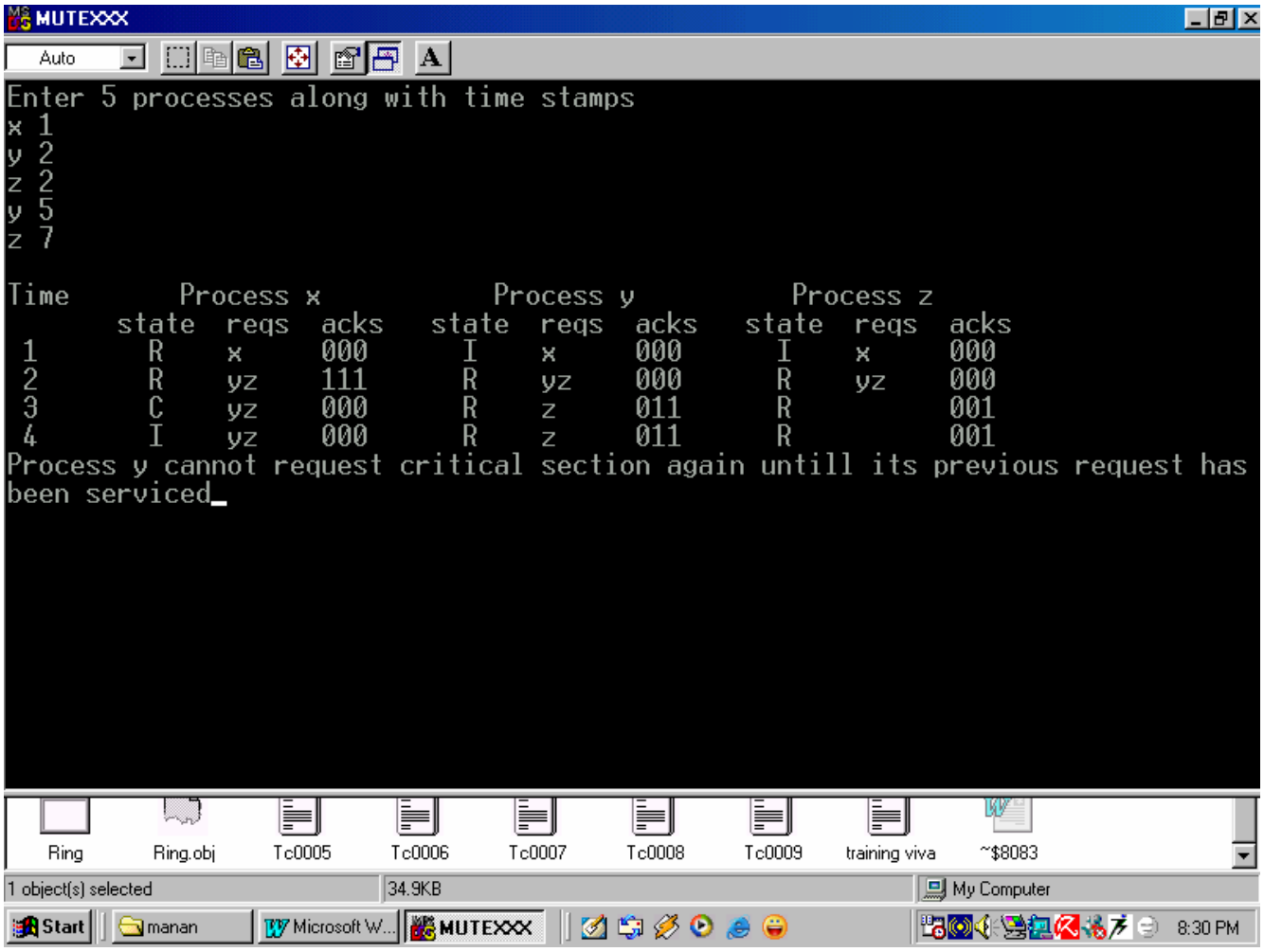
```
                default:cout<<"wrong process name in acksend";
                        getch();
```

```
                                exit(0);
                                break;
                        }
                sendack(reqs[g].name,v);
                reqs[g].name='\0';
                reqs[g].t=50;
                shift++;
                }
        }
 for(int ric=0;ric<=j-shift;ric++) reqs[ric]=reqs[ric+shift];
 for(int agr=ric;agr<=j;agr++)
                                        {
                                 reqs[agr].name='\0';
                                 reqs[agr].t=55;
                                }
 j=j-shift;
 }
```

Example of a correct output



Example of incorrect input leading to an error message



Program no. 2

AIM
This program seeks to illustrate the working of Ricart and Agrawala algorithm in a ring network where messages are passed in a predefined circular order.

