1991

The Problem of Mutual Exclusion: A New Distributed Solution

Rajeev Chawla
rajeev@rchawla.com

College of Humanities and Sciences

Virginia Commonwealth University

This is to certify that the thesis prepared by Rajeev Chawla entitled **The Problem of Mutual Exclusion - A New Distributed Solution** has been approved by his committee as satisfactory completion of the thesis requirement for the degree of Master of Science in Computer Science.

[redacted]

Lorraine M. Parker
Director of Thesis

[redacted]

James E. Ames
Committee Member

[redacted]

Thomas W. Haas
Committee Member

[redacted]

James A. Wood
Director of Graduate Studies

[redacted]

Reuben W. Farley
Chairman, Department of Mathematical Sciences

[redacted]

Elske v. P. Smith
Dean, College of Humanities and Sciences

12/9/91
Date

THE PROBLEM OF MUTUAL EXCLUSION - A NEW DISTRIBUTED SOLUTION

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
at Virginia Commonwealth University.

By
Rajeev Chawla
B.Tech., Institute of Technology, BHU, India, 1988

Director: Dr. Lorraine M. Parker
Associate Professor
Department of Computer Science

Virginia Commonwealth University
Richmond, Virginia
December, 1991

To my loving mother who is not here to see it but remained all the time with me to guide and appreciate every word of this thesis.

ACKNOWLEDGEMENTS

I wish to express my appreciation to my advisor, Dr. Lorraine Parker, for her constant help. It was she who introduced me to the problem of mutual exclusion. Her encouragement, throughout my graduate studies, gave me the confidence to keep going. She posed tough questions, answered others when I got stuck, and praised my results. Her questions and comments have immensely helped improve the quality of this thesis. It has been a great pleasure working with her.

I would also like to thank my committee members, Dr. Ames and Dr. Haas, for their constructive input into my thesis.

I owe a great deal to Dr. Wood for taking care of everything for me when I needed it the most. I would also like to thank him for continuing my graduate teaching assistantship, without which I could not have completed this work.

Thanks are also due to many friends who helped me in their own ways. Names which need a special mention are Kurt Heidelberg - thanks for being a great friend, and Katie Ambruso who showed interest in my work (in spite of being a non-CS major) and tried helping me on the proofs. I would also like to thank her for forcing me to take some time off my work. I would also like to thank Anna Harris for her administrative help and friendly advice.

Finally, I would like to thank my parents, Mr. Jag Mohan Chawla and the late Mrs. Chander Chawla, as well as my brother, Raman, for their unfaltering support, encouragement, and inspiration to keep going on in life. I owe it to them all.

Table of Contents

Page

List of Tables

List of Figures

Page

Abstract

THE PROBLEM OF MUTUAL EXCLUSION - A NEW DISTRIBUTED SOLUTION

Rajeev Chawla

Virginia Commonwealth University, 1991.

Major Director: Dr. Lorraine M. Parker

In both centralized and distributed systems, processes cooperate and compete with each other to access the system resources. Some of these resources must be used exclusively. It is then required that only one process access the shared resource at a given time. This is referred to as the problem of mutual exclusion. Several synchronization mechanisms have been proposed to solve this problem. In this thesis, an effort has been made to compile most of the existing mutual exclusion solutions for both shared memory and message-passing based systems. A new distributed algorithm, which uses a dynamic information structure, is presented to solve the problem of mutual exclusion. It is proved to be free from both deadlock and starvation. This solution is shown to be economical in terms of the number of message exchanges required per critical section execution. Procedures for recovery from both site and link failures are also given.

CHAPTER I

INTRODUCTION

The availability of inexpensive processors has made possible the construction of distributed systems and multiprocessors, that were previously economically infeasible. However, there is more to making this a reality than just hooking the hardware together. There are many problems involved in the design of such systems, such as the management of common memory and memory local to various processors, the allocation of physical and virtual resources, and concurrency protection and management. One fundamental problem that stands out from all those involved in controlling parallelism, is the synchronization of concurrently executing programs. Further, *mutual exclusion*, referred to as a "key-problem" by Dijkstra in [Dijkstra 1971], is one of the most important synchronization problems encountered in concurrent programming [Axford 1989].

1.1 Concurrent Programming :

A concurrent program specifies two or more seqential programs that may be executed concurrently as parallel processes [Andrews 1983]. It can be executed either by allowing processes to share one processor, referred to as multiprogramming, or by running each process on its own processor, referred to as multiprocessing if processors share a common memory or as distributed processing if the processes are connected by a communication network. The problem of synchronization remains independent of whether a concurrent program is executed on multiple processors or on a single multi-programmed processor. Therefore, concurrent programming is the activity of constructing a program containing multiple processes that cooperate in performing some task [Andrews 1991b], and concurrent programming abstraction is the study of interleaved execution sequences of the atomic instructions of sequential processes [Ben-Ari 1990]. Since no assumptions can be made about the execution rates of concurrently executing processes, one process could complete hundreds of instructions before any other process executes one instruction. The only assumption made is that a process does not deliberately halt - it keeps on executing at a positive rate. This is called the *finite progress assumption* [Andrews 1983].

Since arbitrary interleavings of process instructions are possible, a concurrent program behaves in a non-deterministic fashion. Consider for example, a change-giving machine which accepts $1 and $5 bills and offers its customers change in any of the combinations of nickels, dimes, and quarters. The customer cannot predict the combination of denominations he is going to get, and the machine may behave differently at different times for the same input, that is it may change a $1 bill into four quarters one time and ten dimes at a later time. The machine is said to behave in an arbitrary or non-deterministic fashion.

For a concurrent program to be correct, it is required to be correct under all interleavings of execution sequences and this leads to extensive case analysis as the number of interleavings that must be considered grows exponentially with the size of the component sequential processes. For the change-giving machine example, it has to be ensured that the machine offers exact change irrespective of whatever combination of denominations is given to the customer. The exact time of execution of instructions is ignored as it has no relevance to program correctness, except in cases of time-critical (hard real-time) systems [Faulk 1988].

1.2 Process Interactions :

In concurrent programming, problems start appearing when two or more processes interact with each other. These interactions require simultaneous participation of both the processes involved. For example, a chocolate can be extracted from a vending machine only when its customer wants it and only when the vending machine is prepared to give it [Hoare 1985]. For any particular application, there are two kinds of interactions - cooperative and competitive. Cooperative processes are directly or indirectly aware of each other's existence. On the other hand, competing processes are unaware of each other - any interaction between them is indirect. The resolution of a competitive situation may require creation of cooperating processes and conversely, cooperating processes could compete with one another for resources. The following examples from [Andre 1985] clarify this point -

Example 1: Consider a set of processes {P} which share (compete for) a single printer. Access to this printer is gained by cooperation between the calling processes of {P}, on one hand, and a single print-server process, on the other.

Example 2: Suppose there is a set of processes which cooperate in pairs to produce and print certain values. Since the buffers for all producer-printer pairs must reside in a store with limited capacity, the process pairs will compete for storage locations indirectly by means of cooperating producer and printer processes of each pair. This competition can be resolved by cooperation between the processes which allocate and retrieve the buffer space.

In order to cooperate, concurrently executing processes must communicate and synchronize. Communication allows execution of one process to influence execution of another. Inter-process communication is based on the use of shared variables or on message passing [Andrews 1983].

1.3 Definition of Synchronization :

A process is assumed to be executing in discrete steps. At each step, there is an "event", which can be either local to the process in which it occurs and not perceived by the rest of the system, or can have some significance on the whole system, in which case it is relevant to the general problem of synchronization. According to [Andre 1985], *synchronization* consists of controlling the evolution of processes, and therefore the occurrence of events, as a function of the past history of the system. Simply, synchronization can be thought as a set of constraints on the ordering of events [Andrews 1983] - this may involve delaying execution of a process to satisfy these constraints. In example 2 above, a printer process cannot print a value unless it is produced.

Conceptually, there are two major forms of synchronization in a system of sequential processes that run concurrently - one on behalf of accessing shared data and one on behalf of communication [Habermann 1972].

1.3.1 Condition Synchronization :

Communication may lead to a situation where one process is ready to process input which is yet to be produced by another process. In that case, the processes must be synchronized such that the consuming process cannot start processing the input before the producer has produced it. This is referred to as *condition synchronization* [Andrews 1983].

1.3.2 Mutual Exclusion :

In certain circumstances, it is necessary to ensure that portions of two concurrent processes do not run concurrently. These portions are called *critical sections*. For example, if two or more processes share a common resource (memory, peripheral, CPU, clock, etc.), mutual exclusion must be enforced on the sections of these processes which access the shared resource to secure the integrity of the shared resource.

Consider two processes, both of which put (enqueue) items onto a shared queue without bothering about a dequeue operation for this example. Suppose the queue is implemented as an array with variable name “queue”, and an index, “tail”, to the last item put into the queue. The code in two processes could look like that given in Figure 1.1

Since nothing can be assumed about the relative speed of different processors, suppose processor1 executes *tail := tail + 1*, and processor2 executes *tail := tail + 1* followed immediately by *queue[tail] := item2*; then, processor1 executes *queue[tail] := item1*. This order of execution shows that nothing is put into queue[tail+1], and item1 is put into queue[tail+2] and item2 gets lost.

The concurrent execution in the above example led to serious corruption of the queue. Therefore, it is required for all update operations on the shared variables to be mutually exclusive in time.

The design of a mutual exclusion algorithm consists of defining the acquisition and release protocols used to coordinate entry into the critical section - acquisition protocol (entry code) is executed before entering the critical section, and release protocol (exit

```
Shared :    queue   :  Array[0..queuesize] of anytype ;
            tail    :  integer ;

Local :     item1 , item2   :  anytype ;

 Process 1                              Process 2

 ..........                             ..........
 tail := tail + 1 ;                     tail := tail + 1 ;
 queue[tail] := item1 ;                 queue[tail] := item2 ;
 ..........                             ..........
```

Figure 1.1 - Enqueue operation in two processes

code) is executed on leaving the critical section. Thus, all mutual exclusion algorithms can be depicted as -

```
acquisition protocol
< CRITICAL SECTION >
release protocol
```

The acquisition and release protocols ensure that the critical section is used by only one process at a time and any other process trying to enter the critical section waits. In addition, the protocols can play a scheduling role in determining which of several contending processes is allowed to proceed.

Enforcing mutual exclusion is not an easy task. Dijkstra was the first to show whether or not processes could be synchronized with just the standard operators of an ordinary programming language [Dijkstra 1965]. He states that this is the most difficult program he has ever written [Dijkstra 1971].

1.4 Properties of Mutual Exclusion Algorithms :

There are a number of pitfalls to avoid while writing the solution (program) to provide synchronization. The first pitfall is *deadlock*. Consider several processes all attempting to enter their critical section to use the shared resource. As at most one process may be in the critical section, one solution would be to let none of them in. An analogy [Raynal 1986] would be - when several people meet before a doorway (the resource) and suppose the protocol is, if I am alone, I go through; otherwise, I let others go first. If several people with this protocol arrive simultaneously at the doorway, they will all wait, blocking each other's access. This is deadlock. [Silberschatz 1988] defines deadlock as a state of processes where two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes. Although it does provide mutual exclusion, it leads to a situation in which there is no useful activity by any of the processes (in other words, the system is 'hung'); it must therefore be avoided.

The second pitfall to avoid is a situation whereby a process is postponed infinitely in entering its critical section. Consider the case where processor P1 is in the critical section and P2 and P3 are delayed in their acquisition protocol. Once P1 executes its release protocol, the first step is to end the delay of P2 or P3. Assume that P3 now enters the critical section and that P1 executes its acquisition protocol once more. Consider the situation when P1 again enters its critical section once P3 comes out of it. If this keeps on happening, the processes may end up behaving in such a way that P2 is indefinitely delayed in its acquisition protocol and will never get to enter the critical section. This situation is called *starvation*.

It is important to note that in case of n processes competing for one resource, deadlock brings the whole system (all the processes) to a "standstill" state, whereas starvation does that to individual processes.

To rule out starvation, the mutual exclusion solution should be *"fair"*. [Francez 1986] defines fairness as a restriction on some infinite behavior according to eventual occurrence of some events. On the basis of eventuality, there are three main subclasses -

- Unconditional fairness,
- Weak fairness, and
- Strong fairness.

Unconditional fairness implies that for each behavior each event occurs infinitely often without any further qualification [Francez 1986]. For example, consider multi-programmed non-communicating concurrency - here, n processes, totally independent of each other, conceptually are executed in parallel but use one common processor. An event is the execution of an atomic step in one process. In this case, unconditional fairness means that along an infinite execution each process is allocated processor time infinitely many times. But, nothing is implied about the length of the interval between consecutive processor time allocations to any given process, or about the length of time the processor is allocated to any given process.

According to weak fairness, an event will not be indefinitely postponed from occurring provided that it remains continuously enabled from some time instant until it actually occurs. On the other hand, strong fairness guarantees eventual occurrence under the condition of being infinitely often enabled, but not necessarily continuously. Implementation of a strong fairness policy is tougher, but it is a preferred choice in many cases. For example, consider a process in its acquisition protocol waiting for a condition. It must repeatedly (infinitely often) test the shared variable to gain entry into the critical section. This method of delaying a process is called *busy-waiting* and the process is said to be *spinning* on the shared variables, called *spin-locks*. A strongly fair mutual exclusion solution with busy-waiting will guarantee eventual entry to the critical section for each attempting process.

These concepts of fairness are not very practical because they depend on 'eventually' and 'infinitely often'. A practical approach could be basing fairness on the "order of arrival" of the requesting processes. Linear and FIFO (first-in-first-out) fairness fall in this category. These are a by-product of queue implementation. For real-time applications, a "bounded-delay" fairness policy, in which there exists a bound (hopefully small) on the length of the interval between consecutive occurrences of the same event, is preferred.

[Dijkstra 1965] gives a minimum number of properties that all algorithms implementing mutual exclusion must have. These are -

- Algorithm should allow only one process in its critical section at any one point of time.
- Algorithm should be deadlock-free, that is, if several processes are waiting to enter a critical section while no process is actually in its critical section, one of them must enter it within a finite time.
- There should be complete independence between those parts of the algorithm which are involved in access conflicts and those parts that are not.
- Algorithm should treat all processes in the same fashion, that is, it should not have any privileged process.

Dijkstra did not have ‘starvation-free’ property in his list of minimum properties, but it is a desirable property to include.

1.5 Performance Mcasuremcnt :

[Stone 1989] introduces the performance notion of SYPS (SYnchronizations Per Second), measured in MSYPS (Mega SYPS). The number of serial sections executed sequentially in one second gives the MSYPS rate. As a general rule, adding more processors increases the MIPS (Mega Instructions Per Second) rate of the system. But if processes need to have a lot of synchronization operations among themselves, increased MIPS does not effect the speedup at all; the MSYPS rate then determines the increase in the speedup. Thus, throughput is limited both by the MIPS and MSYPS capacity of the system.

1.6 Concurrent Program Correctness :

Programs, especially concurrent programs, are often described informally. A process can be studied without getting into formal notation and proof system. There are two approaches to arguing about the correctness of a program -- **Operational Reasoning** and **Non-Operational (Formal/Axiomatic) Reasoning.** Operational reasoning involves arguments about the unfolding computations of a program, whereas non-operational reasoning focuses on static aspects (such as invariants) of the program [Chandy 1988]. Both approaches are useful and have advantages over each other. Since arbitrary interleavings of process execution sequences are allowed in concurrent processing and for a concurrent program to be correct, it has to be correct under all interleavings, it is hard to convince skeptics about the correctness of concurrent programs by using operational arguments. A common mistake in operational arguments is forgetting to consider certain sequences of events that could occur. Also, operational arguments tend to be longer. Nevertheless, creation of an algorithm is often based on operational reasoning and for this reason, the operational technique is used extensively.

Currently, there are several methodologies for verifying concurrent programs. These concentrate on the concurrency problems and sometimes leave the sequential parts of the program to be analyzed using other methods. Methodologies like **Petri Nets** [Peterson 1981b], **CSP** [Hoare 1985], and **UNITY** [Chandy 1988] require the program to be modeled using their own specific synchronization and communication primitives. These primitives have simple and mathematical definitions, and therefore, permit a rigorous mathematical analysis of the programs written using these primitives. But, it is often inconvenient and tedious as these primitives are not the same as the primitives used in many common programming languages.

Whatever approach is used, all the properties (mutual exclusion, deadlock-free, starvation-free, fairness, etc.) of mutual exclusion protocols discussed in the algorithm need to be satisfied.

1.7 Outline of the Thesis :

This thesis is organized into six chapters. In Chapter II different hardware and low-level mechanisms available for implementing synchronization are discussed. In Chapters III and IV an outline of the existing software solutions for centralized and distributed systems is given. In Chapter V, a new algorithm for providing distributed mutual exclusion is submitted. In Chapter VI, a summary of the thesis and future work problems are given.

CHAPTER II

SHARED MEMORY LOW-LEVEL SOLUTIONS

2.1 Concept of Indivisible Instructions :

It is conventional for computer hardware to be designed to permit interrupts to occur only between instructions. Normally, an instruction may not be interrupted in the middle of its execution. An interrupt request that arrives in the middle of an instruction is not lost but merely made to wait. This has an important effect - a single machine instruction is guaranteed to be indivisible; once execution of it has begun, no other process can interrupt until it has finished.

On multiprocessor machines in which several processors share a common memory, the normal indivisibility of machine instructions does not provide mutual exclusion between different processors. Many instructions involve several memory accesses each. If another processor shares access to the same memory, there is normally nothing to prevent it from accessing memory between accesses by the first processor. Of course, a processor will have no way of knowing if consecutive memory accesses by another processor are part of one instruction or several instructions.

The memory unit enforces mutual exclusion on each individual memory access. This means that there is no risk that a memory location can ever be found in an intermediate state, it must either contain its value before the write access or its value after it. Thus, memory read and write are indivisible operations. But some additional mechanisms are needed to enforce mutual exclusion in a multiprocessing system with shared memory.

The idea of disabling interrupts during the execution of critical section also is ineffective on multiprocessor systems. Disabling interrupts only prevents other processes on the same processor from running concurrently; it has no effect on processes running on different processors.

Multiprocessor machines provide a special ***memory lock*** instruction which is treated as a prefix to the instruction immediately following it, and causes a memory lock to be applied for the duration of that instruction. This means that no other processors or devices are permitted access to the shared memory during execution of the locked instruction.

Memory locked instructions are thus effectively indivisible (i.e. mutually exclusive) on multiprocessor systems, just as all machine instructions are indivisible on a uniprocessor system.

2.1.1 Memory Locks in Intel 8086 series :

The 8086 includes a memory lock prefix instruction which can be used to prefix any other instruction and cause a memory lock to be applied for the duration of that instruction. This can have disastrous effects if misused and it seems the ***LOCK*** instruction was added to the instruction set hurriedly at the eleventh hour, to support concurrency [Axford 1989]. The 8086 instruction set includes string instructions which will operate on strings of arbitrary length. These instructions can take a very long time to execute and involve many memory accesses (depending on the length of string). If memory lock is applied to such instructions, other devices are locked out of memory for relatively long periods of time. This can be a disaster for a fast disk-controller, which has to read data from a disk rotating at a fixed speed, and store that data in memory at the rate at which it comes off disk. There are typically only a few micro-seconds in which to write each data word to memory. If access to memory is not obtained within this time, the data is lost and complete disk transfer has to be aborted.

Probably, this anomaly is because 8086 is a single user machine. It does not provide memory protection, nor are user processes prevented from using I/O instructions or other instructions which the operating system may prefer to keep for its own use alone in other environments.