PROPERTIES AND ASSUMPTIONS

- There are 4 independent processes 1,2,3 and 4.
- A process can be in 3 possible states Idle, Requesting and Critical.
- A process has one queue which is a list of processes which are waiting to be passed on to the next node
- A process cannot make a new request until its previous request has been serviced.
- A process requires 1 complete clock cycle to complete its critical section.
- Communications delay is negligible

INPUT
As input it accepts an ordered list of process names along with their time stamps.

OUTPUT
The output is a snapshot of the system at intervals of 1 clock period.

```
//implementation of ricart and agrawala algorithm for a ring network

#include<iostream.h>
#include<conio.h>
#include<process.h>

struct msg
{
  int no;
  int t;
};

class node
{
  public:
  int t,j;
  msg que[4],temp;
  char prev,next;
  node()
          {  t=50;
             j=-1;
           prev='I';
           next='I';
           que[0].t=55;
           que[1].t=55;
            que[2].t=55;
           que[3].t=55;
           }
  void rcv(msg);
  void cchek(int);
  void pass(int,int);
}
N[5];

void main()
{
  msg in[5],buff[3];
  clrscr();
  cout<<"Enter the order of requests along with time stamps \n";
  for(int b=0;b<5;b++) cin>>in[b].no>>in[b].t;
```

```
int y=1;
 do {
```

```
   //check for criticality
   for(int c=0;c<4;c++)
    N[c].cchek(c);

//passing messages
   for( c=3;c>=0;c--)
    if(N[c].prev!='C')
     N[c].pass(c+1,c);
   if(N[3].prev!='C') N[4].pass(0,3);

 //pass request
   for(int h=0;h<5;h++)
    if(in[h].t==y)
     { if(N[in[h].no-1].prev=='R')
                                        {
                                       cout<<"error";
                                        getch();
                                       exit(0);
                                    }
       if(in[h].no==4) N[0].rcv(in[h]);
       else N[in[h].no].rcv(in[h]);
       N[in[h].no-1].next='R';
       N[in[h].no-1].t=y;
     }
    for(h=0;h<4;h++)
     if((N[h].prev=='C')&&(N[h].next=='C'))
      {
       N[h].t=50;
       N[h].next='I';
      }
   for(h=0;h<4;h++) N[h].prev=N[h].next;
cout<<"t="<<y;
cout<<"                                                                    s1="<<N[0].next<<"
q1="<<N[0].que[0].no<<N[0].que[1].no<<N[0].que[2].no<<N[0].que[3].no;
cout<<"                                                                    s2="<<N[1].next<<"
q2="<<N[1].que[0].no<<N[1].que[1].no<<N[1].que[2].no<<N[1].que[3].no;
cout<<"                                                                    s3="<<N[2].next<<"
q3="<<N[2].que[0].no<<N[2].que[1].no<<N[2].que[2].no<<N[2].que[3].no;
cout<<"                                                                    s4="<<N[3].next<<"
q4="<<N[3].que[0].no<<N[3].que[1].no<<N[3].que[2].no<<N[3].que[3].no;
cout <<endl;
```