When the 80286 was designed, the philosophy changed. This processor was designed to be a multi-user machine with full memory protection and other features to prevent processes from interfering with each other in uncontrolled ways. Of course, the unconstrained use of memory lock cannot be permitted for ordinary user processes. Instead, unconstrained use of memory lock is permitted for the operating system only, and all other programs are prevented from using it - the operating system decides whether or not a user process can use memory lock. Nevertheless, memory lock is automatically implemented on all exchange instructions, whether requested or not, so that all processes can obtain some mutual exclusion facilities in a multiprocessor system. [Liu 1986; Intel 80286]

This solution was fine, until the 386^{TM} was designed. The 386^{TM} includes support for paging. This made it impractical to implement memory lock on some instructions, even though the use of memory lock was confined to the operating system and privileged processes only. The culprit is again the string instruction. Suppose the string crosses a page boundary into a non-resident page. It is impossible to maintain the memory lock while the page is recovered from disk (swap space), as the disk-controller cannot access memory while the lock is on. But if the lock is released, this destroys the mutual exclusion which was the whole point of using the lock in first place. Therefore, the 386^{TM} designers abandoned the 80286 approach. Instead, the 386^{TM} adopts an approach that totally prohibits any process from using memory lock on specified types of instructions, which include string instructions. Having restricted the use of memory lock to only those instructions for which it is always safe, i.e., those whose execution time is always fairly short, there is no longer any need to prevent ordinary user processes from using it. So, on the 386^{TM} machine, memory lock is no longer regarded as a privileged instruction. Table 2.1 lists the instructions which can use *LOCK* instruction as a prefix. [Liu 1986; Intel 386^{TM}]

The 486^{TM} processor also supports paging and therefore, like 386^{TM}, the usage of *LOCK* instruction is restricted to the instructions which have small execution time. Table 2.2 lists the

ADC, ADD, AND, BT	mem,reg/immediate
BTS, BTR, BTC, OR	mem,reg/immediate
SBB, SUB, XOR	mem,reg/immediate
XCHG	reg,reg
XCHG	mem,reg
DEC, INC, NEG, NOT	mem

Table 2.1 - Valid 386^{TM} Instructions with the Lock Prefix

Bit test and Change Instructions	BTS, BTR, BTC
Exchange Instructions	XCHG, XADD, CMPXCHG
1 operand arithmetic and logical instructions	INC, DEC, NOT, NEG
2 operand arithmetic and logical instructions	ADD, ADC, SUB, SBB, AND, OR, XOR

Table 2.2 - Valid 486^{TM} Instructions with the Lock Prefix

486^{TM} CPU instructions which can use the *lock* prefix. An invalid-opcode exception results from using the *lock* prefix before any other instruction, or with these instructions when no write operation is made to memory (that is, when the destination operand is in a register).

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area, if execution is going on in 8086/80286 configuration.

Locked cycles are implemented in hardware with the *LOCK#* pin. When *LOCK#* is active, the processor is performing a **read-modify-write** operation and the external bus is not relinquished until the cycle is complete. Multiple reads or writes can be locked. The 486^{TM} also has a *PLOCK#* pin which indicates that the current bus cycle and the following one should be treated as an atomic transfer. This can be used to generate atomic reads and writes of 64-bit operands.

The 486^{TM} processor always asserts *LOCK#* during an *XCHG* (exchange) instruction which references memory, even if the *lock* prefix is not used. It also provides two instructions *XADD* (Exchange and Add) and *CMPXCHG* (Compare and Exchange), which if prefixed with the *LOCK* instruction, can be used (as explained in the next sections) to implement mutual exclusion protocols. [Intel 486^{TM}]

2.1.2 Memory Locks in Pyramid System :

The Pyramid system supports paging. So, like 386^{TM}, the string instructions provided for commercial applications do not allow memory to be locked. The Pyramid Reference Manual makes it clear by adding a note in all the string instructions, that these instructions are interruptible.

2.2 Synchronization with Indivisible Instructions :

Most modern computers provide a number of special instructions that are particularly useful for concurrent programming because they are guaranteed to be indivisible, i.e., mutually exclusive with other instructions. The most common are listed in Table 2.3.

The common factor linking all these instructions and distinguishing from all others, is the fact that they carry out two

- EXCHANGE INSTRUCTION
- TEST and SET INSTRUCTION
- LOCK INSTRUCTION
- INCREMENT AND DECREMENT INSTRUCTION
- COMPARE and SWAP INSTRUCTION
- FETCH and ADD INSTRUCTION

Table 2.3 - Atomic Instructions for Synchronization

actions atomically - reading and writing or reading and testing of a single memory location with in one instruction cycle. The classical form of READ/MODIFY/WRITE is a key characteristic of synchronizing instructions. [Stone 1989; Raynal 1986]

2.2.1 Exchange Instruction : [Raynal 1986; Axford 1989]

The instruction ***exchange(r,m)*** exchanges the contents of register *r* with those of memory location *m*. During execution of this instruction, access to *m* is blocked for any instruction using *m*.

The mutual exclusion protocol (in Figure 2.1) uses a variable shared by all processes - memory location *bolt* initialized to 1; it may take values 0 or 1. Each process P_i uses a local variable *key* (a processor register) initialized to 0; it also only takes values 0 or 1. The protocol for process P_i is as follows -

A process will only be allowed to enter its critical section if it finds *bolt* set to 1. It will then exclude all other processes from the critical section by setting *bolt* to 0. It releases the critical section by setting *bolt* to 1, thereby allowing a waiting process to

```
var      bolt : shared integer; {initially 1}
         key  : integer; {initially 0}
repeat   exchange($key_i$ , bolt);
until    $key_i = 1$;
< CRITICAL SECTION >
exchange($key_i$ , bolt);
```

Figure 2.1 - Mutual Exclusion Using Exchange Instruction

enter the critical section.

Note that only process P_i in its critical section satisfies $key_i = 1$, and that the following relation on variables *key* and *bolt* is true at all times-

$$\sum_i key_i + bolt = 1,$$

which is the invariant for this solution to the problem - it ensures, given the range of values (integers 0 & 1), that not more than one process is in its critical section; while if bolt equals 1, no process is in its critical section.

Both 386^{TM} and 486^{TM} microprocessors provide this instruction as an atomic instruction.

2.2.2 Test and Set Instruction : [Stone 1989; Raynal 1986]

The instruction ***testset(m)*** (Figure 2.2) carries out a series of actions atomically - it tests the value of variable *m*; if the value is 0, it replaces it by 1, otherwise it makes no change to the value of *m*. It returns value of *m* in condition code in both cases.

```
Definition :  TestSet(mem_address);
{The [ ] operator fetches the contents of the specified
  memory address location.}
begin   condition_code := [mem_address];
        if condition_code = 0 then [mem_address] := 1;
end;    {Return  Condition_code}
```

Figure 2.2 - Definition of Test-and-Set Instruction

The mutual exclusion protocol implemented using this instruction requires a shared memory location initialized to 0. The protocol for each process P_i is given in Figure 2.3.

The only process that can be in its critical section is the one that found *bolt* set to 0.

2.2.3 Lock Instruction : [Raynal 1986]

The definition of this instruction is very similar to that of *testset*. Here, however, the wait loop is an integral part of the instruction itself.

The behavior of atomic lock and unlock instruction can be best described as given in Figure 2.4.

```
var     bolt : shared integer; {initially 0}
repeat  condition_code := Testset(bolt);
until   condition_code := 0;
<  CRITICAL  SECTION  >
bolt  :=  0;
```

Figure 2.3 - Mutual Exclusion Protocol Using TestSet

```
Definition :  lock(m);
begin
   while m = 1 do;
   m ← 1;
end;
Definition :  unlock(m);
   m ← 0;
```

Figure 2.4 - Definition of "Lock"

A separate "unlock" instruction need not be specially provided as its effect can be obtained simply by *m* ← *0*; each memory access being mutually exclusive.

On a uniprocessor machine, the lock instruction will lead to deadlock if interrupts are not permitted during execution of the instruction. If *m* is initially 1, the only way this instruction can terminate is for some other process to reset *m* to 0. On a uniprocessor, the only way to start another process is by an interrupt, hence interrupts must be permitted during this instruction, but only after the first statement, i.e., inside the wait loop. This instruction is intended primarily for use in a multiprocessing environment.

For a shared variable *bolt* initialized to 0, the mutual exclusion protocol is given in Figure 2.5.

The lock instruction is sometimes called *test-and-switch-branch (TSB)*.

If several processors are all waiting for the same *m*, only one will be allowed to proceed when *m* returns to 0. But which one is

```
var bolt : shared integer; {initially 0}

lock(bolt);

< CRITICAL SECTION >

unlock(bolt);
```

Figure 2.5 - Mutual Exclusion Protocol using "Lock"

allowed depends upon the hardware implementation of the *lock* instruction - it may be completely unpredictable in some machines while a fixed priority order may operate in others.

2.2.4 <u>Increment and Decrement Instructions</u> : [Stone 1989;Raynal 1986]

The effect of increment and decrement instructions, *increment(r,m)* and *decrement(r,m)*, is to increment or decrement, respectively, the contents of the memory location *m* by 1 and to load the result into register *r*. Of course, this instruction is executed in one cycle (i.e., is uninterruptible) to make it mutually exclusive.

The exclusion protocol, given in Figure 2.6, for process P_i is implemented using the *decrement(r,m)* instruction alone (as shown). Here, *bolt* is initialized to 1. Mutual exclusion can be implemented in an

```
var bolt : shared integer; {initially 1}
repeat  decrement(key,bolt);
until    key = 0;
< CRITICAL  SECTION >
bolt  ←  1;
```

Figure 2.6 - Mutual Exclusion Protocol using "Decrement"

entirely analogous way using instruction *increment(key,bolt)*, but with variable *bolt* initialized to -1.

In practice, the use of either of these instructions on machines implementing them would have one major disadvantage - if a process remains in possession of the critical section for a long time, while others are trying to access it, variable *bolt* will grow (decrease) indefinitely, when using *increment* (*decrement*) instruction. Depending on the size of the memory location, this could lead to major problems causing mutual exclusion to fail because of one of these two reasons -

- overflow causing memory failure,
- variable returns to value 1(-1) in case of *decrement*(*increment*) instruction[1].

The ICL 2900 series computers provide two similar indivisible instructions, called "dect" and "tinc" [Keedy 1985].

These two instructions together are more powerful than *Test-and-Set* instruction and help in reducing the number of instructions required for synchronization.(See §2.4.1)

[1] This is more likely because of the way numbers are coded in memory words. For example, in case of 16-bits word size, $(7FFF)_{16}$ is maximum positive number and $(8000)_{16}$ is the maximum negative number. Therefore, adding 1 to FFFF (= -1) would yield a 0 and so we can cycle back to value 1. This is a serious matter and gives problems in case of theoretical verification of programs. Verification of program on one machine may not be correct on another because of a different word-size.

2.2.5 Compare and Swap Instruction : [Stone 1989; Hwang 1985]

The *compare-and-swap* instruction (Figure 2.7) uses two registers, one to hold an old value of the shared datum, and one to hold a new value. The advantage of using this instruction is that it computes the new value of the shared datum without locking it; it refetches the shared datum, checks to see if its value is unchanged, and if so, performs the update. If the value has changed, the current value is loaded into the register that holds the old value. This instruction is available on the IBM 370/168.

The *compare-and-swap* instruction is more powerful than the rest of instructions shown before. But, it is useful in a limited number of

```
Definition: compare_and_swap(address,reg_old_val,reg_new_val);
{The [ ] operator fetches the contents of the specified memory address.}
temp := [address];
NOTE : condition-code is returned and it can be used to check
          if the update took place.

if temp = reg_old_val then
begin [address] := reg_new_val;
      condition_code := 1;
end
else begin reg_old_val := temp;
           condition_code := 0
end;
end { of definition };
```

Figure 2.7 - Definition of "Compare and Swap"

important circumstances only, including the queueing and dequeueing of tasks.

The mutual exclusion protocol using *compare-and-swap* instruction for enqueueing an item, assuming no dequeue operations are allowed, is given in Figure 2.8a. A general mutual exclusion solution using compare-and-swap may be obtained by making compare-and-swap behave like the exchange instruction. This is given in Figure 2.8b.

```
[item_address].Link := nil;    { Initialize items for insertion
                                  at end of queue}
Reg_tail := tail;              { Read tail to a register }
LOOP:  compare_and_swap(tail,reg_tail,item_address);
if condition_code = 0 then goto LOOP;
{ Loop back on failure of compare-and-swap }
[reg_tail].Link := item;
```

Figure 2.8 a - Mutual Exclusion Protocol Using "Compare and Swap" for enqueueing an item

```
Initial Values :
[address] := 1;
reg_old_val := 0;
reg_new_val := 1;
Protocol for $P_i$ :
Repeat
compare_and_swap(address,reg_old_val,reg_new_val);
Until condition_code = 0;
< CRITICAL SECTION >
reg_old_val := 0;
```

Figure 2.8 b - General Mutual Exclusion Protocol Using Compare_and_Swap

The reason *compare-and-swap* is more powerful is that the shared datum is locked at the beginning of the instruction, updated during the instruction, and unlocked at the end. This is in contrast to the prior instructions, which lock the critical section and release the critical section by manipulating a shared variable; the critical section remains locked for a long time till its execution. The *compare-and-swap* instruction, therefore, produces the maximum possible MSYPS rate by reducing locked regions of a program to a single instruction.

2.2.6 Fetch and Add Instruction : [Stone 1989; Gottlieb 1987]

Instruction fetch-and-add was proposed by Gottlieb for MIMD (Multiple Instruction Multiple Data) shared memory machines using a message switching network with the geometry of the Omega-network [Lawrie 1975]. The NYU Ultracomputer [Gottlieb 1987] provides a primitive *fetch-and-add* that permits every PE (processing element) to read and write a shared memory location in one cycle. In particular, simultaneous reads and writes directed at the same memory location are done in a single cycle, and therefore, this primitive allows synchronization of multiple processes in a parallel manner.

It is based on the *serialization principle* [Eswaran 1976], which states that the effect of simultaneous actions by the PEs is as if the actions had occurred in some unspecified serial order. Thus, for example, a load simultaneous with two stores directed at the same memory location will return either the original value or one of the two stored values, possibly different from the value that the cell finally comes to contain.

Fetch-and-add is essentially an indivisible add to memory; its format is F&A(V,e), where V is an integer variable and e is an integer expression. This indivisible operation is defined to return the old value of V and to replace V by the sum V+e. If V is a shared variable and many *fetch-and-add* operations simultaneously address V, the effect of these operations is exactly what it would be if they occurred in some (unspecified) serial order, i.e., V is modified by the appropriate total increment and each operation yields the intermediate value of V corresponding to its position in this order. The following example illustrates the semantics of *fetch-and-add*, assuming V is a shared variable. If PE_i (i^{th} processing element) executes

$ANS_i \leftarrow F\&A(V,e_i)$, and if PE_j simultaneously executes

$ANS_j \leftarrow F\&A(V,e_j)$, and if V is not simultaneously updated by yet another processor, then either

$$\boxed{\begin{array}{l} ANS_i \leftarrow V \\ ANS_j \leftarrow V + e_i \end{array}} \quad \underline{\textbf{OR}} \quad \boxed{\begin{array}{l} ANS_i \leftarrow V + e_j \\ ANS_j \leftarrow V \end{array}}$$

and, in either case, the value of V becomes $V + e_i + e_j$.

An example which demonstrates the concurrent updation of shared memory is the concurrent execution of F&A(i,1) by several PEs. Here, ‘i’ is a shared variable used to index into a shared array. Each PE obtains an index to a distinct array element, although one cannot predict which element will be assigned to which PE, and 'i' receives the appropriate total increment.

[Gottlieb 1983] showed that *fetch-and-add* can be generalized to a fetch-and-ϕ operation that fetches the value in V and replaces it with ϕ(V,e). Of course, defining ϕ(a,b)=a+b gives fetch-and-add. It is easy to see that defining ϕ to be a boolean OR function gives *test-and-set* and defining ϕ to be a second value projection function π_2 (i.e., π_2(a,b)=b) gives *exchange* (*swap*). Therefore,

TestSet(V)	is equivalent to	F&OR(V,TRUE), and
Exchange(L,V)	is equivalent to	L ← F& π_2(V,L).

Thus, use of *fetch-and-add operation* allows many processes to perform in a completely parallel manner. No locking and unlocking is required, nor is a retry test and loop required as with *compare-and-swap*.

For multiprocessor systems with fewer processors, *compare-and-swap* is found to be better approach, whereas *fetch-and-add* is preferable as the number of processors increases, since it can execute all the requests for shared memory access simultaneously. But whether or not *fetch-and-add* is cost-effective, is still a matter of research interest [Stone 1989] - its implementation cost is high, and is limited to simultaneous access of the same shared variable by all contending processes.

2.3 PERFORMANCE CONSIDERATIONS :

The MIPS rate of a system can be increased by adding more processors, but MSYPS may not increase at all. The exclusive access requirement limits the performance of most multiprocessor architectures. When access to a shared variable is saturated, no additional speed improvement is possible no matter how many more processors are added to the system. Actually, the computation time may increase by adding more processors, as more processors will be active contending to access the shared data. The MSYPS bottleneck is one of the sources of performance degradation.

The instructions *exchange, test-and-set, lock, increment-and-decrement* are another source of performance degradation - processes attempting to enter critical sections are busy accessing and testing global variables. This is called *busy wait* or *spin-lock.* When a processor is spinning, it actively consumes memory bandwidth that might otherwise have been used more constructively. If the spinning period is too long, a processor is not effectively utilized during that period and therefore ends up wasting lots of computer cycles. Moreover, when many processors are spinning, the contention causes additional cycles of delay while a process is attempting to release the critical section.

A number of methods have been proposed to reduce degradation due to spin-locks. One of the methods is aimed at reducing the request rate to memory and, hence, the degree of memory conflicts. This is accomplished by delaying retesting of the global variable for an interval T. Thus, for example, the mutual exclusion protocol using test-and-set instruction canbe modified as given in Figure2.9.

```
var     bolt : shared integer; {initially 0}
begin
    condition_code $\leftarrow$ testset(bolt);
    while condition_code $\neq$ 0 do
     begin  PAUSE(T);
            condition_code $\leftarrow$ testset(bolt);
     end;
end;
```

Figure 2.9 - Modification of protocol in Figure 2.3

Another method is directed at making available cycles to do useful work. This can be accomplished by suspending the blocked process and enqueueing its status on a queue associated with a global variable; and then reassigning the processor to another ready-to-run process in its local memory. When the processor is signaled that the lock has been allocated, it resumes execution of the waiting process. The resumption would be immediate if the process is not swapped out; but in real-time systems a process may need to be resumed regardless of whether or not it is swapped out. Though suspending and resuming processes appears to be very efficient, the overheads involved with enqueue and dequeue operations would be very high and may be greater in cost than the cost of the cycles lost in spin-lock. Moreover, enqueueing a task means that a processor has to access and update a shared queue pointer. This access itself involves a lock/unlock of some kind. If this lock is not granted, the problem of enqueueing a task at one queue to enqueue it at another queue is encountered, and this could repeat *ad infinitum*. Therefore, this chain of events has to be broken by forcing a lock to be implemented by means of a spin-lock. Now, this is the place where the instruction *compare-and-swap* comes very handy; the shared queue can be accessed and updated using *compare-and-swap* as queue would be locked for just one cycle, i.e., the time for execution of *compare-and-swap* instruction. This implies that, for efficient use, there ought to be at least two primitives built-in the processor - one being *compare-and-swap* and another could be any from *test-and-set/lock/exchange/increment & decrement*.

Blocking a task could be worthwhile, but only if the size of the critical section is very large; otherwise it should be sufficient to delay the retesting of global variable by putting a PAUSE statement to avoid memory contention.

In terms of performance, task enqueueing tends to increase available MIPS by reassigning the idle processors to other useful work; whereas spin-locks tend tσ decrease MIPS by wasting useful machine cycles. Task enqueueing increases the number and length of critical sections protected by locks. By increasing the number of critical sections, the MSYPS demand is increased, and hence the overall effect of task enqueueing is to decrease the maximum potential MSYPS rate.

Another important issue is implementation of *UNLOCK*. The unlocking process has to compete with the (N-1) processes spinning on the shared variable, and the result of this may be a delay of time proportional to N. Giving priority to a WRITE request over a READ/MODIFY/WRITE request can avoid this problem; but it must be done carefully as giving priority to writers may postpone readers forever (starvation).

The distribution of locks in memory is also an important factor in the performance of concurrent processes accessing lockable resources. If all locks are stored in one memory module, the contention for these locks can become excessive.

2.4 SEMAPHORES :

Dijkstra was one of the first to appreciate the difficulties of using low-level mechanisms for process synchronization, and this prompted his development of semaphores [Dijkstra 1968; Dijkstra 1971]. Semaphores can be implemented using statements of the *testset* type, and are generally offered as fundamental tools of system kernels.

A semaphore S is a non-negative integer variable that can be handled only by two primitives[2] P and V (defined in Figure 2.10), besides initialization.

[2] P is the first letter of the Dutch word "passeren", which means "to pass"; V is the first letter of "vrygeven", the Dutch word for "to release". Reflecting on the definitions of P and V, Dijkstra and his group observed P might better stand for "prolagen" formed from the Dutch words "proberen" (meaning "to try") and "verlagen" (meaning "to decrease") and V for the Dutch word "verhogen" meaning "to increase".

The notation *wait* and *signal* is also used for P and V respectively [Habermann 1972]. By definition, both P and V primitives are atomic, i.e., only a single primitive may be executed on any one semaphore at any one time.

```
P(S) :  $S \leftarrow S - 1$;
        if $S \leq 0$ then wait in a queue associated with S endif;
V(S) :  $S \leftarrow S + 1$;
        if $S \leq 0$ then unblock one of the waiting processes endif;
```

Figure 2.10 - Definition of P and V operations

In this definition of semaphores, processes that are blocked within a P operation on a semaphore variable S are distinguished from processes that are about to execute a P(S) but have not yet become blocked. This distinction is important as the execution of a V(S) will cause a blocked process to be selected in preference to a process that is not blocked. However, all blocked processes are treated equally as far as being selected is concerned - no effort is made to distinguish processes that have been blocked for a short length of time from those that have been blocked for a longer period. The group of blocked processes at any instant of time can, therefore be modeled as a *set*, from which a V operation chooses at random a process to be signaled. Stark calls semaphores with this type of blocking discipline *blocked-set semaphores*. He also defines two more types of semaphores - *blocked-queue semaphores* and *weak semaphores*. *Blocked-queue* *semaphores* are like *blocked-set semaphores* except that the group of blocked processes is maintained as a FIFO queue, instead of a set. In case of a *weak semaphore*, a process attempting to perform a P operation on a semaphore variable S executes a busy-waiting loop in which the value of S is continually tested. As soon as S is discovered to have a value greater than zero, it is decremented; the decrement and immediately preceding test are performed as one indivisible step. A V operation simply increments S in an indivisible step. A *weak semaphore* is also called a *busy-wait semaphore*. [Stark 1982]

Each one of these, namely *weak*, *blocked-set*, and *blocked-queue*, semaphores has a different starvation property. These properties can easily be deduced from their definitions, and are given below -

- For a weak-semaphore, starvation is possible.
- For a blocked-set semaphore, starvation is possible, if the number of processes contending for the critical section is greater than two.
- For a blocked-queue semaphore, starvation is impossible.

[Morris 1978] showed that starvation-free mutual exclusion with blocked-set semaphores is possible, but the solution employs three (instead of one) binary blocked-set semaphores. [Stark 1982] showed that weak semaphores can be used to implement starvation-free mutual exclusion if processes can retain and use information about previous synchronization history to modify future synchronization protocols.

Dijkstra also distinguished between *binary* and *counting* (*general*) semaphores [Dijkstra 1968; Dijkstra 1971]. When the semaphore variable S can take values 0 or 1 only, it is called a binary semaphore, and if S takes any integer value, it is called a counting (general) semaphore.

It is important to be able to distinguish between the various definitions of semaphores because the correctness of a program will depend on the exact definition used.

The Venus operating system [Liskov 1972] provides P and V operations as the basic interprocess communication mechanism. [Lausen 1975] describes the internal structure of a semaphore based operating system BOSS2, developed for RC4000.

2.4.1 Implementation of P & V Primitives :

The binary semaphores allow only one process at a time within an associated critical section. The mutual exclusion protocols given in §2.2.1, §2.2.2, and §2.2.3 achieve this, though they do not adhere to the definition of P and V.

The implementation (in Figure 2.11) of counting semaphores is more interesting, and binary semaphores easily follow from this implementation by initializing the semaphore to 1.

```
Initialization :   S = M , where M is the number of processes
                   which can enter critical section concurrently.

P(S) :  decrement(S);
        if $S < 0$ then block the process and put in a queue;
V(S) :  increment(S);
        if $S \leq 0$ then wakeup one of the waiting processes;
```

Figure 2.11 - An Implementation of Counting Semaphores

This implementation has the problem of underflow if a “huge” number of processors execute P(S).

Here, the power of *increment* and *decrement* instructions becomes clear. A semaphore implemented with *test-and-set* permits only one process to pass, whereas, the solution using *increment* and *decrement* instructions permits up to M processes to pass concurrently.

Keedy *et al* proposed to supplement the semaphore integer variable with a set, which can be thought of occupying one or more words adjacent to the integer word. Each bit in the words for the set represents the absence (0) or presence (1) of a member of the set; this set can be used to indicate when the resource is free, or when no resources are free, it may be used to identify the processes, which are waiting on a resource. The MONADS operating system was developed with microcoded set semaphores. [Keedy 1979]

[Hehner 1981] gave an implementation of P and V semaphore operations, based on the local memory concept where no variable is written by more than one process. This implementation technique is very similar to Lamport's Bakery Algorithm [Lamport 1974], discussed in §4.2.1.

2.4.2 Extensions of P and V Primitives :

Semaphores with P and V primitives have been demonstrated to be adequate and sufficient to solve a wide variety of synchronization problems. However, solutions for synchronization problems involving the scheduling of processes or classes of processes according to different priorities, can be very cumbersome and difficult to discover. The reason for this complexity is that while semaphores are well suited to inhibiting other processes, they cannot directly be used by one class of processes to inhibit other classes of processes. As a consequence, new synchronization primitives and some extensions of P and V primitives were proposed to facilitate solutions for complex synchronization problems.

2.4.2.1 Parallel P and V (PV Multiple) :

[Patil 1971] presented a synchronization problem, *Cigarette Smoker's* problem, and proved that the necessary synchronization cannot be achieved with just P and V operations. He suggested a generalization of P to include simultaneous operations over a finite number of semaphores. That is,

```
$P(S_1, \cdots ,S_n)$ :  if $S_1 > 0 \wedge \cdots \wedge S_n$
                         then $S_1 := S_1 - 1; \cdots ; S_n := S_n - 1$
                         else Suspend;
```

For example, $P(S_1,S_2,S_3)$ can get executed when $S_1,S_2,S_3 \geq 1$, and the execution decreases each of S_1,S_2,S_3 by 1.

[Parnas 1975] however gave a realization of *Cigarette Smoker's* problem by using an array of semaphores, and showed that Patil's claim was wrong.

[Kosaraju 1973] presented a two producer two consumer synchronization problem, and proved that it can not be realized with either arrays of semaphores or multiple P and V primitives.

2.4.2.2 PP and VV Operations :

[Campbell 1973] introduced PP and VV operations which allow several processes to execute a procedure simultaneously. Their definition is given in Figure 2.12.

```
procedure PP(integer count;
        semaphore mutex,sem);
begin
    P(mutex);
    count := count+1;
    if count = 1 then P(sem);
    V(mutex);
end;
```

```
procedure VV(integer count;
        semaphore mutex,sem);
begin
    P(mutex);
    count := count-1;
    if count = 1 then V(sem);
    V(mutex);
end;
```

Figure 2.12 - Definition of PP and VV operations

In the solution of readers-writers problem [Courtois 1971], the code within PP and VV appears several times, and therefore can be replaced easily by these two operations.

2.4.2.3 PV Chunk Operations :

[Vantilborgh 1972] defined the generalized operations P(n,s) and V(n,s) based on the concept of “order” of a blocked (on a semaphore) process. The definition of these operations is given in Figure 2.13. Both P(n,s) and V(n,s) are indivisible operations.

From the definition, it follows that semaphore s is always non-

```
P(n,s) : if n $\leq$ s  then s := s-n
                       else add the performing process to the s-queue
                       (i.e., the queue associated with s) and store n; n
                       is called the order of the blocked process and n $\geq$ 0.
V(n,s) : s := s+n;
         remove from the s-queue a set of processes such that their
         order sum is less than or equal to the current value of s and such
         that there is no other set with this property strictly including
         this set; the current value of s is then decreased by that sum.
```

Figure 2.13 - Definition of P(n,s) and V(n,s) Operations

negative. The major changes with respect to the original P and V operations are that semaphore s can be updated by a value greater than one, and V(n,s) can select a “maximal” set of processes, the order sum of which is less than or equal to the semaphore value.

The “order” feature in this set of semaphores operations distinguishes amongst different processes waiting at the same semaphore and thus, makes it easier to find out solutions for complex synchronization problems such as the *reader-writer* problem.

2.4.2.4 <u>PRIORITY SEMAPHORES</u> :

[Freisleben 1989] presented a new set of primitives, called *priority semaphores*, to solve general scheduling problems involving arbitrary levels of priority. Usage of these new primitives is described in terms of the *reader-writer* problem and then generalized by presenting an algorithm which involves arbitrary levels of priority with support for preemption and shared access by certain process classes.

Here, two new primitives *priority_P* and *priority_V* are introduced. The algorithm presented with these new primitives works, but the primitives may be too big in definition to be defined at machine level. Moreover, the implementation of these primitives would be unlike that of semaphores, which could be implemented using one of the indivisible instructions from §2.1.1 to §2.1.5.

Freisleben *et al* implemented these primitives in microcode for an ICL PERQ system. This implementation revealed execution times of about 8 microseconds for *priority_P* and 5 microseconds for *priority_V* instruction.

2.4.2.5 <u>Higher-level Constructs</u> :

Although semaphores can be used to program almost any kind of synchronization, P and V are rather unstructured primitives. It is easy to make mistakes while using these P and V primitives and the protection of critical sections is left to the programmer. Therefore, structured concurrent programming notations like *conditional critical region* [Hansen 1972a; Hansen 1972b], *monitors* [Hoare 1974; Howard 1976], *distributed processes* [Hansen 1978] and *path expressions* [Campbell 1974] were proposed for specifying synchronization.

2.5 <u>CONCURRENT READING and WRITING</u> :

Mutual exclusion effectively creates a serialization and therefore reduces parallelism. So the search for synchronization solutions which do not implement mutual exclusion is consistently growing. [Lamport 1977] and [Peterson 1983] presented algorithms which show that it is possible to solve synchronization problems like *readers-writers* without resorting to mutual exclusion. There is no serialization at all in these algorithms, and therefore, if MSYPS is a big bottleneck, it would be worth implementing synchronization using one of these two solutions, and then better use of the power of parallel computing would be achieved.

2.5.1 Lamport's Solution :

[Lamport 1977] suggested a synchronization mechanism which permits concurrent reading and writing. In all of the previous solutions, the global variable/semaphore is a basic (atomic) unit of data in memory. However, a data item may be composed of several atomic units. Lamport considered the problem of concurrent reading and writing without introducing mutual exclusion for two reasons - (1)Mutual exclusion requires that a writer wait until all current read operations are completed [Courtois 1971]. This may be undesirable if the writer has higher priority than the readers. (2)The concurrent reading and writing may be needed to implement mutual exclusion.

His paper assumes that there are certain basic units of data whose reading and writing are indivisible, i.e., hardware automatically sequences concurrent operations to the basic unit of data; a basic unit of data may just be a single bit. Lamport considered the case of n readers and one writer and therefore, in his algorithm mutual exclusion of writers is not provided; it needs to be enforced using some other algorithm. A simple solution using semaphore *w* is given in Figure 2.14.

```
P(W);

WRITER's CODE

V(W);
```

Figure 2.14 - Mutual exclusion of writers

Concepts Involved in Concurrent Reading and Writing :

Let V = $d_1 d_2 \ldots d_m$ be an m-digit variable that assumes a sequence of values $V^{[0]}, V^{[1]}, \ldots$ such that $i \leq j$ implies $V^{[i]} \leq V^{[j]}$, that is, write of $V^{[i]}$ precedes the write of $V^{[j]}$. Also, for $k \leq l$, let $V^{[k,l]}$ denote both the value obtained by a read and the assertion that the read saw versions $V^{[k]}$, $V^{[k+1]}, \ldots, V^{[l]}$ and no other versions. Since reading may be concurrent with writing, reading V yields $V^{[k,l]} = d_1^{[i_1]} \ldots d_m^{[i_m]}$, where $d_j^{[i_j]}$ is a part of the version $V^{[i_j]}$ of V; and $k = \min(i_1, i_2, \ldots, i_m)$, and $l = \max(i_1, i_2, \ldots, i_m)$.

If $k = l$, then the read obtained the consistent version $d_1^{[k]} \ldots d_m^{[k]} = V^{[k]}$. It is possible for the read to obtain a consistent version even if $k \neq l$. For example, if $d_1^{[5]} = d_1^{[6]}$, then a read could obtain the value $V^{[5,6]} = d_1^{[5]} d_2^{[6]} \ldots d_m^{[6]} = V^{[6]}$.

If a read of V obtained the value $V^{[k,l]}$, then

(i) The beginning of the read preceded the end of the write of $V^{[k+1]}$.

(ii) The end of the read followed the beginning of the write of $V^{[l]}$.

A read(write) of V is performed *from left to right* if for each j, the read(write) of V_j is completed before the read of V_{j+1} is begun. Reading or writing from *right to left* is defined in the analogous way.

The following results proved in [Lamport 1977] form the basis of the solution (given in §2.5.1.2) to the *readers-writers* problem -

- If V is always *written from right to left*, then a *read from left to right* obtains a value $V_1^{[k_1,l_1]} \ldots V_m^{[k_m,l_m]}$ with $k_1 \leq l_1 \leq k_2 \leq \ldots k_m \leq l_m$. This result holds even when V is not composed of digits; V could be made up of any basic data items then.

- If V is always *written from right to left*, then a *read from left to right* yields a value $V^{[k,l]} \leq V^{[l]}$. In other words, a left-to-right reading of V while V is changing from $V^{[k]}$ to $V^{[l]}$ yields a value that will not exceed $V^{[l]}$. If k = l, then V was not changed during the reading process.

- If V is always *written from left to right*, then *reading* V *from right to left* yields a value $V^{[k,l]} \geq V^{[l]}$. In this case, it is assured that the value will be at least as large as the stored value at the beginning of the read operation.

It is worth demonstrating these theorems using an example. For the example, a digit is an atomic unit of data. If $V^{[0]} = 0999$, $V^{[1]} = 1000$, $V^{[2]} = 1001$, reading V may produce a value $V^{[0,1]}$, $V^{[0,2]}$, $V^{[1,2]}$, depending on the relative speed of the read and write operations, and assuming that V actually changes during the reading :-

(writing from right to left and reading from left to right : result # 2)

$V^{[0,1]} = 0^{[0]}9^{[0]}9^{[0]}0^{[1]}$ or $0^{[0]}9^{[0]}0^{[1]}0^{[1]}$ or $0^{[0]}0^{[1]}0^{[1]}0^{[1]}$.

In any case, $V^{[0,1]} \leq V^{[1]} = 1000$.

$V^{[0,2]} = 0^{[0]}9^{[0]}9^{[0]}1^{[2]}$ or $0^{[0]}9^{[0]}0^{[1]}1^{[2]}$ or $0^{[0]}0^{[1]}0^{[1]}1^{[2]}$ or $0^{[0]}9^{[0]}0^{[2]}1^{[2]}$ or $0^{[0]}0^{[1]}0^{[2]}1^{[2]}$ or $0^{[0]}0^{[2]}0^{[2]}1^{[2]}$;

Thus, $V^{[0,2]} \leq V^{[2]} = 1001$.

Similar examples can be used to demonstrate the third result.

Concurrent Reading & Writing of Readers-Writers Problem :

Using the results in §2.5.1.1, Lamport gave a solution (Figure 2.15) to the general readers-writers problem in the case of a single writer.

This algorithm may be used if either (i)it is undesirable to make the writer wait for a reader to finish reading, or (ii)the probability of having to repeat a read is small enough so that it does not pay to incur the overheads of a solution employing mutual exclusion. The algorithm allows the possibility of a reader looping forever (starving) if writing is done often enough.

```
WRITER                          READER

$\overrightarrow{v1}$ :> v1;    repeat temp := $\overrightarrow{v2}$;

WRITE                           READ

$\overleftarrow{v2}$ := v1;     until $\overleftarrow{v1}$ := temp;
```

NOTE :

1. ':>' means set greater than. Therefore, v_1 :> v_1 can be replaced by $v_1 := v_1 + 1$.
2. The arrow directions on top of variables give the direction of read and write. The variables without arrowheads can be read or written in any direction at that place.

Figure 2.15 - Lamport's Concurrent Reading and Writing Solution to the "Readers-Writers" Problem

On the basis of the results given in §2.5.1.1, [Lamport 1990] gives algorithms for implementing (without forcing mutual exclusion) both a monotonic and a cyclic multiple-word clock that is updated by one process and read by one or more other processes.

2.5.2 Peterson's Solution :

[Peterson 1983] considered the more general Concurrent Reading While Writing (CRWW) problem and provided algorithms which simulate atomic reads or writes for a data item composed of several atomic units so that the writer can modify the data while the readers can obtain a correct, recent value.

[Lamport 1977] considered the concurrent reading and writing problem where the writer is not allowed to wait, but the readers can then be locked out. Lamport's solution (in Figure 2.15) depends on the direction of read and write, and requires shared variables whose values are unbounded (v1:>v1 statement sets v1 to a higher value than before every time a write operation is performed). Peterson solved the CRWW problem with higher level constructs with no direction specification for read and write, and used a bounded number of small, indivisible shared variables.

Peterson's solution is given in Figure 2.16. Here, the writer is wait-free and readers may be locked out (starved). The writer uses a flag *wflag* to signal when it is writing the buffer. A reader can test this *wflag* before and after reading the buffer and determine if it partially overlapped a write. The shared variable *switch* is inverted

Algorithm for the i^{th} reader

```
T1: reading[i] := not writing[i];
T2: sfalg := wflag;
sswitch := switch;
< Read Buffer >;
if sflag or wflag or
  switch ≠ sswitch then goto T2;
if reading[i]=writing[i] then goto T1;
```

Algorithm for the writer

```
wflag := true;
< Write Buffer >;
switch := not switch;
wflag := false;
for j := 1 step 1 until N do
  if reading[j] ≠ writing[j] then
    writing[j] := reading[j];
```

Figure 2.16 - Peterson's CRWW Solution to the Readers-Writers Problem

after each write to detect a write that may have occurred entirely during the read. The problem of two or more writes occurring during a read is handled by a pair of variables, namely *reading*[i] and *writing*[i], per reader - one variable for that reader and the other for the writer.The reader initially sets them to be different, with the writer setting them equal between writes. Hence, the reader, after its read of the buffer, can determine if it overlapped part of a write, an entire write, or two or more writes, in which case it repeats. It is possible that readers may be locked out.

2.6 Summary :

This chapter contains low-level mechanisms which are provided as primitives to the user. The mutual exclusion solutions can be developed from these primitives without much difficulty. In the next chapter, software solutions, which do not depend on any such primitive, are given.

CHAPTER III

SHARED MEMORY HIGH-LEVEL SOLUTIONS

3.1 Introduction :

The problem of how to implement mutual exclusion has been studied extensively. Until 1962, the problem of whether or not processes could be synchronized using just the standard operators of an ordinary programming language had still to be resolved. The first solution to this problem for two processes is credited to the Dutch mathematician T. Dekker. Dijkstra extended this solution to N processes, where N could take any value [Dijkstra 1965]. Since that time numerous extensions have been devised to simplify the algorithm [Doran 1980; Peterson 1981] or improve upon one of the issues of concurrent programming [Knuth 1966; deBruijn 1967; Eisenberg 1972; Lamport 1974; Burns 1982; Lamport 1987].

All of the software solutions in a centralized system use shared variables to achieve mutual exclusion. They assume that the memory hardware mechanism allows exclusive access to the storage locations, that is, several simultaneous accesses (reading and/or writing) to the same location are serialized in an order that is unknown beforehand. None of these solutions makes use of instructions that can perform uninterruptible **read-modify-write** operations because an ordinary higher level language does not supply this operation in any form as a primitive operation. But they do assume that if processor **A** performs **WRITE X** followed by **WRITE Y**, then all other processors will observe the **WRITEs** performed in this order. That is, if processor **A** executes **WRITE X** and then **WRITE Y**, no other processor that executes **READ Y** followed by **READ X** will see the new value of **Y** and the old value of **X**. If it sees the old value of **X**, it will also see the old value of **Y** because **X** is changed before **Y** is changed. This assumption is totally reasonable, yet it need not be obeyed in a multi-processor system unless it is specifically designed into the architecture.

Any system that uses a multi-level switching network between processors and memory can potentially violate this assumption and then all of these software solutions will fail. In a switched network multi-processor system, it is possible that **WRITE X** hits a *hot-spot*[1] and is buffered, while **WRITE Y** succeeds in reaching memory and updating **Y**. In the meantime, another processor issues **READ Y** and **READ X**. Now, **READ Y** obtains the new value of **Y** and also, it is possible for **READ X** to avoid the *hot-spot* that is holding back **WRITE X**, and get the old value of **X**. [Stone 1989]

All of these software solutions also assume that processes do not start in their critical sections and they do not halt outside of their non-critical sections. These are very reasonable assumptions, as otherwise a process halted in its critical section would prevent all other processes from entering the mutually exclusive critical section, and starting straight away in the critical section (without going through the entry (acquisition) protocol) would defeat the

[1] A **hot-spot** is the region of memory that receives more than its share of access in a multi-processor system.

whole purpose of achieving mutual exclusion through these solutions.