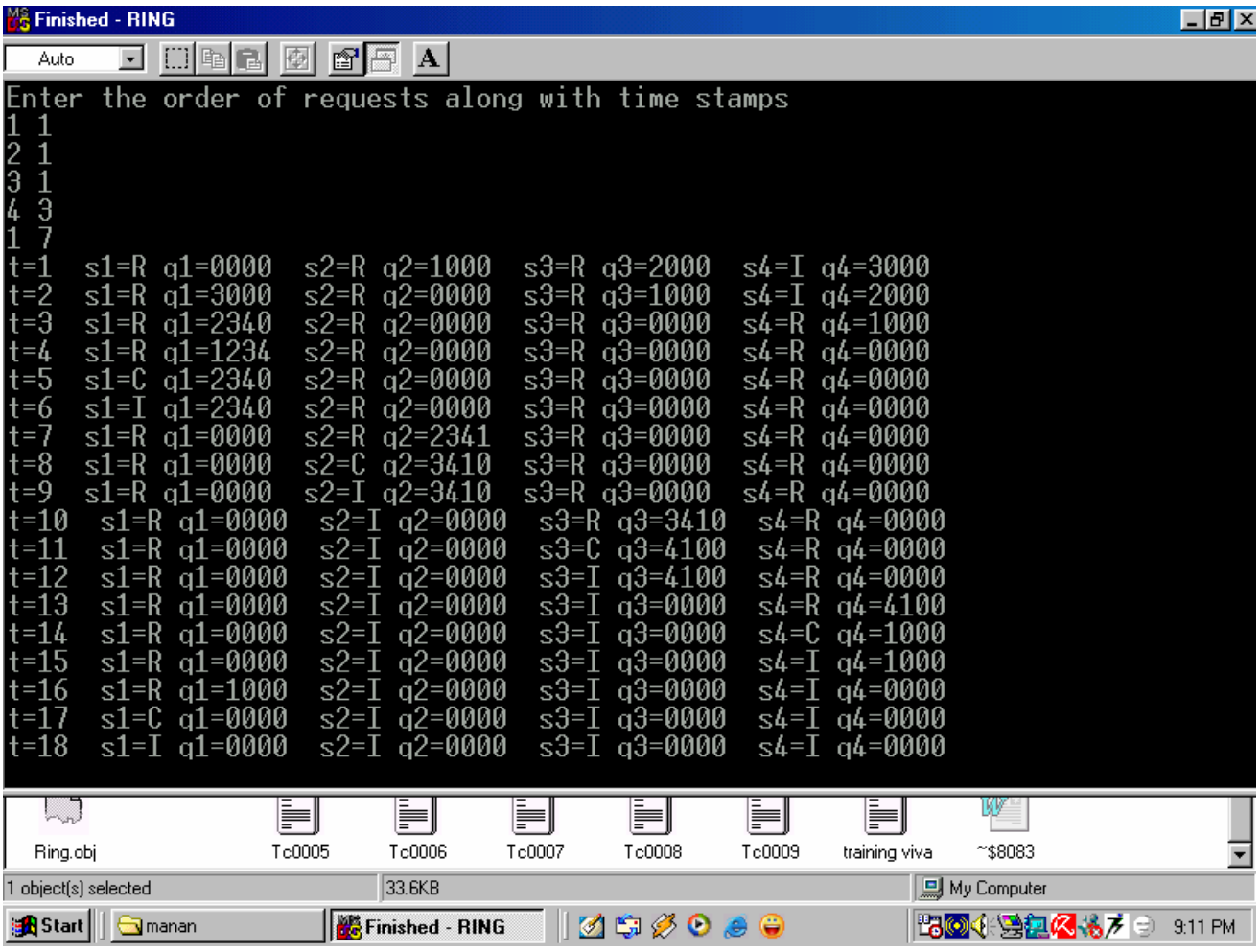
```
getch();
y++;}
```

```
while((y<=in[4].t)||!((N[0].next=='I')&&(N[1].next=='I')&&(N[2].next=='I')&&(N[3].ne
xt=='I')));
getch();
}
//end of main

void node::cchek(int n)
{
  if((que[0].t==t)&&(que[0].no-1==n))
   {
    next='C';
   for(int l=0;l<j;l++)que[l]=que[l+1];
   que[j].no=0;que[j].t=55;
   }
}

void node::pass(int nxt,int i)
{  int shift=0;
   for(int c=0;c<=j;c++)
    {
     if((que[c].t<t)||((que[c].t==t)&&(que[c].no-1<i)))
         {
          N[nxt].rcv(que[c]);
          shift++;
         }
    }
   for(int ric=0;ric<=j-shift;ric++) que[ric]=que[ric+shift];
   for(int agr=ric;agr<=j;agr++)
                                    {
                                    que[agr].no=0;
                                    que[agr].t=55;
                                    }
   j=j-shift;}
void node::rcv(msg m)
{ que[++j]=m;
    for(int i=0;i<=j-1;i++)
     for(int k=i+1;k<=j;k++)
          if(que[i].t>=que[k].t)
            { temp=que[i];
              que[i]=que[k];
              que[k]=temp;
```