Although at present there are more efficient hardware solutions (discussed in Chapter 2) to the problem of mutual exclusion, study of these software solutions is important to realize the inherent difficulty of the problem. Even if there are a dozen or so lines of code in these software solutions, parallelism makes it difficult to understand their behavior and analyze their correctness. To make people realize that these solutions are far from trivial, Dijkstra in his paper [Dijkstra 1965] asked the readers to try (before reading his solution) writing a program to solve this problem. Hyman's incorrect solution is a good citation [Hyman 1966].

The rest of this chapter contains an outline of each of the software solutions (listed in Table 3.1) along with an informal

- Hyman's Incorrect Solution
- Dekker's Solution
- Doran and Thomas' Solution
- Dijkstra's Solution
- Knuth's Solution
- deBruijn's Solution
- Eisenberg and McGuire's Solution
- Peterson's Solution
- Burn's and Lamport's Improvements

Table 3.1 - List of Software Solutions

argument as its proof of preserving mutual exclusion and other properties of the mutual exclusion problem.

3.2 Hyman's Incorrect Solution :

[Hyman 1966] proposed a solution for two processes P_0 and P_1, which compete for access to their critical sections. Knuth showed that Hyman's solution did not preserve mutual exclusion for all interleavings of the execution sequences of two processes [Knuth 1966].

Hyman's solution, consisting of twelve lines of ALGOL program, contained 15 syntactic errors [Knuth 1966]. A structured version of his solution is given in Figure 3.1.

Shared Variables:

```
b : array[0..1] of boolean;   (initialized to true)
k : 0..1;                     (can be either 0 or 1)
```

Protocol for P_0	**Protocol for P_1**
b[0] := **false**;	b[1] := **false**;
while (k ≠ 0) **do**	**while** (k ≠ 1) **do**
begin	**begin**
while not b[1] **do**;	**while not** b[0] **do**;
k := 0;	k := 1;
end;	**end**;
< CRITICAL SECTION >	< CRITICAL SECTION >
b[0] := **true**;	b[1] := **true**;

Figure 3.1 - Hyman's Solution

For the counterexample, consider the case when k=0, and b[0]=b[1]=true. Now, process P_1 sets b[1] to false and then finds b[0] to be true. P_0 then sets b[0] to false, finds k=0 and enters its critical section. But P_1 now sets k=1 and executes its critical section at the same time. Thus, this solution does not achieve mutual exclusion.

3.3 Dekker's Algorithm : [Dijkstra 1968; Silberschatz 1988]

The first software solution to the problem of mutual exclusion was given by Dekker, but was described and proved correct by Dijkstra. The algorithm is given in Figure 3.2.

```
Shared Variables :
flag  : array [0..1] of boolean;  (Initialized to false)
turn  : 0..1;
Note : i contains the process number and j is the other
         process's number.
The Protocol for P_i is -
flag[i] := true;
while flag[j] do begin
   if turn=j then begin
      flag[i] := false;
      while turn=j do;
      flag[i] := true;
   end;
end;
< CRITICAL  SECTION >
turn := j;
flag[i] := false;
```

Figure 3.2 - Dekker's Algorithm

It is clear from the algorithm (in Figure 3.2) that a process would enter its critical section only if other process's *flag* is set to false and its own flag is set to true. If both P_0 and P_1 set their flag to true, *turn* decides who goes inside the critical section. So mutual exclusion is preserved. Since *turn* can be updated only in the postlude, the process with its "turn" will definitely enter the critical section (if it wishes to do so) and hence the algorithm is deadlock-free. However, there is a risk of starvation. It may happen if P_i is a very fast repetitive process; it may constantly find *flag*[j] = false, while P_j cannot set *flag*[j] to true because of P_i's constant reading of the variable *flag*[j] and access to a memory location is exclusive. Process P_j will definitely be able to enter its critical section but only after it sets *flag*[j] to true. Therefore, the "fairness" of this algorithm depends on the fairness of the memory hardware. If the hardware is fair, P_j will get to set *flag*[j] to true in a finite time and eventually enter its critical section.

3.4 Doran and Thomas' Algorithm :

Doran and Thomas presented two variants of Dekker's algorithm as they thought Dekker's algorithm to be difficult (with its nested loops) to comprehend [Doran 1980]. The first variant, given in Figure 3.3, is a rephrasing of Dekker's algorithm, but it consists of two loops in succession rather than the nested loops of Dekker's solution. And the second variant, given in Figure 3.4, has just one loop.

Shared Variables :

```
boolean A_needs, B_needs;
integer turn;
```

Note : The construct "**wait until** <cond.>" is used as an abbreviation for **L: if not** <cond.> **then goto L.**

Protocol for Process A

```
1.  A_needs := true;
2.  if B_needs then begin
3.    if turn = 'B' then begin
4.      A_needs := false;
5.      wait until turn = 'A';
6.      A_needs := true;
7.    end;
8.    wait until not B_needs;
9.  end;
10. < CRITICAL SECTION >
11. turn := 'B';
12. A_needs := false;
13. < NON-CRITICAL SECTION >
```

Protocol for Process B

```
B_needs := true;
if A_needs then begin
  if turn = 'A' then begin
    B_needs := false;
    wait until turn = 'B';
    B_needs := true;
  end;
  wait until not A_needs;
end;
< CRITICAL SECTION >
turn := 'A';
B_needs := false;
< NON-CRITICAL SECTION >
```

Figure 3.3 - Doran and Thomas' Algorithm Version 1

Mutual exclusion in the first variant (Figure 3.3) is guaranteed by each process setting a flag before entering its critical section and then testing the other process's flag immediately before entry (in lines 2 or/and 8). If one process is excluded because the other has already entered its critical section, then the "turn" indicator guarantees that the excluded process will be the next to enter. The proofs for mutual exclusion and no deadlock are similar to the ones given for Dekker's solution.

Both, Dekker's algorithm and this first variant, avoid deadlock by having each process reset its critical section flag before awaiting its turn. This is, in effect, like saying "after-you" to the other process, but use of the critical section flag for this purpose obscures the intended politeness [Doran 1980]. The second variant, given in Figure 3.4, introduces an explicit pair of flags to stand for "after-you". This second variant appears to mirror real life - 'if the other process is using something, or wants to, then say "after-you" politely and wait until it has finished or until it also says "after-you" in which case be well-mannered and do not go first if you had the last turn' [Doran 1980].

In the second variant, the condition tested before entry to the critical section is weaker than in Dekker's solution or the first variant, since it is possible for one process, say A, to enter the critical section while the critical section flag of the other process, i.e. *B_needs*, is true. However, when this occurs, the *B_said_after_you* flag guarantees that B is in its entry protocol and has not entered the critical section. This second variant uses one wait loop and two flags as compared to two wait loops and one flag in the first variant.

The proofs for mutual exclusion and no deadlock for the second variant (in Figure 3.4) are based on *reductio ad absurdum*. Assume that

Shared Variables :

```
boolean A_needs, B_needs, A_said_after_you, B_said_after_you;
integer turn;
```

Note : The construct "**wait until** <cond.>" is used as an abbreviation for **L: if not** <cond.> **then goto L.**

Protocol for Process A

```
1.  A_needs := true;
2.  if B_needs then begin
3.    A_said_after_you := true;
4.    wait until not B_needs or
5.        (turn = 'A' and
6.          B_said_after_you);
7.    A_said_after_you := false;
8.  end;
9.  < CRITICAL SECTION >
10. turn := 'B';
11. A_needs := false;
12. < NON-CRITICAL SECTION >
```

Protocol for Process B

```
B_needs := true;
if A_needs then begin
  B_said_after_you := true;
  wait until not A_needs or
      (turn = 'B' and
        A_said_after_you);
  B_said_after_you := false;
end;
< CRITICAL SECTION >
turn := 'A';
B_needs := false;
< NON-CRITICAL SECTION >
```

Figure 3.4 - Doran and Thomas' Algorithm Version 2

both processes are in their critical section at the same time. Both A and B must get past lines 4-6 in the program to get to their critical section. If A is in its critical section, it must have set *A_needs* to true at line 1, and found either *B_needs* false at line 4 or (*turn* = 'A' and *B_said_after_you*) true at lines 5 and 6 respectively. Since by assumption B is also in its critical section, it must have set *B_needs* true and found either *A_needs* false or (*turn* = ‘B’ and *A_said_after_you*) true. Therefore, the only way A and B could have got into their critical section at the same time is when A found *turn* = ‘A’ when B was in its critical section or attempting to enter it, and B found *turn* = ‘B’ when A was in its critical section or attempting to enter it. Since *turn* is a shared variable and is updated only in the exit protocol, it can be either ‘A’ or ‘B’ and never both ‘A’ and ‘B’. This is a contradiction and thus mutual exclusion is achieved.

Deadlock cannot occur in this algorithm. If one of the processes is halted in its non-critical section, the other process will find the halted process’s *needs* flag to be false and will then be able to go past lines 2 to 8 in the protocol in Figure 3.4 and hence access the critical section without any resistance from the halted process. If both A and B are attempting to enter the critical section at the same time, then *turn* (being either ‘A’ or ‘B’) will decide who goes into the critical section. Therefore, both ‘A’ and ‘B’ cannot be blocked and hence there is no deadlock.

Both of these variants, like Dekker’s, depend on the fairness of the memory hardware to be “fair” to the processes competing to enter the critical section.

3.5 Dijkstra’s Generalization to N Processes :

[Dijkstra 1965] generalized Dekker’s solution to the case of N processes. Dijkstra’s algorithm is given in Figure 3.5.

This algorithm allows a process to enter its critical section

Shared Variables :

Boolean array b, c [1:N]; (initialized to true)

integer k;

Note : $1 \leq k \leq N$. b[i] and c[i] are set by P_i only, where as all other processes can only read them. Here, i contains the process number, and N is the total number of processes.

Local Variables : integer j;

Protocol for Process P_i $(1 \leq i \leq N)$ is -

```
Li0:  b[i] := false;
Li1:  if k ≠ i then
Li2:  begin  c[i] := true;
Li3:         if b[k] then k := i;
             goto Li1;
      end
Li4:  else begin
             c[i] := false;
             for j := 1 step 1 until N do
               if (j ≠ i) and (not c[j]) then goto Li1;
      end;
      < CRITICAL SECTION >
      c[i] := true;
      b[i] := true;
      < NON-CRITICAL SECTION >
      goto Li0;
```

Figure 3.5 - Dijkstra's Solution

only when it finds all other *c*'s true after having set its own *c* to false. Mutual exclusion is achieved and to prove this, assume, to the contrary, that two processes P_i and P_j are in their critical section simultaneously. To enter critical section, P_i must set c[i] to false and find c[j] to be true. On the other hand, P_j must have found c[i] to be true after setting c[j] to false. This leads to a contradiction, and hence the assumption that two processes are in their critical section at the same time is wrong, and mutual exclusion is achieved.

This solution also avoids "after you"-"after you" kind of blocking (deadlock). If the process P_k is not trying to enter the critical section, b[k] will be true and all the other processes trying to enter the critical section will find $(k \neq i)$ true. As a result, several processes may execute the assignment statement in Li3, i.e. *k* := *i*. After the first assignment, no new process can assign a new value to *k* as they all will find b[k] false. Since *k* is a shared variable, it will contain the number of the last process, say i, to have had carried out the assignment (*k* := *i*), and will not change until b[i] becomes true. Now, P_i will wait (in Li4) until all other processes set their *c* true, and then P_i will enter its critical section. Therefore, when none of the processes is in the critical section, one process will be able to do so. Hence, there is no deadlock.

If a number of processes are constantly competing for the critical section, there is nothing to stop one of the processes from always entering the critical section as this process can always be the last one to modify *k*. So in this algorithm it is possible that a process may get starved.

3.6 Knuth's Solution :

[Knuth 1966] presented the first "fair" software solution to the problem of mutual exclusion. By providing "fairness" in the algorithm, a process trying to enter its critical section is guaranteed to do so within a finite time. The bound is given by the number of times that other processes may enter the critical section between the moment a process submits a request to enter its own critical section and the moment it actually does so [Knuth 1966]. After this algorithm, it became possible to measure the maximum waiting time (in number of times) for a process to enter its critical section. Knuth's algorithm appears in Figure 3.6.

This solution guarantees mutual exclusion for it is impossible for two processes to go past the loop in line L2 in Figure 3.6. To prove this, assume that two processes P_i and P_j are in the critical section simultaneously. For P_i to be in the critical section, *control*[i] must be equal to 2, and *control*[j] must be either 0 or 1 (i.e., not 2). But, for P_j to be in its critical section, *control*[j] must be 2. This shows that the assumption that two processes may be in the critical section simultaneously leads to a contradiction.

The algorithm also guarantees that the critical section is reachable, that is the system cannot be deadlocked; because if no process enters the critical section, the value of *k* remains constant and the first process (in the cyclic ordering k,k−1,...,1,N,N−1,...,k+1) attempting to enter will have no restraint to do so.

```
Shared Variables :
integer array control [1:N];          (initialized to 0)
integer k;                            (initialized to 0)
Note : Here, i contains the process number,
and N is the total number of processes.
Local Variables : integer j;
Protocol for Process $P_i$ is -
L0:   control[i] := 1;
L1:   for j := k step -1 until 1, N step -1 until 1 do
      begin
         if j = i then goto L2;
         if control[j] $\neq$ 0 then goto L1
      end;
L2:   control[i] := 2;
      for j := N step -1 until 1 do
         if (j $\neq$ i) $\wedge$ (control[j] = 2) then goto L0;
L3:   k := i;
      < CRITICAL SECTION >
      k := if i = 1 then N else (i-1);
L4:   control[i] := 0;
L5:   < NON-CRITICAL SECTION >
      goto L0;
```

Figure 3.6 - Knuth's Solution

Knuth's solution is "fair". Since the critical section is reachable, a process P_i can be blocked only if there is at least one other process P_j that gets to execute its critical section arbitrarily often. But every time P_j gets through from L0 to L4, with *control*[i] $\neq$ 0, it encounters the value of *k* (in L1) that must have been set by a process P_l which follows i and precedes j in the cyclic ordering N,N-1,...,2,1, i.e., $i > l > j$. Since by assumption P_j

continually overtakes P_i, the effect allowing this to happen must occur continually, i.e. P_l should always then enter the critical section before P_j. There must therefore be a process $P_{k'}$ that follows P_i and preceded P_l, that is $i > k' > l$, and so on. Since the number of processes N is finite, P_i must at some stage enter its critical section. Thus, the fairness of the solution is guaranteed.

Knuth claims that a process has to wait at most $2^{N-1} - 1$ *turns*, where N is the number of processes, and *turn* is defined as one process using its critical section. Proof for this maximum delay function is not trivial [deBruijn 1968]. In unfavorable circumstances, a process P_i trying to enter its critical section may be positioned at L1, and all other (N-1) processes at L2. The worst case ($2^{N-1} - 1$ *turns*) would be when P_i misses all the momentary values of *k* which would enable it to get through to L2. But after this worst case delay, the value of *k* cannot be changed further by any other process and then P_i would be able to enter its critical section.

The value of *k* is changed in L3 only, and a careful look shows that the new value of *k* is computed using modulo N arithmetic. The process numbers 1..N can be mapped to 0..N-1 and *k* can be thought to be computed as $k := (k\text{-}1) \bmod N$. For the worst case computation, assume P_j, such that $j = (i + 1) \bmod N$, gets into the critical section first, and therefore sets $k = i$ after exiting the critical section. Now, there are (N-2) processes (left at L2) which can potentially change the value of *k* and assume that P_j joins P_i at L1. Again, assume P_l, such that $l = (i + 2) \bmod N = (j + 1) \bmod N$, enters the critical section and therefore, changes *k* to j. Now, process P_j can cross the barrier at L1 and join the other processes at L2. Assume this happens and therefore there are again $(N-2)$ processes at L2 with P_i at L1. Further, assume that P_j again enters the critical section before all the other waiting processes at L2. Now, there are (N-3) processes left at L2 and possibly, P_j waiting at L1 with P_i and P_l at L0. Again assume that P_m, such that $m = (i+3) \bmod N = (j+2) \bmod N = (l+1) \bmod N$, is the next to enter the critical section, and therefore the new value of *k* is 1, after P_m exits its critical section. This new value of *k* enables both P_j and P_l to cross L1, assuming P_i again misses the chance. Now, assume that P_j goes inside the critical section first, then P_l, and then again P_j. Thus, by continuing the situations unfavorable to P_i, it can be seen that P_j, such that $j = (i+1) \bmod N$, enters the critical section (1 (enabled itself) + 2^0 (enabled by $P_{(j+1)modN}$) + 2^1 (enabled by $P_{(j+2)modN}$) $+ \cdots +$ 2^{N-3} (enabled by $P_{(j+N-2)modN\, =\, (i-1)mod\ N}$) times $= 1 + \frac{2^0(2^{N-2}-1)}{2-1} =$) 2^{N-2} times. And then generalizing,

$P_{(i+1)modN}$	enters critical section	2^{N-2} times
$P_{(i+2)modN}$	enters critical section	2^{N-3} times
...	...	...
...	...	...
...	...	...
$P_{(i+N-1)modN}$	enters critical section	2^0 times

The sum total of the number of times would give the maximum delay. Therefore, maximum delay is $= 2^{N-2} + 2^{N-3} + \ldots + 2^0$

$$= \frac{2^0 * (2^{N-1}-1)}{(2-1)} \text{ (Sum of a Geometric Progression)}$$

$$= \boxed{2^{N-1}-1}.$$

Knuth gave a simpler version of his N process algorithm to solve the mutual exclusion problem in case of two processes [Knuth 1966]. It appears in Figure 3.7.

It is interesting to compare Knuth's two process solution (Figure 3.7) with that of Dekker (Figure 3.2), and Doran and Thomas (Figures 3.3 and 3.4). The latter solutions depend on the fairness of the hardware, whereas Knuth's algorithm ensures fair behavior.

```
L0:   control[i] := 1;
L1:   if k = i then goto L2;
      if control[j] ≠ 0 then goto L1;
L2:   control[i] := 2;
      if control[j] = 2 then goto L0;
L3:   k := i;
      < CRITICAL SECTION >
      k := j;
L4:   control[i] := 0;
```

Figure 3.7 - Knuth's Two Process Solution

3.7 deBruijn's Solution :

Knuth's solution is not efficient, when N is very large. [deBruijn 1967] proposed an improvement to Knuth's algorithm. deBruijn suggested a very small change in Knuth's solution and reduced the order of maximum number of waits from exponential to polynomial time. The change is in line L3 of Knuth's algorithm, that is the part of the protocol where *k* is updated. The change is given in Figure 3.8 and the complete algorithm in Figure 3.9.

```
                              Replace
L3:    k := i;                          L3:    < CRITICAL SECTION >
< CRITICAL SECTION >              by    if $(control[k] = 0) \vee (k = i)$
k := if i = 1 then N else i – 1;          then k := if k = 1 then N else k – 1;
```

Figure 3.8 - Changes in Knuth's Solution

The modification made to Knuth's algorithm does not effect the proofs of mutual exclusion and of critical section reachability; they

```
Shared Variables :
integer array control [1:N];          (initialized to 0)
integer k;                            (initialized to 0)
Note : Here, i contains the process number,
and N is the total number of processes.
Local Variables :  integer j;
Protocol for Process $P_i$ is -
L0:    control[i] := 1;
L1:    for j := k step -1 until 1, N step -1 until 1 do
       begin
          if j = i then goto L2;
          if control[j] $\neq$ 0 then goto L1
       end;
L2:    control[i] := 2;
       for j := N step -1 until 1 do
          if $(j \neq i) \wedge (control[j] = 2)$ then goto L0;
L3:    < CRITICAL SECTION >
       if $(control[k] = 0) \vee (k = i)$ then
          k := if i = 1 then N else (i-1);
L4:    control[i] := 0;
L5:    < NON-CRITICAL SECTION >
       goto L0;
```

Figure 3.9 - deBruijn's Solution

remain the same. The alterations however affect the “fairness” issue. With these changes, if at a given moment k is i, and if *control*[i] $\neq 0$, then k does not change its value before process P_i has executed the critical section. For the time k is constant, no process can enter the critical section twice. Suppose P_j passes twice, then $j \neq k$ and *control*[k] $\neq 0$, for otherwise k would have changed the first time P_j went through the critical section. Further, P_k does not pass its critical section before P_j does; otherwise the value of k would change before P_j gets its second turn. Therefore, *control*[k] $\neq 0$ all the time between the two turns of P_j, and this means that P_j cannot get to L2 after its first turn. Now following the arguments similar to those in §3.6, the worst case delay can be computed. Therefore, if a process P_i attempts to enter the critical section, then in the worst case -

$P_{(i+1)\text{mod}N}$	can enter the critical section	$(N-1)$ times,
$P_{(i+2)\text{mod}N}$	can enter the critical section	$(N-2)$ times,
...	...	...
...	...	...
...	...	... , and
$P_{(i+N-1)\text{mod}N}$	can enter the critical section	1 time,

before P_i gets inside the critical section.

$$\text{Therefore, maximum delay } = (N-1) + (N-2) + \ldots + 1$$

$$= \sum_{j=1}^{N-1} (N-j)$$

$$= \boxed{\frac{N*(N-1)}{2}}.$$

deBruijn’s algorithm highlights the difficulties encountered in writing efficient concurrent programs - a small change in the algorithm can do wonders!