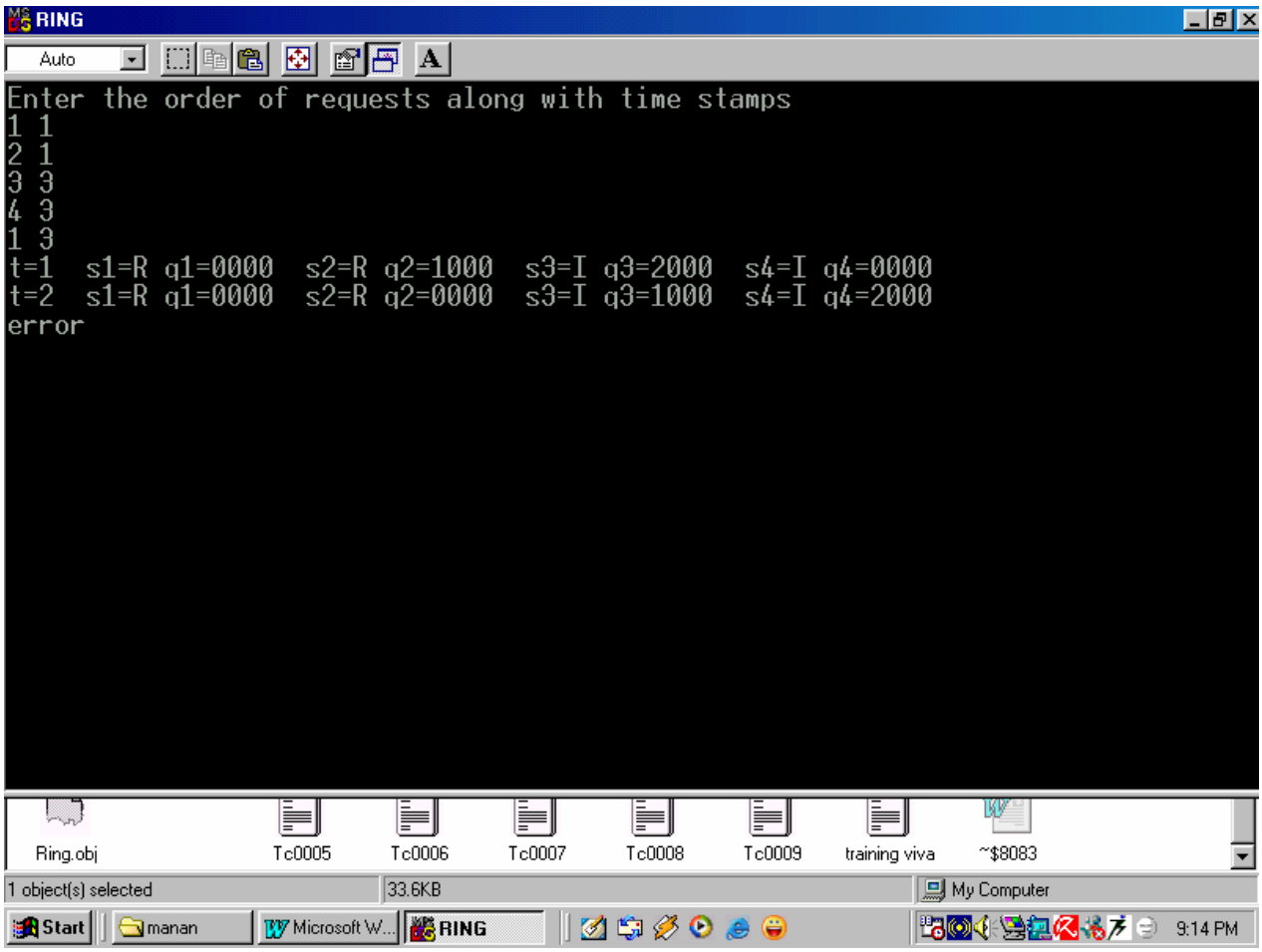
```
if(que[i].t==que[k].t)
 { if(que[i].no>que[k].no)
```

```
{ temp=que[i];
  que[i]=que[k];
  que[k]=temp;
} } } }
```

Correct output for a ring network



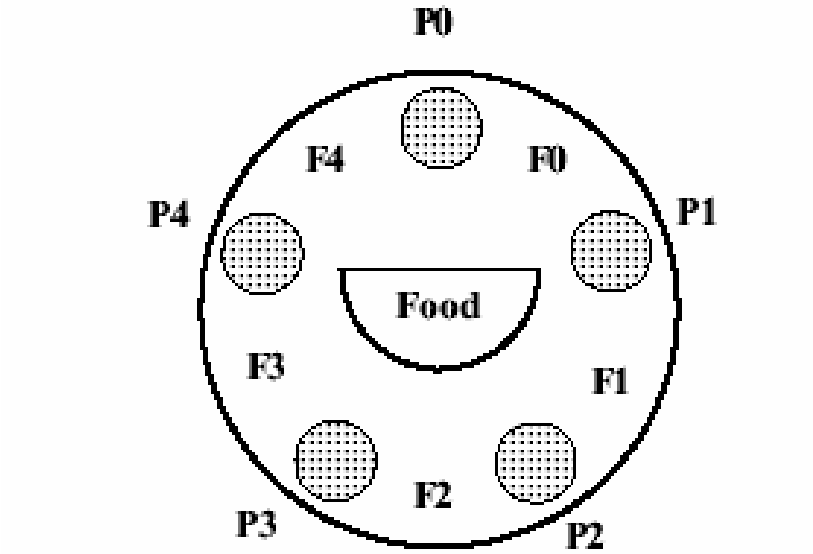
Incorrect input where process 1 makes a request without the service of the first request