3.8 Eisenberg and McGuire's Algorithm :

After deBruijn's optimization over Knuth's solution, [Eisenberg 1972] proposed a solution which further optimizes the maximum delay by guaranteeing it to be no more than $(N-1)$ turns, i.e. a linear function over deBruijn's quadratic. Eisenberg and McGuire's algorithm is given in Figure 3.10.

The solution in Figure 3.10 ensures that no two processes are simultaneously processing between their statements L3 and L6 for the same reason as in Knuth's algorithm, and hence mutual exclusion is achieved. Also, the algorithm is deadlock-free, for if no process has yet passed the statement L3 before entering the critical section, the value of *k* will be constant and the first contending process in the cyclic ordering (k,k+1,...,N,1,...,k-1) will meet no resistance and enter the critical section.

The algorithm is "fair" and guarantees that no process will be starved. When a process exits its critical section, it designates the first contending process (in the cyclic ordering) as its unique successor by setting *k* to that process's identification number. This also ensures that, in the worst case, where all processes are attempting to enter the critical section, the maximum wait for process P_i (with $k=i+1$) to enter the critical section is limited to $(N-1)$turns. Since the delay function is linear, a process may overtake another at most once.

Shared Variables :

integer array control [1:N]; (initialized to 0)

integer k;

Note : $1 \leq k \leq N$. Each element of control is either 0, 1, or 2. Here, i contains the process number and N is the total number of processes.

Local Variables : integer j;

Protocol for process P_i is -

```
L0:   control[i] := 1;
L1:   for j := k step 1 until N, 1 step 1 until k do
      begin
         if j = i then goto L2;
         if control[j] $\neq$ 0 then goto L1
      end;
L2:   control[i] := 2;
      for j := 1 step 1 until N do
         if (j $\neq$ i) $\wedge$ (control[j] = 2) then goto L0;
L3:   if (control[k] $\neq$ 0) $\wedge$ (k $\neq$ i) then goto L0;
L4:   k := i;
      < CRITICAL SECTION >
L5:   for j := k step 1 until N, 1 step 1 until k do
         if (j $\neq$ k) $\wedge$ (control[j] $\neq$ 0) then
         begin
            k := j;
            goto L6
         end;
L6:   control[i] := 0;
L7:   remainder of cycle;
      goto L0;
```

Figure 3.10 - Eisenberg and McGuire's Solution

3.9 Peterson's Solution :

[Peterson 1981] presented a very simple solution to the problem of mutual exclusion for two processes and claimed to put an end to the "myth" that the two process mutual exclusion problem requires complex solutions with complex proofs.

Peterson gave two primitive algorithms (given in Figures 3.11 and 3.12), which preserve mutual exclusion but suffer from deadlock, and then derived the working algorithm (Figure 3.13) from these two primitive algorithms.

<table>
<tr><td colspan="2"><b>Shared Variables :</b>
turn : <b>integer;</b>
<b>Note :</b> The construct <b>wait until</b> &lt;cond.> means that wait until condition is true and can be replaced by the standard construct <b>repeat until</b> &lt;cond.&gt;.</td></tr>
<tr><td><b>Protocol for Process 1</b>
turn := 1;
<b>wait until</b> (turn = 2);
< CRITICAL SECTION &gt;</td><td><b>Protocol for Process 2</b>
turn := 2;
<b>wait until</b> (turn = 1);
< CRITICAL SECTION &gt;</td></tr>
</table>

Figure 3.11 - Peterson's First Primitive Algorithm

The first primitive algorithm (Figure 3.11) suffers from deadlock only when one of the processes does not cyclically try for the critical section. The second primitive algorithm (Figure 3.12) has deadlock only when both the processes are attempting to get into their critical section.

Shared Variables :

Q1, Q2 : **boolean;**

Note : The construct **wait until** <cond.> means that wait until condition is true and can be replaced by the standard construct **repeat until** <cond.>.

Protocol for Process 1	**Protocol for Process 2**
Q1 := true;	Q2 := true;
wait until not Q2;	**wait until not** Q1;
< CRITICAL SECTION >	< CRITICAL SECTION >
Q1 := **false;**	Q2 := **false;**

Figure 3.12 - Peterson's Second Primitive Algorithm

To prove that the algorithm in Figure 3.13 achieves mutual exclusion, assume that both processes P1 and P2 are in their critical section at the same time. That would then mean Q1 = Q2 = true. Now, the compound condition in the wait loop could not be true for both the

Shared Variables :

Q1, Q2 **boolean;**

turn : **integer;**

Note : The construct **wait until** <cond.> means that wait until condition is true and can be replaced by the standard construct **repeat until** <cond.>.

Protocol for Process 1	**Protocol for Process 2**
Q1 := **true;**	Q2 := **true;**
turn := 1;	turn := 2;
wait until (**not** Q2) **or** (turn = 2);	**wait until** (**not** Q1) **or** (turn = 1);
< CRITICAL SECTION >	< CRITICAL SECTION >
Q1 := **false;**	Q2 := **false;**

Figure 3.13 - Peterson's Solution to Two Process Mutual Exclusion Problem

processes at the same time, as the shared variable *turn* would be favorable to only one of the processes and the other condition (*not Q1* for P1 and *not Q2* for P2) would have failed for both. It implies that one process first passed its test and therefore entered its critical section. Now the second process can enter its critical section only when it finds *turn* favorable to it, but it can only make *turn* unfavorable to itself. Therefore the second process is definite to fail the test and thus mutual exclusion is preserved.

Deadlock is also not possible for the algorithm in Figure 3.13. To prove this, consider P1 blocked in its wait loop forever. After a finite amount of time, P2 will be doing one of three things - not trying to enter its critical section, waiting in its protocol for entry to critical section, or using the critical section again and again. In the first case, P1 finds that *Q2* is false and then it may proceed to enter its critical section. The second case is impossible as *turn* must be either 1 or 2, and this will make the condition true for one of the processes to proceed. In the third case, when P2 attempts to use its critical section again, it will set *turn* to 2 (unfavorable to itself), and therefore permit P1 to proceed. The third case demonstrates that this algorithm guarantees fairness also.

Peterson thought that there was no need of a formal proof for this simple algorithm and opined that “possibly the prevalent attitude on formal correctness arguments is based on poorly structured algorithms and good parallel programs are not really that hard to understand”. In fact, he found Dijkstra’s formal proof of mutual exclusion for the “simple” algorithm in Figure 3.13

"unnaturally complex".

Peterson also showed that his two process solution could easily be generalized to N processes. The N process solution, given in Figure 3.14, is formed by using the two process solution repeatedly (N − 1) times to eliminate at least one process each time until only one remains. In this algorithm, variable Q has been generalized to take N values, ranging from 0 to N − 1. The value 0 plays the same role as "false" does for two process solution, that is to convey that the process is not in its critical section. A process's entry to the critical section, expressed by "true" in two process solution, is now specified with respect to the other processes.

Shared Variables :

```
Q     :  array [1..n]   of integer;      (initially 0)
turn  :  array [1..n-1] of integer;      (initially 1)
```

Local Variables : i, n, j : **integer;**

Note : The construct **wait until** <cond.> means that wait until condition is true and can be replaced by the standard construct **repeat until** <cond.>. Here, i contains the process number and N is the total number of processes.

Protocol for P_i is -

```
for j := 1 to N-1 do
begin
   Q[i] := j;
   turn[j] := i;
   wait until $(\forall k \neq i, Q[k] < j) \lor (turn[j] \neq i)$
end;
< CRITICAL SECTION >
Q[i] := 0;
```

Figure 3.14 - Peterson's Solution to N Process Mutual Exclusion

3.10 Further Improvements :

The mutual exclusion algorithms in §3.6 to §3.9 are improvements over Dijkstra's solution in terms of either simplicity or "fairness". All of these algorithms do not pay any attention to the number of shared memory variables used and the number of reads and writes to the shared memory. These aspects of concurrent programming cannot be overlooked if the shared memory size and the execution time of the mutual algorithm is as critical as providing mutual exclusion.

3.10.1 Burn's Improvements :

Burns proved that any protocol (based on exclusive read and write access to the shared memory) providing deadlock-free mutual exclusion of N processes must use at least N+1 shared variables, N of size 2 (i.e., contains only one of the two values, like boolean variables) and one whose size must be at least N (i.e., contains one out of N different values) [Raynal 1986]. Burns also studied mutual exclusion solutions based on test-and-set operations and gave upper and lower limits on the amount of shared memory, measured by counting the number of distinct values which it can assume. Table 3.2 contains

Mutual Exclusion Algorithms	Upper-limit	Lower-limit
	Values	Values
Deadlock-free	2	2
Deadlock-free and Starvation-free	$\lfloor N/2 \rfloor + 9$	$\sqrt{2N} + \frac{1}{2}$
Deadlock-free and Bounded waiting	N + 3	N + 1

Table 3.2 - Burn's results for the amount of shared memory used

these limits. Based on these results, he developed mutual exclusion solutions which use the optimal number of shared values [Burns 1982].

3.10.2 Lamport's Improvement :

A great deal of effort was spent in developing algorithms (in §3.6 to §3.9) that do not allow a process to wait longer than it "should" while other processes are entering and leaving the critical section. However, the current belief among operating system designers is that contention for a critical section is rare in a well-designed system; most of the time, a process will be able to enter without having to wait. Even an algorithm that allows individual processes to starve, while other processes keep on entering the critical section, is considered to be acceptable, since such situations are unlikely to occur. [Lamport 1987]

Lamport judged the solutions by how fast they are in the absence of contention. With modern high-speed processors, an operation that accesses shared memory takes much more time than one that can be performed locally. Hence, the number of reads and writes to shared memory is a good measure of an algorithm's execution time.

All the published N process solutions require a process to execute O(N) operations to shared memory in the absence of contention [Lamport 1987]. Lamport presented an N process mutual exclusion solution that does only 5 writes and 2 reads of shared memory.

Lamport gave a step-by-step description of his solution to support his claim of minimum sequence of memory accesses needed to guarantee mutual exclusion. He claimed that the best possible algorithm is one in which the sequence of reads and writes is given by the sequence -

write x, read y, write y, read x, critical section, write y.

The arguments for this sequence run like this -

- There is no point making the first operation in the sequence a read, since all processes could execute the read and find the initial value before any process executes its next step. So the first operation should be a write of some variable x.

- It makes no sense for the second operation in the sequence to be another write to x. There is also no reason to make it a write to another variable y, since the two successive writes could be replaced by a single write to a longer word. Therefore, the second operation in the sequence should be a read. This operation should not be a read of x because the second operation of each process could be executed immediately after its first operation, with no intervening operations from other processes, in which case every process reads exactly what it had just written and obtains no new information. Therefore, each process must perform a write to x followed by a read of another variable y.

- There is no reason to read a variable that is not written or write a variable that is not read. So the sequence must also contain a read of x and a write of y.

- The last operation performed, before entering the critical section in the absence of contention, should not be a write because that write could not help the process decide whether or not to enter the critical section. Therefore, the best possible algorithm,before entering the critical section, has the following sequence of memory accesses -

write x, read y, write y, read x.

The algorithm based on these arguments is given in Figure 3.15. Here, each process first writes x, then reads y. If it finds that y has its initial value, then it writes y and reads x. If it finds that x has the value it wrote in the first operation, then it enters the critical section.

Shared Variables :

x, y : **integer**; (y = 0 initially)

Note : 1. Atomic operations are enclosed by angular brackets.

2. The '**delay**' in the second **then** clause must be long enough so that, if another process j read y equal to 0 in the first **if** before i set y = i, then j will either enter the second **then** clause or else execute the critical section and reset y to 0 before i finishes executing the **delay**.

Protocol for P_i is -

```
start:  < x := i >;
        if < y ≠ 0 > then goto start fi;
        < y := i >;
        if < x ≠ i > then delay;
            if < y ≠ i > then goto start fi fi;
        CRITICAL SECTION
        < y := 0 >;
```

Figure 3.15 - Lamport's Algorithm for N Process Mutual Exclusion with 3 Writes and 2 Reads to the shared memory

- After executing its critical section, a process must execute at least one write operation to indicate that the critical section is vacant, so processes entering later realize there is no contention. It cannot be done with a write of x, since every process writes x as the first access to shared memory when performing the protocol. Therefore, a process must write y, resetting y to its initial value after exiting the critical section.

The algorithm in Figure 3.15 requires not only an upper bound

```
Shared Variables :
x, y: integer;                  (y = 0 initially)
b   : array [1..N] of boolean;  (initially false)
Note : 1. Atomic operations are enclosed by angular brackets.
       2. await cond. is an abbreviation for while not cond. do;
Protocol for P_i is -

start:  < b[i] := true >;
        < x := i >;
        if < y ≠ 0 > then < b[i] := false;
            await < y = 0>;
            goto start fi;
        < y := i >;
        if < x ≠ i > then < b[i] := false >;
            for j := 1 to N do await < not b[j] > od;
            if < y ≠ i > then await < y = 0 >;
                goto start fi fi;
        CRITICAL SECTION
        < y := 0 >;
        < b[i] := false >;
```

Figure 3.16 - Lamport's Algorithm for N Process Mutual Exclusion with 5 writes and 2 Reads to the shared memory

on the time required to perform an individual operation such as memory reference, but also on the time needed to execute the critical section. In most situations, an algorithm that does not require this upper bound is needed. Therefore, the algorithm in Figure 3.15 is not acceptable. Lamport introduced a new variable *b*[i] to improve upon the algorithm in Figure 3.15. This new variable indicates when a process is inside its critical section and therefore removes the knowledge of how long a process can stay in its critical section. Now, a process must set this new variable to indicate that it is in its critical section, and must reset that variable to indicate that it has left the critical section. But, it brings two additional memory writes to the shared memory. This optimal algorithm, in Figure 3.16, performs only 5 writes and 2 reads to the shared memory to achieve mutual exclusion for N processes.

Both the algorithms (Figures 3.15 and 3.16) guarantee deadlock-free mutual exclusion, but allow starvation of individual processes.

3.11 Summary :

The software solutions in this chapter depend on the availability of shared memory to implement mutual exclusion. In distributed systems, there is no common memory. Therefore, these solutions will not work. In the next chapter, solutions which use message-passing primitives are given to solve the mutual exclusion problem in distributed systems.

CHAPTER IV

DISTRIBUTED SOLUTIONS

4.1 Introduction :

A distributed system is a collection of independent processors (referred to as *sites* or *nodes*) which are spatially separated and which communicate with one another only by exchanging messages [Lamport 1978]. Independent processors have neither a shared memory nor a common clock. Instead, each processor has its own local memory to which it has the sole access. [Enslow 1978] characterized a distributed system to contain the following five components - a **multiplicity** of resources, a **physical distribution** of the resources, a **high-level operating system**, **system transparency**, and **cooperative autonomy**. [Silberschatz 1991] gave four major reasons for building distributed systems - **resource sharing**, **computation speedup**, **reliability**, and **communication**.

An important characteristic of distributed systems is that the message transmission delay is not negligible, compared to the time between events in a single process [Lamport 1978]. In the following discussion, the term 'process' means a process on a site and the term 'processes' is used to mean processes on different sites.

A distributed system can have any topology of a physical communication network, e.g. fully connected, partially connected, tree, ring, bus, etc. [Tanenbaum 1988]. The only assumption made is that there is a logical connection between any two sites and a routing mechanism exists that delivers messages between sites. The mutual exclusion algorithms in a distributed system may logically organize the sites to form a structure such as tree, ring, etc.

Each site is assigned a unique identifier to distinguish it from other sites and almost all distributed algorithms assume this as a precondition. The task of assigning unique identifiers is referred to as the *naming problem* [Beauquier 1990]. One method of naming is to have a token circulating in the network that has an integer variable whose value at start is 1. A site chooses the value of this integer variable as its unique *name* (identifier) on the token's first arrival at that site, and it increases the value by 1 on the token's first departure. Thus each site gets a unique identifier even if it does not know about the entire network. However, this method depends on the fact that each site transmits the token and increases its value correctly. If a given site decreases the value of the token instead of increasing it, two sites would receive the same identifier. Also, a failed site may decide to keep the token forever, and then this naming method would fail. [Lamport 1982] referred to the problem of "bad" sites performing anything (for example, sending false messages or not sending messages at all) as the *Byzantine Generals Problem*. [Beauquier 1990] studied the naming problem in distributed systems in which some sites can have byzantine faulty behavior. All of the mutual exclusion algorithms in this chapter assume that sites do not malfunction on failing, that is, there are no byzantine failures.

The underlying network is also assumed to be reliable. The network protocols are responsible for error-free and loss-free delivery of messages.

Two primitives, namely ***send*** and ***receive***, are defined for interprocess communication. All processes in a distributed system exchange information using these two primitives. Since a message can be received only after it has been sent, message passing also forms the basis of process synchronization. In fact, ‘send’ and ‘receive’ can be considered as **V** and **P** semaphore operations on the number of queued messages [Andrews 1991b].

Interprocess communication can be *synchronous* or *asynchronous*. With synchronous message passing, a process sending a message is delayed until the other process is ready to receive the message. Asynchronous message passing, on the other hand, does not cause the sending process to block, rather it allows the process to continue executing while a message is being sent on its behalf. Since synchronous communication may decrease the overall throughput of the system, asynchronous communication is a preferred choice [Schneider 1982]. But with asynchronous message passing, senders can get far ahead of receivers, and therefore receivers can never be sure of obtaining the current state of the sending process - a sender process may change its state by the time the receiver receives the message containing sender’s state information. Consequently, in a distributed system, no single process can have a complete knowledge of the global state of the system [Chandy 1985].

The problem of mutual exclusion arises in distributed systems like it does in centralized shared memory systems. That is, concurrent access to a physically or logically shared resource by several sites needs to be serialized. But it is more complex to implement mutual exclusion in distributed systems because of the lack of knowledge about the global state of the system (which is not a problem in centralized systems as it can be obtained from the shared memory), lack of a common physical clock and unpredictable message delays.

One of the inherent advantages of a distributed system is *failure-tolerance* [Sanders 1987], since when one site fails, others can continue operating. When a site fails, or when the communication subsystem (links between sites) fails, a failure-tolerant mutual exclusion algorithm should be able to adapt to the new conditions so that it continues to operate with the remaining processors and still maintain mutual exclusion. [Spector 1984] describes how the first launch of the space shuttle was delayed because of a fault in the synchronization between the main computer and the back-up computer. Distributed solutions tend to be more fault-tolerant than centralized systems, because they do not depend on any global variables [Raynal 1986].

The shared memory mutual exclusion solutions in Chapters II and III assume that access to a shared memory location is mutually exclusive. Since these solutions assume a lower level hardware solution (conflicting memory access arbiter) to the problem they are solving, Lamport found them unsatisfactory [Lamport 1986a]. He defined interprocess communication based on a communication variable that does not assume any lower-level mutual exclusion [Lamport

1986a], and then gave a mutual exclusion solution using this communication variable [Lamport 1986b]. The distributed mutual exclusion solutions in this chapter do not assume any lower-level mutual exclusion and can be used to achieve mutual exclusion in centralized systems which provide 'send' and 'receive' primitives. [Kessels 1982] showed that the shared modifiable variables of Peterson's algorithm (see §3.9) could be distributed to form a distributed mutual exclusion algorithm which does not require an arbiter on a lower-level.

The mutual exclusion problem in distributed systems can be solved by two kinds of mechanisms - *centralized control* or *distributed* (*decentralized*) *control*. In centralized control mechanisms, all requests to use the critical section pass through a single site which is responsible for granting access to the critical section. In a distributed control mechanism, each site in the distributed system is equally responsible for controlling mutual exclusion. The primary disadvantages of a centralized control mutual exclusion algorithm are that the central site becomes a source of contention and when the central site is "down" or inaccessible because of communication network failure, the critical section cannot be reached by any process. On the other hand, distributed control, in principle, allows at least one process to access the critical section even when one or more sites are inaccessible. Creation of a mutual exclusion solution in a computer network under distributed control is not trivial [Maekawa 1985]. This chapter describes only distributed control algorithms. Centralized control algorithms can be derived from some of these as a special case.

The distributed mutual exclusion algorithms can be classified into two categories - one which does not use message passing primitives explicitly (§4.2) and the other that uses 'send' and 'receive' primitives explicitly (§4.3).

A distributed mutual exclusion algorithm is evaluated in terms of the number of messages exchanged, the number of information bits exchanged, delay and resilience to failure-tolerance [Suzuki 1985].

All solutions in this chapter implement mutual exclusion at the node level. If there is more than one process within a site trying to access the shared resource, then it is assumed that these intra-site conflicts are resolved using one of the techniques given in Chapters II and III.

4.2 Solutions Without Explicit Usage of Message Passing Primitives :

The solutions in this section achieve mutual exclusion by having a site, which is trying to access critical section, obtain state information from other sites. The state information of a site corresponds to the values of the variables used for serializing access to the shared resource. And the act of obtaining a site's (say A's) state information by another site (say B) involves transmission of a message from B to A requesting the value of A's variables, followed by B's receipt of a message from A containing the values.

These algorithms are given using high-level abstraction and therefore do not use 'send' and 'receive' primitives explicitly in the solution. Instead, these algorithms have ‘read variable’ statements to obtain the state information. In a distributed system, ‘read variable’ statement is implemented using ‘send’ and ‘receive’ primitives. This abstraction hides the implementation details and makes the algorithm easier to understand.

Another way to look at these solutions is that a process can both read from and write to its local memory, but can only read from other process’s local memory. So there is no global variable, like centralized systems, which is written by more than one process.

Multiprocessor systems offer read only access to a processor’s memory by another processor. So the algorithms in this section may be implemented on a multiprocessor system without using message passing primitives.

4.2.1 Lamport’s Bakery Algorithm : [Lamport 1974]

The first distributed algorithm for implementing mutual exclusion was proposed by Lamport. His algorithm is based upon one commonly used in bakeries, in which a customer receives a number upon entering the store and the holder of the lowest number is the next one served. In the algorithm in Figure 4.1, each process chooses its own number. The sites are named 1,2,...,N. So, if two processes choose the same number, then the process with lower identification number goes first.

Two important features of this algorithm are that it allows for a bounded number of process failures and restarts, and the possibility of read errors occurring during an overlapped read and write of the same (shared) memory location.

In the algorithm in Figure 4.1, the statement labeled L2 appears to be redundant at first glance. But it is important to have

State Variables :

integer array choosing[1..N], number[1..N]; {Both initially 0}

Note : 1. The pair (choose[i],number[i]) belongs to the process at site i. P_i may read and write these variables, but P_j, such that $j \neq i$, may only read them.

2. The relation "less than" on ordered pairs of integers is defined by $(a,b) < (c,d)$ if $a < c$, or if $a = c$ and $b < d$.

Local Variable at each site : **integer** j;

Protocol for P_i is -

```
begin
L1: choosing[i]  := 1;
    number[i]    := 1 + maximum(number[1], ... , number[N]);
    choosing[i]  := 0;
    for j := 1 step 1 until N do
    begin
       L2: if choosing[j] ≠ 0 then goto L2;
       L3: if number[j] ≠ 0 and (number[j],j) < (number[i],i) then goto L3;
    end;
    < CRITICAL  SECTION >
    number[i] := 0;
    < NONCRITICAL  SECTION >
    goto L1;
end;
```

Figure 4.1 - Lamport's Bakery Algorithm

it there to preserve mutual exclusion. Assume that a process P_i is in the process of selecting a ticket (i.e. choosing[i] = 1) and another process P_j, such that $j > i$, has selected the ticket and is in the process of finding out if it has the lowest ticket. If L2 were not there, it is possible that P_j would find number[i] = 0 and enter the critical section. Now if P_i selects a ticket whose value is same as that of P_j, P_i would find that it has the lowest ticket (since $i < j$), and therefore enter the critical section at the same time. Thus, L2 makes a process wait if there is another process selecting a value for its ticket at the same time. Any process entering at the time, when other processes have already chosen a value, would select a higher value for its ticket and would cause no danger to the mutual exclusion property of the algorithm.

This algorithm achieves mutual exclusion as can be shown by proving that if process P_i is in its critical section, while another process P_k $(k \neq i)$ is in the 'for' loop (i.e., P_k has calculated number[k]), then the assertion (number[i],i) < (number[k],k) is true and consequently P_k cannot go past L3. The proof given below uses times from P_i's viewpoint.

Let t_{L2} be the time at which P_i· read choosing[k] during its last execution of L2 for $j = k$, and let t_{L3} be the time at which P_i began its last execution of L3 for $j = k$. So $t_{L2} < t_{L3}$. Let t_e be the time just after P_k set choosing[k] to 1, t_w the time at which it finished writing the value of number[k], and t_c the time just after it reset choosing[k] to 0. Then $t_e < t_w < t_c$. Since choosing[k] is equal to zero at time t_{L2}, then either $t_{L2} < t_e$ or $t_c < t_{L2}$. The first case, $t_{L2} < t_e$, implies that $number[k] \geq 1 + number[i]$, which in turn implies that $number[i] < number[k]$, and so the assertion $(number[i],i) < (number[k],k)$ is true. The second case, $t_C < t_{L2}$, implies that $t_W < t_C < t_{L2} < t_{L3}$, which in turn implies that $t_W < t_{L3}$. This means that at t_{L3}, P_i read the current value of number[k]. Since P_i did not execute L3 again for $j = k$, it must have found $(number[i],i) < (number[k],k)$. Hence at most one process can be in its critical section at any given time.

The protocol also avoids deadlock and guarantees fairness. Assume that a process P_k sets choosing[k] to 1, when P_i is past the statement choosing[i]:=0. This means that number[i] contains its current value at the time P_k chooses the current value of number[k]. Therefore, P_k must choose a value such that $number[k] \geq 1 + number[i]$. Hence P_i would enter its critical section before P_k. This protocol therefore implements mutual exclusion on a first-come-first-served basis.

The bakery algorithm is fault-tolerant assuming that when a site fails, it immediately goes to its noncritical section and halts and it is restarted in its noncritical section only. With this assumption, the system continues to operate despite a bounded number of site failures. However, if a process P_i breaks down and restarts an infinite number of times, the system could deadlock. If P_i constantly breaks down as it enters its protocol, then the other processes may always find choosing[i]=1, and hence loop forever at L2.

There is a problem with this algorithm. If there is always at least one process past the statement choosing[i]:=0, the value of number[i] can become arbitrarily large and this could cause overflow errors (depending on the size of memory allocated to an element of the *number* array).

4.2.2 Improvements to Lamport's Bakery Algorithm :

[Hehner 1981] gave a version of Lamport's Bakery algorithm for implementing P and V semaphore primitives. Hehner and Shayamasundar used only one variable (*number*) per process as compared to two (*choosing* and *number*) in Lamport's algorithm.

[Peterson 1983b] proposed an algorithm that keeps every feature, (except first-in-first-out waiting for access to the critical section) of Lamport's algorithm, overcomes Lamport's unlimited growth of the *number* variable, and allows for unbounded process failures and restarts. In fact, Peterson's algorithm uses just four values of (shared) memory per process as compared to Lamport's (shared) variables of unbounded size.

4.2.3 Dijkstra's Self-Stabilizing Distributed Algorithm :

A system is said to be *self-stabilizing* if it can recover from an illegitimate state within a finite number of state transitions. [Dijkstra 1974] proposed a distributed mutual algorithm that has this self-stabilizing property and [Dijkstra 1986] gave a correctness proof for this algorithm.

The algorithm is given in Figure 4.2. Here sites 0,1,2,...,N are assumed to be connected in a ring topology (can be a logical ring structure imposed on the physical network). A site can only exchange information with its neighbors. The decision to enter the critical section by a process (P_i) is made based on its own state variable and that of its left hand neighbor $P_{(i-1)\mathrm{mod}(N+1)}$.

An important point to note is that Dijkstra's solution in Figure 4.2 is not symmetric - the "bottom" machine is differentiated from the other machines by having a different protocol.

The beauty of this algorithm is that even if the system is not properly initialized to a legitimate configuration, the algorithm will drag the system to a legal configuration. The system may not preserve mutual exclusion in the illegitimate states, but once it is

State Variable for Each Machine : nr

Note - 1. In the algorithm below, for site i

L : refers to the state of its left hand neighbor, machine nr.(i-1)mod(N+1),

S : refers to the state of itself, machine nr.i,

R : refers to the state of its right hand neighbor, machine nr.(i+1)mod(N+1).

2. K is an integer such that K > N.

Protocol for Site 0 -
(the "Bottom" machine)

```
L1: if $L \neq S$ then goto L1;
    < CRITICAL SECTION >
    $S := (S + 1) \bmod K$;
```

Protocol for site i -
($i \neq 0$, i.e. other machines)

```
L1: if $L = S$ then goto L1;
    < CRITICAL SECTION >
    $S := L$;
```

Figure 4.2 - Dijkstra's Self-Stabilizing Algorithm

in a legal state, mutual exclusion is preserved.

In this solution, the privilege to enter the critical section moves around the ring, and this causes a major drawback - a process is forced to wait to enter its critical section even when there is no other process attempting to enter the critical section. Also, a process, even when it is not trying to enter its critical section, is forced to execute its exit protocol to pass the privilege to its right hand neighbor.

In the algorithm in Figure 4.2, each machine takes K states, where $K > N$. [Dijkstra 1974] proposed two more mutual exclusion algorithms, with the self-stabilization property, that have machines with three states and four states respectively.

[Kruijer 1979] also gave a self-stabilizing distributed algorithm for sites connected in a tree network topology, instead of the ring network of Dijkstra.

4.3 Solutions which use Message Passing Primitives Explicitly :

The mutual exclusion algorithms in this section are based on explicit communication of messages among processes. The main characteristic of these solutions is that a site does not request another site for its state information. Rather, every time a process changes its state, and if that can affect the global state of the system, it broadcasts information about its new state to other processes on the system. For example, the following state changes would necessitate a broadcast message - from “non-critical section” to “attempting to enter critical section”, from “using critical section” to “exiting critical section”, and from “failed” state to “restarting” state.

[Raynal 1986] characterized these algorithms as “send-information” type as compared to the “request-information” type of algorithms in §4.2. The advantage of “send-information” type algorithms is that communication costs are kept low as it avoids exchange of messages between processes if their states have not changed.

4.3.1 Event Ordering :

In a distributed system, synchronization among processes relies uniquely on establishing an order between events. Since there is no common real physical clock between different sites, this order can be realized only by exchanging messages.

4.3.1.1 Logical Clocks :

Lamport examined the relationship of physical time and event ordering and then defined the *happened-before* relation without using physical clocks [Lamport 1978].

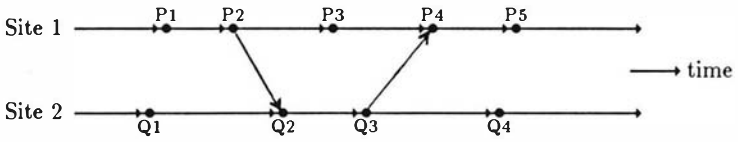
A distributed system can be viewed as a collection of processes and a process as a sequence of events. The definition of an event depends on the application. Execution of a procedure, execution of a single machine instruction, sending or receiving a message are some examples of an event in a process. A single process is defined to be a set of events with an *a priori* total ordering. The ***happened-before*** relation, denoted by →, satisfies the following three properties -

1.) If A and B are events in the same process, and if A is executed before B, then A→B.
2.) If event A is the sending of a message by one process and event B is the receipt of the same message by another process, then A→B.
3.) If A→B, and B→C, then A→C. (Transitivity Property)

It is assumed that A↛A for any event A. The restrictions given above imply that → (***happened-before***) is an irreflexive partial ordering over all system events.

Two distinct events A and B are said to be concurrent if A↛B and B↛A. This order is illustrated in Figure 4.3.

Lamport associated this partial ordering of events with a system of logical clocks which can be implemented by counters.



Local Order of Events :

P1 → P2 → P3 → P4 → P5

Q1 → Q2 → Q3 → Q4

Transitivity :

P1 → P2 → Q2 → Q3 → Q4

Q1 → Q2 → Q3 → P4 → P5

P1 → P2 → Q2 → Q3 → P4 → P5

Exchange of Messages :

P2 → Q2, and

Q3 → P4

Some Incomparable Events :

Q1 and P1, P2, P3

P3 and Q2, Q3, Q4

Figure 4.3 — Example of Partial Ordering of System Events

A *logical clock* C_i for a process P_i is defined as a function which assigns a number C_i<A> to any event A in that process. This number can be thought of as the time at which the event occurred. Therefore, if an event A occurs before another event B, then C<A> < C<B>. But the converse, if C<A> < C<B> then A happened before B, is not necessarily true.

The following two rules are followed to satisfy the *happened-before* relation →, when implementing logical clocks by counters -

1.) The logical clock value is incremented between any two successive events of the same process.

2.) A site, which sends a message m, dates it with a *timestamp* T_m which equals the current value of the logical clock. Upon receiving the message m, the receiver site sets its own clock value greater than or equal to its present value and greater than T_m. The "message reception" event at the receiving site is then dated by this new value of clock. This rule ensures that the time of message reception is later than that of its sending.

[Andre 1985] showed that certain synchronization problems like the *producer-consumer* problem can be resolved by means of partial order only. However, all synchronization problems cannot be solved using a partial order. For example, it is necessary to totally order system events to solve the following problems - the problem of ensuring that there are identical copies of the same item of information at different sites, the problem of equitability, and the problem of introducing priority.

Lamport extended *happened-before* → partial ordering to a strict total ordering by ordering the events by the times at which they occur, and breaking ties by using any arbitrary total ordering of the processes. One such tie-breaking relation used very often is the unique identification number of sites. Then, for two events, A in process P_i and B in process P_j, the total order ⇒ is defined as -

$$A \Rightarrow B \quad \Leftrightarrow \quad (C_i\langle A\rangle < C_j\langle B\rangle) \vee ((C_i\langle A\rangle = C_j\langle B\rangle) \wedge (i < j)).$$

Based on the above definition of total ordering, Lamport gave an algorithm to synchronize events on a first-come-first-served basis, and then applied it to the problem of synchronizing clocks.

Most distributed mutual exclusion algorithms use time stamping to provide fairness in the system.

4.3.1.2 Eventcounts and Sequencers :

[Reed 1979] proposed another synchronization mechanism based on observing and signaling the occurrence of events in the course of an asynchronous computation. Two abstract objects, namely *eventcount* and *sequencer*, are defined for this purpose. An eventcount is an object that counts the number of events in a particular class that have occurred in the execution of the system. Three operations are defined on an eventcount - *advance*, to signal the occurrence of an event associated with a particular eventcount, *await* and *read* to obtain the value of an eventcount. Reed and Kanodia modified Lamport's formalization of time (in §4.3.1.1) as a partial ordering of the events in the system. In their definition, execution of advance, await, and read primitives constitute events.

Synchronization among processes is shown to be obtained from the ability of the eventcount primitives to maintain partial ordering of events, rather than by mutual exclusion. Thus, all processes can be concurrent.

For those cases where a total ordering is necessary, the use of a ticketing operation on a sequencer object is proposed. A *sequencer* S is a non-decreasing integer variable initialized to 0. There is only one operation, called *ticket*(S), that can be applied to a sequencer, and this returns a non-negative integer as its result. Two uses of the *ticket*(S) operation always give different values. Unlike eventcounts, implementation of a sequencer requires some form of underlying mechanism to achieve mutual exclusion.

4.3.1.3 Causal Ordering :

[Birman 1987] proposed a weaker ordering than total ordering and called it causal ordering.

Suppose occurrence of an event Send(M1), corresponding to the site S1 sending M1, and timestamped with logical time T1. Suppose then a second event Send(M2), with timestamp T2, occurring on site S2 after S2 has received message M1. Lamport's logical clocks (in §4.3.1.1) ensure that $T1 < T2$. The "causal timestamping" ensures that event Send(M1) precedes event Send(M2) for every site in the system. This does not say anything about the order in which messages M1 and M2 arrive at any given site in the system. That is, it is possible that a given site gets message M2 before M1, even though event Send(M1) occurs before event Send(M2). However, *causal ordering* of the events Send(M1) and Send(M2) means that every recipient of both M1 and M2 receives messages M1 before message M2.

Causal ordering can be achieved by having every message M, sent by a site, carry every other message sent before M that the site knows of. Causal ordering was first implemented in the ISIS system developed at Cornell University. The advantage of causal ordering in a distributed system is that it is cheaper to realize than total ordering [Joseph 1989].

4.3.2 Previous Work on Distributed Mutual Exclusion Algorithms :

Several algorithms have been proposed to achieve mutual exclusion in distributed systems. These algorithms differ in their communication topology, degree of distribution of control (which is determined by the amount of information a site maintains about other sites), and failure-tolerance. The differences in the algorithms influence the number of messages exchanged and delay incurred per invocation of critical section.

All of the algorithms make some assumptions about the system. The common assumptions are listed below -

- Any site can communicate with any other site.
- The communication subsystem is reliable and therefore there are no transmission errors nor message losses.
- The communication delay is unpredictable, and therefore no assumption is made about the delay between the time a message is sent and received. It is assumed that the delay is finite.
- Each site executes the same algorithm and thus there are N control processes.
- Sites do not crash. A separate failure-recovery mechanism is to be followed.

- There exists a method for serializing multiple requests for mutual exclusion within a site.
- Access to common variables in entry code, exit code, and message handling routines is serialized.

The only assumption where some of these algorithms differ is the order of message delivery between a pair of sites - some assume that messages between any pair of nodes are delivered in the order they are sent, and some do not assume so. This is not a big restriction as ordering between pairs can be implemented in network protocols by having message sequence numbers and message acknowledgements.

In the following discussion, N is the number of nodes in the system.

The distributed mutual exclusion algorithms can be classified into the following two categories -

Category I Solutions : Use a special unique message, called **Token** or **Privilege** message, to obtain mutual exclusion. The privilege to enter the critical section is equated to possession of the token.

Category II Solutions : Do not have any special message to achieve mutual exclusion.

The first algorithm for mutual exclusion (in Category II) was proposed by [Lamport 1978]. In this algorithm, sites maintain logical clocks and all requests to use the critical section are assigned a timestamp. Mutual exclusion is achieved by having a requesting site communicate with all other sites. A site enters the critical section only after it has received a message from every other site, timestamped later than its request message. A first-come-first-served discipline is thus observed by allowing sites to enter the critical section in the order of their request timestamps. A “RELEASE resource” message from a process causes removal of its request from every site’s request queue. This algorithm was later improved in [Ricart 1981] by eliminating the need for the “RELEASE resource” message. Ricart and Agrawala’s algorithm requires $2*(N-1)$ messages per invocation of critical section as compared to $3*(N-1)$ messages in Lamport’s algorithm. In their algorithm, a site intending to execute critical section sends a REQUEST message to all other sites and executes critical section only after it has received a REPLY (permission) message from all other sites. Ricart-Agrawala’s algorithm was further improved in [Carvalho 1983]. In Carvalho and Roucairol’s algorithm, first-come-first-served discipline is not observed - once a site i has received a REPLY message from a site j, site i does not have to ask for site j’s permission to enter the critical section until site i sends a REPLY message to site j, and which can happen only after site j sends a REQUEST message to site i. Thus, site i can enter its critical section more than one time without consulting site j and therefore, the number of messages exchanged per critical section invocation is between 0 and $2*(N-1)$.

Carvalho-Roucairol’s algorithm violated Ricart-Agrawala’s definition of a *symmetric* algorithm, which required an algorithm to have at least one message into and one message out of each site. This opened the door for more algorithms with improvements in terms of the number of messages exchanged and delay incurred per mutual exclusion enforcement.