A Classic Problem - Dining Philosophers

The Dining Philosophers problem is a classic OS problem that's usually stated in very non-OS terms: There are *N* philosophers sitting around a circular table eating spaghetti and discussing philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only *N* forks, one between each 2 philosophers. Philosopher can be in one of the three states : Thinking, Hungry, Eating. Design an algorithm that the philosophers can follow that ensures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

Why describe problems this way? Well, the analogous situations in computers are sometimes so technical that they obscure creative thought. Thinking about philosophers makes it easier to think abstractly. And many of the early students of this field were theoreticians who like abstract problems.



HERE'S AN APPROACH TO THE DINING PHILOSOPHER THAT IS SIMPLE AND WRONG:

```
void philosopher()
{
while(1) {
sleep();
get_left_fork();
get_right_fork();
eat();
put_left_fork();
put_right_fork();
}
}
```

If every philosopher picks up the left fork at the same time, none gets to eat - ever.
Suppose all five Philosophers take their left fork simultaneously . None will be able to take their right forks , and their will be deadlock.

Some other suboptimal alternatives:

- Pick up the left fork, if the right fork isn't available for a given time, put the left fork down, wait and try again. (Big problem if all philosophers wait the same time - we get the same failure mode as before, but repeated.) Even if each philosopher waits a different random time, an unlucky philosopher may starve (in the literal or technical sense).
- Require all philosophers to acquire a binary semaphore before picking up any forks. This guarantees that no philosopher starves (assuming that the semaphore is fair) but limits parallelism dramatically.

An optimal solution to Dining Philosopher Problem should have the following characteristics:

- It enforces mutual exclusion. Each fork can only be held by *at most* one person at a time.
- It is deadlock free . It successfully avoids the situation in which 2 or more philosophers are involved in a cyclic waiting: each philosopher is waiting for a fork that is held by another philosopher while none of them can get enough forks to eat.
- It enforces fairness . All philosophers should perform all required actions in sequence. No privilege or discrimination is given to any of the 5 philosophers.

CORRECT SOLUTION

In correct solution , all the Philosophers try to acquire both the forks simultaneously by calling function Test which tests that neither of the Philosopher's neighbor are eating .

If it is true it returns with both the forks and enters the state Eating. Else if one of it's neighbor is eating the Philosopher blocks on a it's semaphore. So when his neighbor stop eating , he is signaled . The solution uses an array of semaphore one for each Philosopher and an array to store the state of each Philosopher. It uses semaphore mutex to restrict entry to the critical section.

IMPLEMENTATION

The above solution is implemented in Java using multithreading. Each Philosopher runs as a separate thread. There is also one main thread which shows the sate of five Philosophers at various points of time . The State 0 shows that the Philosopher is Thinking, State 1 shows that the Philosopher is Hungry and State 3 depicts Philosopher Eating. Initially all the Philosophers are in State 0 i.e. Thinking. Philosopher are numbered from 0 to 4. The class Semaphore implements the Up() and Down() functions using Signal() and Notify(). Class Newthread is multithreaded to implement five Philosophers.