[Thomas 1979] proposed a majority consensus algorithm to maintain synchronization of multiple copy databases in the presence of update activity. In order to obtain mutual exclusion, a site must obtain permission from a majority of sites in the network. Since there can be only one majority at any given time, mutual exclusion is achieved. Therefore, the number of permission messages required to obtain mutual exclusion is reduced to $\lceil (N+1)/2 \rceil$. The algorithm is robust with respect to lost and duplicate messages and is resilient to both site and communication failures.

[Gifford 1979] presented a weighted-voting algorithm. In his solution, a site can cast more than one vote as compared to one vote in Thomas' algorithm. Therefore, in order to achieve mutual exclusion, it is sufficient to obtain a majority of votes, which may not be from a majority of sites. Gifford's algorithm can be reduced to a centralized algorithm by assigning all votes to one site.

An important property of majority consensus is that the intersection of any two majorities has at least one site in common.

[Maekawa 1985] presented a mutual exclusion algorithm which requires between $3\sqrt{N}$ and $5\sqrt{N}$ messages per mutual exclusion. In his algorithm, a set of sites is associated with each site using the property of finite projective planes, which makes any two such sets in the system have at least one site in common and the size of each of these sets to be $\sqrt{N}$. A requesting site must obtain permission from all sites in the set associated with it. Since this set satisfies the nonnull intersection property with every other set, mutual exclusion is guaranteed.

The assignment of votes and the choice of a set consisting of sets of nodes with nonnull intersection property has a crucial effect on obtaining mutual exclusion and reliability of distributed systems. [Garcia-Molina 1985] studied vote assignments and sets of nodes with pairwise nonnull intersections and showed that these two strategies of obtaining mutual exclusion are not equivalent, though they appear to be so. Garcia-Molina and Barbara proposed the notion of a *coterie*, which is a set of groups, where group is a set of nodes, with the property that any two members of a coterie (i.e. groups) have at least one common node. Coteries are shown to be more powerful than vote assignments by proving that there are coteries such that no vote assignments correspond to them. Maekawa's sets can be considered as a special case of a coterie where each group is of same size.

[Agrawal 1991] proposed another distributed mutual exclusion algorithm based on the notion of coteries. The communication network is assumed to be logically organized into a tree and intersecting quorums are formed by selecting paths starting from the root and ending with any of the leaves. In case of failure or inaccessibility of a site, the algorithm substitutes for that site two paths, both of which start with the children of that site and terminate with leaves. A tree quorum cannot be formed if any of the leaf nodes is inaccessible. For a tree with each nonleaf node having d children, it is shown that, in the best case, when there are no failures, $\lceil \log_d N \rceil$ sites are necessary to form a tree quorum. The worst case would be when $(N - \log_d N)$ sites fail and then the size of tree quorum is shown to be equal to $\lceil((d-1)N+1)/d\rceil$.

In Maekawa's protocol, only one set is associated with each site, and therefore failure of any site in the associated set of a site prevents that site from accessing the critical section. Agrawal and Abbadi's scheme provides several alternative sets to a site and is therefore resilient to failures. They claimed their algorithm to be the first distributed mutual exclusion protocol which tolerates both site and network partitioning and requires O(log N) messages in the best case.

In the **token-based** algorithms, the site possessing the token has the privilege to access the critical section. Since there is only one token in the system, only one site can possess it at any given time, and therefore, mutual exclusion is achieved as this site will be the only one executing the critical section. One of the earliest token-based mutual exclusion algorithm is by Lelann. He assumed the sites to be connected in a ring network and the token to be circulating on this ring of sites. A site is required to capture the token before entering critical section [Silberschatz 1991].

[Suzuki 1985] and [Ricart 1983] presented token-based mutual exclusion algorithms which require at most N message exchanges for one mutual exclusion invocation. A site possessing the token can enter the critical section without taking permission from any other site and therefore, no message exchanges are involved in this case. If a requesting site does not have the token, it sends a REQUEST message to all other sites. When the site holding the token receives the REQUEST message, it transmits the token to the requesting site when it no longer needs the token. Thus, at most N messages are required per mutual exclusion invocation - $(N-1)$ REQUEST messages and 1 for transmission of the token.

This approach was improved in [Singhal 1989]. In Singhal's algorithm, each site maintains information about the state of other sites. This state information is used to guess the sites which could be holding the token. A site intending to enter its critical section, sends REQUEST messages to these probable token holding sites only, and not to all other sites. Thus, the number of messages exchanged is between 0 and N per each execution of the critical section. In fact, the basic idea of this algorithm is very similar to the improvement made by Carvalho-Roucairol over Ricart-Agrawala algorithm. Use of a token to grant the privilege to enter the critical section saves $(N-1)$ permission (REPLY) messages over Carvalho-Roucairol's algorithm and incurs the extra cost of one message to transmit the token message.

[Raymond 1989] proposed another token-based algorithm which uses a spanning tree of the interconnection network topology. In Raymond's algorithm, a site communicates to its neighbors only and therefore it does not have to be aware of complete network topology. Because of the spanning tree topology, there exists a unique path from each site to the site holding the token. The request messages and the token travel along this path. For this algorithm the average number of messages exchanged per mutual exclusion invocation is

O(logN).

[van de Snepscheut 1987] presented a similar tree-based algorithm and then extended the solution to the case in which the network is an arbitrary connected graph. He showed his mutual exclusion algorithm for a general graph to be fair.

In the algorithms by Raymond and van de Snepscheut, the internal nodes in the tree receive and send a higher number of messages compared to the leaf nodes.

Helary, Plouzeau, and Raynal presented a token-based algorithm in a network with an *a priori* unknown topology. All other mutual exclusion solutions are based on *a priori* known topology - complete, ring, tree, etc. [Helary 1988]. In their algorithm, a request is propagated in the network with a flooding broadcast (wave) technique - a site on receiving a request from one of its neighbors propagates it to its other neighbors. The path followed by a request from a requesting site to the token owner is marked. The token is transmitted along that path in the opposite direction to reach the requesting site.

These token-based algorithms suffer from a major drawback - if the token is lost, the critical section cannot be reached by any site. To preserve mutual exclusion, it is necessary that the token is regenerated by only one site. [Garcia-Molina 1982] presented election algorithms that may be used for recovery from failures. [Nishio 1990] presented a token-based mutual exclusion algorithm which has failure detection and recovery from failures as an integral part of the algorithm.

[Sanders 1987] introduced the concept of "**information structures**" as a unifying framework for different distributed mutual exclusion algorithms. The information structure describes which processes maintain state information about other processes and from which processes permission must be requested before entering the critical section. Information structures can be either static or dynamic.

A comparison of some of these algorithms, in terms of the number of messages exchanged per critical section invocation, the logical structure imposed on the physical network topology, the number of nodes about which each node keeps static information, the number of nodes about which each node keeps dynamic information, and the kind of information structure (static or dynamic) used, is given in Table 4.1. It also lists whether the algorithm is token based or not, and whether it assumes that messages are delivered in the order they are sent or not.

Once the idea behind an algorithm is clear, it is easier to give the details of the actual implementation of the algorithm. The following four algorithms are chosén as representative and their details are given below - (1)Ricart-Agrawala's Algorithm, (2)Suzuki-Kasami's Algorithm, (3)Maekawa's Algorithm, and (4)Raymond's Algorithm.

Algorithm	Number of Messages Exchanged	Network Topology Assumed (Logical)	Token Based	Messages received in the order sent	Static Information at each node (in # of nodes)	Dynamic Information at each node (in # of nodes)	Type of Information Structure Used
Lamport	$3 * (N-1)$	Complete	×	✓	N	N	Static
RA[1]	$2 * (N-1)$	Complete	×	×	N	N	Static
CR[2]	0 to $2*(N-1)$	Complete	×	×	N	N	Dynamic
Maekawa	$3\sqrt{N}$ to $5\sqrt{N}$	FPP[3]	×	✓	$\sqrt{N}$	$\sqrt{N}$	Static
SK[4]	0 or N	Complete	✓	×	N	N	Static
Raymond	O(log N)	Tree	✓	×	Neigh.[5]	Neigh.[5]	Static
Singhal	0 to N	Complete	✓	✓	N	N	Dynamic
SN[6]	O(log N)	Tree	✓	×	Neigh.[5]	Neigh.[5]	Static
Agrawal	O(log N)	Tree	×	✓	$\log_d N$[7]	$\log_d N$[7]	Static
HPR[8]	–	Unknown	✓	×	Neigh.[5]	Neigh.[5]	Static

Table 4.1 - A Comparison of Some Distributed Mutual Exclusion Algorithms

[1] Ricart and Agrawala Algorithm

[2] Carvalho and Roucairol Algorithm

[3] Logical structure imposed by finite projective planes

[4] Suzuki and Kasami Algorithm

[5] Neighbors

[6] van de Snepscheut Algorithm

[7] d is the degree of a node

[8] Helary, Plouzeau, and Raynal Algorithm

4.3.2.1 Ricart-Agrawala Algorithm : [Ricart 1981]

The algorithm is given in Figure 4.4. The sequence number concept used here implements Lamport's logical clocks. The algorithm implements a first-come-first-served discipline for entry to the critical section. It is achieved through the virtual ordering among requesting nodes formed by the sequence numbers and node numbers. A site enters its critical section only after it has received a REPLY message for its REQUEST message from all other sites. A site upon receiving a REQUEST message updates the value of its Highest_Sequence_Number, and then sends a REPLY message to the requesting node if it has not requested the critical section for itself or if the requesting node made a request to enter the critical section before it did.

Therefore, **total number of messages exchanged** $= \mathbf{(N-1)Request + (N-1)Reply}$

$= \mathbf{2*(N-1).}$

4.3.2.2 Suzuki-Kasami Algorithm : [Suzuki 1985]

In this algorithm, a PRIVILEGE message is used to determine the (privileged) node which can enter the critical section. A node requesting the privilege sends a REQUEST message to all other nodes. A node receiving the PRIVILEGE message is allowed to enter its critical section repeatedly until the node sends PRIVILEGE to some other node. The algorithm is given in Figure 4.5.

A REQUEST message from a site contains site's identification number and a sequence number indicating the number of times the site

Shared Variables (Information Held by Each node) :

```
CONSTANT
   me,     {This node's unique identification number}
   N;      {The number of nodes in the network}
INTEGER {Variable list begins here}
   Our_Sequence_Number,     {Sequence number chosen by a request made here}
   Highest_Sequence_Number,{The highest sequence number seen in any
                            REQUEST message sent or received. Initially 0}
   Outstanding_Reply_Count;{Number of REPLY messages expected}
BOOLEAN
   Requesting_Critical_Section{Initially False; True when this node is requesting
                            access to the critical section}
   Reply_Deferred [1..N];   {Initially False; Reply_Deferred[j] is True when
                            this node defers REPLY to j's REQUEST message}
BINARY SEMAPHORE
   Shared_vars;  {Initially 1; To interlock access to the above shared variables}
```

Process Which Invokes Mutual Exclusion For This Node :

```
   P(Shared_vars);
     Requesting_Critical_Section := True;
     Our_Sequence_Number := Highest_Sequence_Number + 1;
   V(Shared_vars);
   Outstanding_Reply_Count := N − 1;
   For j := 1 To N Do
     If j ≠ me then Send_Message(REQUEST(Our_Sequence_Number,me), j);
   Waitfor (Outstanding_Reply_Count = 0);
   < CRITICAL  SECTION >
   Requesting_Critical_Section := False;
   For j := 1 To N Do
     If Reply_Deferred[j] then
        Begin  Reply_Deferred := False;
               Send_Message(REPLY, j);
        End;
```

Process Which Receives Request(k,j) Messages : (Defer_it is a local variable)

```
   Highest_Sequence_Number := Max(Highest_Sequence_Number,k);
   P(Shared_vars);
     Defer_it := Requesting_Critical_Section AND ((k > Our_Sequence_Number)
                 OR (k = Our_Sequence_Number AND j > me));
   V(Shared_vars);
   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;
```

Figure 4.4 — Ricart-Agrawala Algorithm

has requested the critical section invocation. Each site maintains an array RN (of size N) for recording the largest sequence number ever received from each one of the other nodes.

The PRIVILEGE message contains a queue of requesting nodes and an array LN (of size N) for recording the number of times each site has entered the critical section. When a site finishes executing the critical section, the LN entry for that site in the PRIVILEGE message is updated, and all new requesting sites are appended to the queue. The next node to get the PRIVILEGE is the one at the head (front) of the queue.

The algorithm requires, at most N message exchanges per one mutual exclusion invocation - $(N-1)$ REQUEST messages and 1 PRIVILEGE message, or no message at all if the node having the PRIVILEGE is the only requesting node in the system.

4.3.2.3 Maekawa's Algorithm : [Maekawa 1985]

In this algorithm, each site i in the system has a set S_i of sites associated with it such that any two such sets S_i and S_j have at least one node in common. The problem of finding a set of S_i's is equivalent to finding a finite projective plane of N points [Maekawa 1985]. The size of each set S_i is found to be $\sqrt{N}$.

Shared Variables (Information Held by Each Node) :

```
Const
   I : Integer;          {the identifier of this node}
Var
   HavePrivilege, Requesting : Boolean;
   {Initially HavePrivilege=true in node 1 only; and Requesting=false initially}
   j, n : Integer;
   Q : Queue of Integer;   {initially empty}
   RN, LN : Array [1..N] of integer; {Initially RN[j]=LN[j]= $-1$, $\forall$j=1,...N}
```

Note: Request Message Handler is executed indivisibly whenever a Request arrives.

Process Which Invokes Mutual Exclusion For This Node :

```
begin
   Requesting := true;
   if not HavePrivilege then
     begin
        RN[I] := RN[I] + 1;
        for all j in {1,2,...,N} – {I} do
           Send Request(I,RN[I]) to node j;
        Wait Until PRIVILEGE(Q,LN) is received;
        HavePrivilege := true;
     end;
   < CRITICAL SECTION >
   LN[I] := RN[I];
   for all j in {1,2,...,N} – {I} do
     if not in (Q, j) and (RN[j] = LN[j] + 1) then Q := append(Q, j);
   if  Q $\neq$ empty then
     begin
        HavePrivilege := false;
        Send PRIVILEGE(tail(Q), LN) to node head(Q)
     end;
   Requesting := false
end;
```

Process Which Receives Request(j, n) Messages : {executed indivisibly}

```
begin
   RN[j] := max(RN[j], n);
   if HavePrivilege and not Requesting and (RN[j] = LN[j] + 1) then
     begin
        HavePrivilege := false;
        Send PRIVILEGE(Q, LN) to node j
     end
end;
```

Figure 4.5 – Suzuki-Kasami Algorithm

The algorithm is given as -

1. When site i wants to enter critical section, it sends a REQUEST message to every member of S_i. The REQUEST message contains site's identification number and a timestamp.

2. Upon receiving a REQUEST, a member node of S_i makes itself "locked" for the REQUEST, if it is not currently locked for another REQUEST, and then returns a LOCKED message to the requesting node i. If the node is locked for a REQUEST from another node, site i's REQUEST is placed in the WAITING QUEUE of the node. It is then tested to determine whether the current locking REQUEST or any other outstanding REQUEST in the Queue at the node precedes the received REQUEST. If so, a FAILED message is returned to node i. Otherwise, an INQUIRE message is sent to the node originating the current locking REQUEST to inquire whether this originating node has succeeded in locking all its members. If an INQUIRE has already been sent for a previous REQUEST and its reply has not yet been received, it is not necessary to send INQUIRE again.

3. When a node receives an INQUIRE message, it returns a RELINQUISH message if it knows that it will not succeed in locking all its members, that is, it has received a FAILED message from some of its members. This RELINQUISH message relinquishes the member node to a more preceding request and thus deadlock is avoided. The node sending the RELINQUISH message cancels the LOCKED message previously received from the member node. If an INQUIRE message arrives before it is known whether the node will succeed or fail to lock all its members, a reply is deferred until this becomes known. If an INQUIRE message arrives after the node has sent a RELEASE message, it is simply ignored.

4. When a node receives a RELINQUISH message, it relieves itself of the current locking REQUEST, and then locks itself for the most preceding REQUEST in the WAITING QUEUE. A LOCKED message is then returned to the node originating the new locking REQUEST.

5. If all members of S_i have returned a LOCKED message, node i enters its critical section.

6. Upon completing the critical section, node i sends a RELEASE message to each member of S_i.

7. When a node receives a RELEASE message, it relieves itself from the current locking REQUEST. It deletes this locking REQUEST and then relocks itself for the most preceding REQUEST in the WAITING QUEUE, if the Queue is not empty. A LOCKED message is returned to the node originating the new locking REQUEST. If the Queue is empty, the node marks itself unlocked.

In case of light demand for the critical section, the algorithm requires $3\sqrt{N}$ messages per critical section invocation - $\sqrt{N}$ REQUEST messages, $\sqrt{N}$ LOCKED messages, and $\sqrt{N}$ RELEASE messages. Under heavy demand, a new REQUEST will most likely fail to lock its destination node and therefore, a total of $4\sqrt{N}$ ($\sqrt{N}$ REQUEST, $\sqrt{N}$ FAILED, $\sqrt{N}$ LOCKED, and $\sqrt{N}$ RELEASE) messages are required per mutual exclusion. The worst case is when a new REQUEST is initiated from a node that has neither requested mutual exclusion nor participated in the algorithm as a member node for a certain period. It then causes an INQUIRE message to be sent, for which a RELINQUISH message is returned. Thus, a total of $5\sqrt{N}$ ($\sqrt{N}$ REQUEST, $\sqrt{N}$ INQUIRE, $\sqrt{N}$ RELINQUISH, $\sqrt{N}$ LOCKED, and $\sqrt{N}$ RELEASE) messages are required to obtain mutual exclusion.

4.3.2.4 **Raymond's Algorithm** : [Raymond 1989]

In this algorithm, the communication network is assumed to be a spanning tree of the actual network topology. Each node communicates with only its neighboring nodes in the spanning tree and holds information pertaining only to those neighbors. There exists a PRIVILEGE message in the network; a site must possess this PRIVILEGE message in order to enter its critical section. The complete algorithm is given in Figure 4.6.

Each node has a variable HOLDER that stores the location of the privilege relative to the node itself. Because of the spanning tree network topology, a unique directed path exists from a non-

Shared Variables (Information Held by Each node) :

HOLDER: Values = "self" or the name of one of the immediate neighbors. Indicates the relative position of the privileged node with respect to the node itself.

USING: A Boolean Value. USING indicates if the node itself is currently executing the critical section.

REQUEST_Q: A first-in-first-out queue. Possible elements are the names of immediate neighbors and "self". It holds the name of those nodes that have sent a REQUEST but have not yet got the PRIVILEGE.

ASKED: A Boolean Value. It is true when a nonprivileged node has sent a REQUEST message to its HOLDER value (=name of a node).

Process Which Makes Request (MAKE_REQUEST Process) :

```
if HOLDER ≠ self ∧ REQUEST_Q ≠ empty ∧ not ASKED
then begin
   Send REQUEST to HOLDER;
   ASKED := true;
end;
```

Process Which Sends PRIVILEGE message (ASSIGN_PRIVILEGE Process) :

```
if HOLDER = self ∧ not USING ∧ REQUEST_Q ≠ empty
then begin
   HOLDER:= dequeue(REQUEST_Q);
   ASKED  := false;
   if HOLDER = self
      then USING := true
      else Send PRIVILEGE to HOLDER;
end;
```

Node Wishes to Enter the Critical Section :

```
enqueue(REQUEST_Q, self); {If this is the privileged node then Assign_
ASSIGN_PRIVILEGE; Privilege will allow this node to enter the critical section.
MAKE_REQUEST;     Otherwise, it makes a REQUEST to obtain the privilege.}
```

Node Receives a REQUEST Message From Neighbor X :

```
enqueue(REQUEST_Q, X);  {If this node is the holder then Assign_Privilege
ASSIGN_PRIVILEGE; may send the Privilege to the requesting node.  Otherwise,
MAKE_REQUEST;     it propagates the Request to obtain the privilege.}
```

Node Receives a PRIVILEGE Message :

```
HOLDER := self;             {Assign_Privilege may pass the privilege to
ASSIGN_PRIVILEGE; another node. And then, Make_Request may request
MAKE_REQUEST;     that the privilege be returned.}
```

Node Exits the Critical Section :

```
USING := false;             {On releasing the critical section, Assign_
ASSIGN_PRIVILEGE; Privilege may pass the privilege to another node and
MAKE_REQUEST;     may then request it back through Make_Request}
```

Figure 4.6 — Raymond's Algorithm

privileged site to the privileged site. When a nonprivileged node wishes to enter the critical section, it sends a REQUEST message to the holder of the PRIVILEGE message, as viewed by it. Upon receipt of a request message, a nonprivileged node on this unique path then makes a request to its "believed" holder if a request was not already made by it for itself or on behalf of some other node. In the algorithm, a variable (ASKED) is used to find out if a request was already made by the node. Thus the number of request messages made is reduced.