/* SEMAPHORE CLASS*/

```
class Semaphore {
        private int count;
        public Semaphore(int n) {
                this.count = n;
        }

        public Semaphore()
    {
      this.count=0;
    }
    public synchronized void down() {
                while(count == 0) {
                        try {
                          wait();
                        } catch (InterruptedException e) {
                          System.out.println("hello thread intercepted");
                        }
                }
                count--;
        }

        public synchronized void up() {
                //System.out.println("inside up");
           count++;
                notify(); //alert a thread that's blocking on this semaphore

 }
  }
```

```
 /* DINING PHILOSOPHER */

class Newthread implements Runnable
{
 static private Semaphore mutex=new Semaphore(1);
 static private Semaphore s =new Semaphore(0);
 static private Semaphore t =new Semaphore(0);
 static private Semaphore u =new Semaphore(0);
 static private Semaphore v =new Semaphore(0);
 static private Semaphore w =new Semaphore(0);

 static int state[]= {0,0,0,0,0};
 int serial;
 String name;
 Thread a;
 static int i=1000;

  public Newthread(int i, String name)
  {
   serial=i;
     a=new Thread(this,name);
     a.start();
  }

  public void run()
 {
   while(i>=1)
  {
   int randy=(int)(Math.random()*10);

   try
   {
   Thread.sleep(randy);
   }
   catch (InterruptedException e)
   {
   System.out.println("hello thread intercepted");
   }

   takeforks(serial);
   try {
   Thread.sleep(1000); }
```

```
   catch (InterruptedException e){
   System.out.println("bye thread intercepted");
    }
   i--;
   putforks(serial);
  }
 }

  void takeforks(int i)
  {
   mutex.down();
   state[i]=1;

   test(i);

//   System.out.println("inside fork");
   mutex.up();

   if(i==0)
    s.down();
   else if(i==1)
    t.down();
   else if(i==2)
    u.down();
   else if(i==3)
    v.down();
   else if(i==4)
    w.down();
}

   void putforks(int i)
   {
    mutex.down();
    state[i]=0;
    test((i+5-1)%5);
    test((i+1)%5);
    mutex.up();
   }

   void test(int i)
   {
    if(state[i]==1 && state[(i+5-1)%5]!=2 && state[(i+1)%5]!=2) {
       state[i]=2;
```

```
      //System.out.println("inside test");
      if(i==0)
   s.up();
  else if(i==1)
   t.up();
  else if(i==2)
   u.up();
  else if(i==3)
   v.up();
  else if(i==4)
   w.up();
    }
  }
 }


 class Philosopher{

public static void main (String args[]){
 Newthread ob1=new Newthread(0,"one");
 Newthread ob2=new Newthread(1,"two");
 Newthread ob3=new Newthread(2,"three");
 Newthread ob4=new Newthread(3,"four");
 Newthread ob5=new Newthread(4,"five");

/* ob1.start(); ob2.start(); ob3.start(); ob4.start(); ob5.start();*/

   for(int i=0;i<8;i++)
   {
    try {
   Thread.sleep(1000);

   System.out.println("Philosopher"+ob1.serial+"  "+ Newthread.state[ob1.serial]);
   System.out.println("Philosopher"+ob2.serial+"  "+ Newthread.state[ob2.serial]);
   System.out.println("Philosopher"+ob3.serial+"  "+ Newthread.state[ob3.serial]);
   System.out.println("Philosopher"+ob4.serial+"  "+ Newthread.state[ob4.serial]);
   System.out.println("Philosopher"+ob5.serial+"  "+ Newthread.state[ob5.serial]);
   System.out.println(" ");

}
   catch (InterruptedException e){
    System.out.println("main thread intercepted");
   }}}}
```

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\Documents and Settings\hunny>c:

C:\>cd jdk1.5.0_01

C:\jdk1.5.0_01>cd bin

C:\jdk1.5.0_01\bin>javac Philosopher.java

C:\jdk1.5.0_01\bin>java Philosopher
Philosopher0  2
Philosopher1  1
Philosopher2  1
Philosopher3  2
Philosopher4  1

Philosopher0  1
Philosopher1  2
Philosopher2  1
Philosopher3  1
Philosopher4  2

Philosopher0  2
Philosopher1  1
Philosopher2  2
Philosopher3  1
Philosopher4  1

Philosopher0  1
Philosopher1  0
Philosopher2  2
Philosopher3  0
Philosopher4  2

Philosopher0  2
Philosopher1  1
Philosopher2  1
Philosopher3  2
Philosopher4  1
```

Philosopher0 1
Philosopher1 2
Philosopher2 1
Philosopher3 1
Philosopher4 2

Philosopher0 2
Philosopher1 1
Philosopher2 2
Philosopher3 1
Philosopher4 1

Philosopher0 1
Philosopher1 2
Philosopher2 1
Philosopher3 2
Philosopher4 1

OUTPUT

The output of the program shows the states of the five Philosopher at various instances of time.

VERIFICATION

From the output generated we can see that program fulfills all characteristics of a good solution to Dining Philosopher problem with five Philosophers.

- None of the neighbors are eating simultaneously which ensures mutual exclusion.
- Only two Philosopher are eating at a time .
- All the Philosopher get their turn to eat . So it ensures fairness.
- The program is deadlock free.

THE BOUNDED-BUFFER PRODUCER-CONSUMER PROBLEM

A buffer of size N is shared by several processes. The producer process adds items to the buffer; the consumer process removes items from the buffer. Both processes must be synchronized so that the producer does not try to add to a full buffer, and the consumer does not try to remove an item from an empty buffer.

A simple solution to this problem is to use a variable Count to keep the track of the number of items in the buffer . If the maximum number of items buffer can hold is N, the

Producer's code will first test to see if Count is N. If it is , the Producer will go to sleep, if it is not the Producer will add an item and increment Count.

The Consumer also first test Count to see if it 0. If it is , it goes to sleep , if nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other is sleeping, and if it is wakes it up.

```
Void producer(void)
{
 while(true) {
                produce_item(item);
                if(count==N)sleep();
                enter_item(item);
                count=count+1;
                 if(count==1)wakeup(consumer);
 }
}
```

```
void consumer(void)
{
 int item;

while(true) {
if(count==0) sleep();
remove_item(item);
count=count-1;
if(count==N-1) wakeup(producer);
consume_item(item);
}
```

In the above solution , race condition occur. It can occur because access to count is unconstrained . The following soulution can occur. The buffer is empty and the consumer has just read Count to see if it is 0. At that instant , the scheduler decides to stop running the consumer temporarily and start running the Producer. The Producer enters the item in the buffer , increments Count and notice that it is now 1. Reasoning that Count was just 0, and thus the Consumer must be sleeping , the Producer calls wakeup to wake the consumer up.

The consumer is not yet logically sleep, so the wakeup signal is lost. When the consumer next runs , it will test the value of count previously read, find it to be 0 and go to sleep. Sooner or later the producer will fill up the buffer and go to sleepforever.

So the above solution does not solves the problem.

In our optimal solution ,we want functions Add (item) and Remove(item) such that the following conditions hold:

1. Access to buffer is mutually exclusive: At any time at most one process should be executing add (item) or remove (item).
2. No buffer overflow: (i.e., the producer process waits if the buffer is full).
3. No buffer underflow: (i.e. the consumer process waits if the buffer is empty).
4. No busy waiting.

5. No producer starvation: A process does not wait forever at Add()
   provided the buffer repeatedly becomes non-full.
6. No consumer starvation: A process does not wait forever at Remove()
   provided the buffer repeatedly becomes non-empty.

CORRECT SOLUTION

The correct solution to Producer Consumer can be implemented using Semaphore. It use three semaphores mutex, , full , empty . The semaphore are used in two different ways.

The mutex semaphore is used for mutual exclusion. It is used to guarantee that only one process at a time will be reading and writing buffer and the associated variables.

The other semaphores are used for synchronization. The Full and Empty semaphores are needed to guarantee that certain event sequences do not occur. They ensure that the producer stops running when the buffer is full and the consumer stop stops running when it is empty.

```
/*      Producer_consumer       */

public class Producer_Consumer implements Runnable
{
 static private Semaphore mutex = new Semaphore(1);
 static private Semaphore full  = new Semaphore(0);
 static private Semaphore empty = new Semaphore(20);
 static private int COUNT=0;
 private String thread_name;
 static int buffer[] = new int[20];
 static private int current=0;

 public Producer_Consumer (String name)
 {
         thread_name=name;
 }

 public void run ()
 {
  do
        {

                int randy = (int)(Math.random() * 100);
           if (((randy%2)==0) & current>0) // CONSUMER
                {
                  full.WAIT();
                  mutex.WAIT();
                  // start of critical section
                  COUNT++;
                  current--;
                  System.out.println(thread_name +" removing buffer["+current

+"]="+buffer[current] );
                  buffer[current]=-1;
                  // end of critical section
                  mutex.SIGNAL();
                  empty.SIGNAL();
                  try {
                                Thread.currentThread().sleep((int)(Math.random()          *
100));}
                  catch
                                (InterruptedException e){}
```

```
                }
                else  //PRODUCER
                {
                  empty.WAIT();
                  mutex.WAIT();
                  // start of critical section
                  COUNT++;
                  buffer[current]=randy;
                  System.out.println(thread_name +" adding buffer["+current
+"]="+buffer[current] );
                  current++;
                  // end of critical section
                  mutex.SIGNAL();
                  full.SIGNAL();
                  try {
                                Thread.currentThread().sleep((int)(Math.random()          *
100));}
                  catch
                                (InterruptedException e){}

                }
   } while (COUNT<20);
  }



  public static void main (String args [])
  {

          for(int i=0;i<20;i++) buffer[i]=-1;


     Thread T1 = new Thread(new Producer_Consumer("T1"));
     Thread T2 = new Thread(new Producer_Consumer("T2"));

          T1.start();
          T2.start();
  } }
        class Semaphore {
                  private int count;
                  public Semaphore(int n) {
                          this.count = n; }
```

```
        public synchronized void WAIT() {
                while(count == 0) {
                    try {
                      wait();
                    } catch (InterruptedException e) {
                      System.out.println("hello thread intercepted");//keep
trying
                    }
                }
                count--;
            }

            public synchronized void SIGNAL() {
                //System.out.println("inside up");
              count++;
                notify(); //alert a thread that's blocking on this semaphore

      }
        }
```

```
C:\jdk1.5.0_01\bin>javac Producer_Consumer.java

C:\jdk1.5.0_01\bin>java Producer_Consumer
T1 adding buffer[0]=14
T2 adding buffer[1]=81
T1 adding buffer[2]=13
T1 removing buffer[2]=13
T2 removing buffer[1]=81
T1 adding buffer[1]=51
T2 adding buffer[2]=3
T1 adding buffer[3]=63
T1 adding buffer[4]=83
T2 removing buffer[4]=83
T2 adding buffer[4]=23
T2 removing buffer[4]=23
T1 removing buffer[3]=63
T1 removing buffer[2]=3
T2 adding buffer[2]=43
T1 removing buffer[2]=43
T1 removing buffer[1]=51
T2 adding buffer[1]=79
T1 removing buffer[1]=79
T2 removing buffer[0]=14
```

OUTPUT

The output shows the two threads producing and consuming items of the buffer and its index.

VERIFICATION

The above solution to Producer Consumer has all the characteristics of a good solution.

- No buffer overflow
- No buffer underflow
- No over-writing of values and hence ensures mutual exclusion.

CONCLUSION

- The Ricart and Agrawala algorithm implements mutual exclusion in a computer network.
- It uses 2*(N-1) messages per critical section in a totally connected network and N messages per critical section for a ring network.
- No algorithm uses fewer messages, operates faster, and exhibits concurrent, symmetric, and distributed control.
- The algorithm is safe and live and mechanisms exist to handle node insertion, removal, and failure.
- Modifications can be made to reduce the number of messages by taking advantage of serial processing, through omitted responses.
- The sequence numbers can be stored in limited memory by keeping them as residues of a modulus that is at least twice as large as the number of nodes.
- The readers and writers problem is solved by the same algorithm with a simple modification.
- We also implemented the Producer-Consumer problem using the mutual exclusion algorithm, as these classical problems are test for any new primitive.

JAVA

Thread

Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Thread can exist in several states. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily suspends its activity.

Java defines two ways of implementing threads:

- Implement the Runnable interface.
- Extend the Thread class.

Synchronization

Java provides unique , language-level support for synchronization. Key to synchronization is the concept of monitor. Only one thread can own a monitor at a given time. All oother threads attempting to enter to enter the locked monitor will be suspended until the first thread exists the monitor. These other threads are said to be waiting for the monitor.

To enter an object's monitor , just call a method that has been modified with the synchronized keyword.

Interthread Communication

Java includes an elegant interprocess communication mechanism via the wait(), notify() methods. These methods are implemented as final methods in object , so all classes have them.

- Wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- Notify() wakes up a thread that called wait() on the same object.

BIBLIOGRAPHY

BOOKS REFERRED:

- Silberschatz, Galvin, Gagne “Operating System Concepts”, sixth edition
- Milan Milenkovic, “Operating Systems- concepts and design”, second edition
- Tanenbaum, “Operating Systems-concepts and design”, third edition
- Synchronization in Distributed Systems Workbench Ricart and Agrawala's Algorithm Paper
- survey on software Mutual Exclusion Algorithms.htm
- A simple parameterized mutual exclusion algorithm.htm
- Herbert Schildt, The Complete Reference JAVA
- Balaguruswamy, An Introduction to Object Oriented Programming with C++

WEB PAGES REFERRED:

- IPC semaphores.html
- OS.html
- OOPWeb_com - Operating Systems Lecture Notes by Martin C_ Rinard.htm
- OOPWeb_com - Concurrent Programming using Java By Stephen J_ Hartley.htm
- Ut.htm

ORIGINAL RESEARCH PAPERS REFERRED:

- 2003DistribME.pdf
- ricart.pdf
- mutex_talk.pdf