A node can transmit the PRIVILEGE message only if it holds the PRIVILEGE but not be using it, and the oldest request for the privilege came from another node. The PRIVILEGE is transmitted using the same path as used by the REQUEST message but in the opposite direction.

There are four events that can alter the assignment of privilege and/or necessitate the sending of a REQUEST message - node wishing to enter the critical section, node exiting the critical section, the receipt of a REQUEST message by a node, and the receipt of the PRIVILEGE message.

The upper bound for the number of messages exchanged per critical section is $= 2*D$, where D is the diameter (longest path length) of the tree. The worst possible topology for this algorithm is a straight line arrangement, since the diameter of such a topology is $N-1$. The best topology for this algorithm is a radiating star formation. The diameter of such a topology, with k as the valence of each nonleaf node, is given by $2 * \lceil \log_{k-1} (\frac{(N-1)(k-2)}{k} + 1) \rceil$.

Thus the worst case for this topology is $O(\log_{k-1} N)$.

4.4 Summary :

The availability of such a variety of distributed mutual exclusion algorithms is a good evidence in itself of the nontrivial nature of the problem and the crucial role it plays in distributed systems. A distributed system designer would have to be very careful in selecting the "right" algorithm. Some of the factors to consider include network topology, reliability, cost (efficiency), and extensibility.

In the next chapter, a new distributed mutual exclusion algorithm is developed by finding solutions to some real-life situations which require mutual exclusion.

CHAPTER V

A NEW DISTRIBUTED MUTUAL EXCLUSION SOLUTION DERIVED FROM REAL-LIFE EXAMPLES

5.1 Introduction :

An extensive amount of work has been done to solve the problem of mutual exclusion in distributed systems. Chapter IV discussed all the available distributed mutual exclusion algorithms. This chapter presents some new solutions to achieve mutual exclusion in a distributed system when there is only one shared resource and also when there are M (≥ 1) identical instances of the resource. These solutions are obtained by considering real-life situations where mutual exclusion is required. Some of the solutions discussed in Chapter IV appear here again; they have been tailored to suit our real-life examples.

The examples used through out this chapter are -

Example 1 - Consider the situation when a book (resource) is shared among N persons (sites). For convenience, assume their names to be 1 through N such that they are unique and $1 < 2 < \cdots < N$ (think of lexicographic sorting). No two persons can read (use) the book at the same time. The only way to find out if anybody is using the book is through exchange of messages. Everybody can talk to (communicate with) everyone else with the condition that communication between any two-persons is limited to exchange of postcards (messages) only. It is assumed that a postcard always reaches its destination without any changes to its contents. But a postcard from one person may take any amount of time to reach another person (Postal delays are possible!).

The assumptions made in the above example fit a distributed model.

Example 2 - An extension to the first example is when there are M (> 1) copies of the same book. Considering M ≥ N, i.e., when there are at least as many books available as the number of persons, is of no interest as each person then could have a personal copy of the book without any trouble. Therefore, we assume M < N, that is, at most M persons could be reading the book at the same time. Other assumptions are as made in the first example.

One such real-life situation is seen everyday in a bank where tellers provide service to customers. We will have to modify the actual situation a little bit to fit a distributed model. Some of the assumptions to be made are - customers enter from different doors, they cannot see each other and communicate through messages only, and a customer is not allowed to turn back to ask the teller a quick question once he/she is left the window.

In the following discussion, informal language (as in the first example) is used. This can easily be replaced with formal terminology. The words "call" and "call back" are used only for better understanding of the problem; they don't imply immediate delivery of the message.

5.2 Search for Distributed Mutual Exclusion Solutions :

5.2.1 In Case of One Shared resource :

Initially, assume that the book is lying at a place known to everyone and everybody replaces the book back at that place after using it.

A very simple and intuitive solution is -

Any person needing the book "calls" everybody else to inform "I need the book". On receiving the "call", a person answers back one of the following three things - "Go ahead", or "I am using it. I will call you back when I am done", or "I also want the book and so I will call you back when I am done". When the person currently using the book is finished reading it, he "calls back" all the "callers" to say "I am done. Go ahead and use the book". A person receiving "Go ahead" from everybody else can be sure that it would be then safe to use the book.

The "I will call you back" message needs to reach a person before "Go ahead" message to avoid confusion. So assume that messages are delivered in the order they are sent. This restriction will be removed later on.

This solution will work if not more than one person needs to use the book at the same time (formally, when there are no concurrent requests to use the shared resource); otherwise it will not work. For example, assume persons i and j need to use the book at the same time. Also assume that they have got "Go ahead" from everyone else but from each other. Now, person i would wait for j to "call back" and j would wait for i to "call back" and it will never happen. This problem stems from the "selfish" approach in the solution. To avoid it, we introduce some arbitration scheme in the protocol. One such rule is to let the book go to the person who asked for it first. It can be implemented using time of the "call" and person's name (it is possible to have two persons to have exactly the same time even if they have different watches and therefore names having a lexicographic ordering are used to break the ties. This is similar to Lamport's logical clocks [Lamport 1978]).

Assume that a person marks the same time (actually time of the "first call") on all the "calls" made for each use of the book. By incorporating the arbitration rule into the protocol, person i requesting for the book would give "Go ahead" to another person j if i sees that j had started asking for the book before he did; otherwise, i "calls back" j to say "Sorry, you will have to wait since I asked for the book before you. I will call you later when I am done".

The above protocol guarantees exclusive access to the book and it can be shown that it is free from deadlocks and starvation.

Since the cost of a distributed algorithm is generally determined by the number of message exchanges, we determine for the above protocol the total number of message exchanges for each use of the book . It requires -

In the best case, when nobody is using the book -
$(N-1)$ "Call" (request) messages and $(N-1)$ "Go ahead" messages. And therefore, a total of $2*(N-1)$ messages per use of the book.

In the worst case, when someone is using the book and everybody else had already started asking for it -
$(N-1)$ "Call" (request) messages, $(N-1)$ "I will call back when I am done" messages, and $(N-1)$ "Go ahead" messages. And therefore, a total of $3*(N-1)$ messages per use of the book.

Improvements :

This protocol has been improved upon (in terms of number of message exchanges) in the literature (except [Lamport 1978]). [Ricart 1981] improved upon it by eliminating "I will call back when I am done" messages. In Ricart-Agrawala algorithm, a person can defer the reply if he either is using the book or made the "call" before the other requesting person (since the protocol is to be used with processes and not persons, one can afford to be discourteous!). This saves (N − 1) messages and therefore their algorithm requires a total of 2∗(N − 1) messages to use the book.

[Carvalho 1983] further improved it by reducing the number of "call" (request) messages and thereby the number of "go ahead" messages. The reduction is achieved by having a person assume for next requests "go ahead" from the persons that sent "go ahead" for the current request.

The number of requests and "go aheads" are also shown to be reduced by forming logical groups according to some rule in quorum-based algorithms [Maekawa 1985; Agrawala 1991; Garcia-Molina 1985]. These algorithms require a person to "call" other persons in his group and obtain "go ahead" from them only.

The number of "go ahead" messages is reduced from (N − 1) to 1 by letting the book always stay with a person unless it is in transit; that is relax the initial assumption of replacing the book back at the previously known place. So, a person, after using the book, may pass it to one of the requesting persons or keep it if no one has asked for it. Thus, "go ahead" messages are replaced with actually passing the book.

This new assumption corresponds to a situation where the shared resource is passed among processes. Since this is not physically possible, a special "token" message is introduced and possession of the token is assumed to be equivalent to possession of the shared resource. For simplicity, we will stick to our assumption of circulating the shared resource.

So with this assumption, if the person holding the book needs to use it, that person can "go ahead" without asking anyone. But if a person does not have the book and wants to use it, then he would have to "call" everyone else as the identity of the person holding the book is not known to anyone. Since it is allowable to be discourteous, one does not reply to a "call" if one does not possess the book. The person with the book replies by passing the book, after using it, to the person who requested for it first. Therefore, this protocol requires (N − 1) "call" messages to make sure that the request reaches the "right" person (one with the book) and one more to pass the book.

Again, this protocol will work only if there are no concurrent requests to use the book. [Suzuki 1985] gave a similar algorithm and handled concurrent requests by having the book carry a list of persons who need to use the book. The person holding the book updates this list by removing from it the name of the person to whom the book will be passed and adding the names of the persons who requested for the book but their names are not on the list (to avoid duplicate names). Thus, the number of message exchanges is reduced to 0 or N per use of the book.

Objective :

The goal in this chapter is to design a protocol which further reduces the number of message exchanges. Since there is no scope left to reduce "go ahead" type of messages (they have been reduced to 1 in the above protocol), the aim is to reduce the number of "calls" (requests) a person has to make to get the book.

[Singhal 1989] reduced the number of request messages by introducing asymmetry. In Singhal's algorithm, the initial configuration is such that the person named N is required to ask persons 1 through $N-1$, person named $N-1$ is required to ask persons 1 through $N-2$, and so on to person 1 who does not need to ask anyone. This forms a step-ladder arrangement of persons. Asymmetry is maintained by letting people go up and down this ladder. The person at the bottom of the ladder is the one who possesses the book. [Raymond 1989] and [van de Snepscheut 1987] reduced the number of request messages by imposing a tree structure arrangement on people.

The number of request messages can be reduced to one if everyone at any given time knows the name of the person who possesses the book. We reduce the number of request messages to **at most $(N-1)$** by using a heuristic which helps in determining the location of the book. This heuristic is developed through a series of protocols and the next section gives a description of them.

5.2.1.1 Informal Description of the New Algorithm's Development :

The objective is to reduce the number of "calls" (requests) one has to make before getting the book and thereby reduce the total number of message exchanges.

We will start from the protocol in the last section where the book is held by the person who uses it last. Assume that, at the start, the book is given to the person named 1 and this fact is known to everyone in the system.

Protocol 1 - A simple idea is that the person with the book informs everyone of the name of the person to whom he is going to pass the book. Since everybody always knows the name of the person who has the book, anyone needing the book has to make only 1 "call" to the person with the book. So the number of "call" (request) messages is reduced to 1 by introducing $(N-2)$ "inform" messages (the person passing the book and the person going to get the book do not need to be informed).

The "inform" messages are not only an overhead, but also a source of new problems. It is likely that everybody had already made a request to use the book before they got the information about new holder of the book. So there is a risk of $(N-2)$ "calls" going waste. Another serious problem with this protocol is that a person may just end up chasing the book. This is more likely to happen when there is a heavy demand for the book, but can happen otherwise as no assumption is made about the delay between the time a message is sent and received.

The cause of all the problems in the above protocol is the transmission of "inform" messages, containing the name of the new holder of the book, to everyone. So we remove the broadcast of "inform" message and introduce "inform on request" with short-term memory.

Protocol 2 - Consider the case when person i is going to pass the book to person j. Person i, instead of broadcasting j as the new holder of the book, remembers the name j as the one with the book. Person i then informs the next "caller" (the person whose request is received next) that "Sorry, I gave/am giving the book to j. So call j now" and changes its (short-term) memory value from j to the name of this "caller". So the next "caller" is told to call the previous caller, and so on. This protocol thus requires a person to remember the name of the person who he thinks has the book. Since postal delays are possible, it is likely that a person has not yet received the book and there is already a "call" waiting to be serviced (that is, someone has already asked the book back from that person). It is also possible that while a person is using the book somebody else asks him for it. If it is the first "call" received by that person, then both of the above situations can be handled by having a person (i)tell the "caller" that he will pass the book after using it, and (ii)remember the name of the "caller" as the one with the book; otherwise, it is handled as described before.

An example to explain this protocol is given below -

Assume that 4 people (1, 2, 3, and 4) share a book and at start the book is with 1. Consider the situation when 2, 3, and 4 need the book. So, all three of them "call" 1. Assume 1 receives the requests in the order 3, 2, and 4. So 1 is ready to pass the book to 3. 2 is told to call 3, and 4 is told to call 2. Now, 2 "calls" 3, and 4 "calls" 2 to ask for the book. Assume 4's request reaches 2 before 2 has got the book. So, 2 "informs" 4 to wait until he is done. Now, 3 finishes using the book and passes it to 2, and then 2 starts using it. In the meantime, 1 and 3 decide to use the book again. 2 has finished reading the book and so 2 passes it to 4. Since 1 thinks the book is with 4 and 3 considers it to be with 2, 1 “calls” 4 and 3 “calls” 2 respectively. On getting 3’s call, 2 “informs” 3 to “call” 4 . Assume 1’s message has still not reached 4 where as 3 has found out that 2 does not have the book and it may be with 4. So 3 “calls” 4 and assume it reaches 4 before 1’s request does. Therefore, 4 passes the book to 3 and then on receiving 1’s call “informs” him to “call” 3. When 3 receives 1’s call, he passes the book to 1 after using it , and then 1 can use it.

The final state of the system is - 1 has the book, 2 thinks 4 has the book, 3 thinks 1 has the book, and 4 thinks 1 has the book.

A careful look shows that the system begins with a directed star topology (everybody knows 1 has the book) and the second protocol tries to maintain it. If a directed star topology is maintained, only 1 “call” is needed to get to the person with the book. The example given above shows that the second protocol does not accomplish such a topology always - if 2 wanted to use the book, when the state of the system is as given at the end of the example, 2 will have to call 4 and 1 in this order to get the book provided no other requests are created. However, the protocol could be modified such that it always maintains a directed star topology. The change would be - whenever a person changes his value of the variable that holds the name of the person considered by this person to be the current holder of the book, he “informs” the person, from whom he got the book, of this change. A person on receiving such information then sets his value to the name contained in that message. By doing this, only 1 “call” message is required and possibly 1 “call” message is wasted because of unpredictable message delays. But this is achieved at the cost of more “inform” messages. Since the ultimate goal is to reduce the total number of message exchanges per use of the book, this approach is abandoned.

The third protocol given below accomplishes a reduction by being both “discourteous” and “helpful”.

__Protocol 3__ - In the second protocol, on receiving a “call”, a person not having the book informs the “caller” who to “call” (as he sees it) to get the book. These “inform” messages increase the total number of messages, and therefore, an attempt to eliminate them is made here.

Ideas from [Ricart 1981] and [Raymond 1989] are used to get rid of “inform” messages and still be able to maintain an approximate directed star topology. So, a person not having the book does not “call back” a “caller” (discourteous approach from [Ricart 1981]) to provide information about who to call to get the book; rather, he forwards the “call” (helpful approach from [Raymond 1989]) on behalf of the “caller” to the person who he thinks has the book and changes his value to contain the name of the caller as the new holder of the book for handling future requests. The other rules remain the same as those in the second protocol.

In the next section, a formal description of this protocol is given. It is shown to be both deadlock-free and starvation-free. The cost of the algorithm is shown to be between 0 and N messages per use of the book.

5.2.1.2 Formal Description of Protocol 3 :

The following assumptions are made in this mutual exclusion algorithm for a distributed system consisting of N nodes -

(1)any two nodes can communicate with each other,
(2)messages are neither lost nor changed,
(3)messages may be delivered out of order, and
(4)there are no failures. (Recovery from failures is considered separately in §5.2.1.3)

It is assumed that there exists a special privilege message, called **token**, in the system. A site can execute its critical section only if it possesses the token. The site holding the token is referred to as the privileged site.

The complete algorithm is given in Figure 5.1.

Initialization - The token is initially assigned to site 1. Therefore, initially holder_as_I_see_it is set to "self" for site 1 and 1 for all other sites, and have_token is true at site 1 only and false at all other sites.

It is assumed that there are no requests at system start-up, and therefore, initially requesting_CS and using_CS are false, and who_to_pass_token is set to none for all the sites.

The Algorithm - Each node has three processes - one for invoking mutual exclusion, one for handling receipt of request messages, and one for handling receipt of token message. These three processes execute in local (within a node) mutual exclusion which can be implemented using a shared memory mutual exclusion solution, such as semaphores, monitors, etc. However, **wait** and **execution** of the

Shared Variables (Information Held by Each node) :

Holder_as_I_see_it: Values = "self" or the name of one of the nodes.
Indicates the current holder of the token as viewed by this site.
Initially, site 1's value is "self" and all other sites' value = 1.

Using_CS: A Boolean Value. Using_CS indicates if the node itself is currently executing the critical section. Initially False for all the sites.

Have_token: A Boolean Value. Initially true at site 1 and false at all other sites.

Requesting_CS: A Boolean Value. True when a node is requesting access to the critical section. Initially false at all the sites.

Who_to_pass_token: Values="none" or name of one of the nodes. Indicates the node to whom the token is passed next by this site. Initially, its value = "none" at all the sites.

Process Which Invokes Mutual Exclusion for this node i :

```
who_to_pass_token := none; {There cannot be a request pending}
requesting_CS := true;
if not have_token then begin
   Send Request(i) to holder_as_I_see_it; {Send a request message containing its
   holder_as_I_see_it := "self";           name to the node it thinks has token}
   Wait Until have_token = true;           {Wait is interruptible}
end;
using_CS := true;
< CRITICAL SECTION >                       {Can handle request messages here}
requesting_CS := false;   using_CS := false;
if who_to_pass_token $\neq$ "none" then begin  {Transmit the token to the site which
   Send token to who_to_pass_token;        requested for it when this site was using
   have_token := false;                    or waiting to use its critical section}
end;
```

Process Which Receives Request(k) messages :

```
if holder_as_I_see_it $\neq$ "self"          {If this site does not have the token,
then begin                                 it forwards the request to the site who
      Send Request(k) to holder_as_I_see_it;   it thinks has the token and then
      holder_as_I_see_it := k;             updates its variable's name}
end else if ((using_CS $\wedge$ (who_to_pass_token = "none")) $\vee$
            (requesting_CS $\wedge$ (who_to_pass_token = "none")))
      then begin                           {If a request comes when this site is
            who_to_pass_token := k;        waiting to execute or executing the
            holder_as_I_see_it := k;       critical section, then save this name
      end else begin                       for later use}
            Send token to k;               {If this node has finished executing its
            have_token := false;           critical section, pass the token to the
            holder_as_I_see_it := k;       requesting node}
            end;
```

Process Which Receives Token message :

```
have_token := true;
```

Figure 5.1 - Formal Description of Protocol 3

critical section in the process which invokes mutual exclusion are interruptible, that is the other two processes can be executed that time. It is also assumed that multiple requests within a node to access the critical section are serialized.

A site not holding the token and wishing to enter the critical section sends a request message to the site given by holder_as_I_see_it. A non-privileged site, on receiving a request, forwards the request to the site who it thinks holds the token, and updates its variable holder_as_I_see_it to contain the requesting site's identifier for directing the next request to that site. The privileged node passes the token to the requesting node when it no longer needs the token for itself, that is when it has finished executing its critical section.

There is only one implementation detail which is not covered in the informal discussion of the protocol. A privileged or going-to-be privileged site must remember who to pass the token to separately since it is possible for this site to view the ultimate holder of the token different from the site to whom the token is passed by it. This happens when there is more than one request directed at a site while it is executing its critical section or waiting to execute the critical section as the token has not reached it yet.

The proposed algorithm uses a dynamic information structure. Each site at any given time keeps dynamic information about two nodes only - the current holder of the token as viewed by it (represented as holder_as_I_see_it in Figure 5.1), and the node which is passed the token next by it (represented as who_to_pass_token in Figure 5.1).

Token Size - The token used in this algorithm does not contain any information other than it is a “special” message. This is an advantage over most of the other token-based algorithms where size of the token message is considerably big. For example, in the Ricart-Agrawala algorithm, the token contains an array of size N to store sequence numbers of the sites [Ricart 1983]; in the Suzuki-Kasami algorithm, the token contains an array of size N to store sequence numbers of the sites and a queue, whose size varies from 0 to N-1, of requesting nodes [Suzuki 1985]; in Singhal's algorithm, the token contains an array of size N to store sequence numbers of the sites and a vector of size N to store the state information of all the sites in the system [Singhal 1989].

Message Overtaking - In the proposed algorithm, the order of message delivery does not have to be preserved. Consider the situation when site i sends the token to site j and then issues a request to j to access the critical section again. There is no problem even if i's request is serviced by j before the token reaches j. On receiving i's request, j will set who_to_pass_token to i and will transfer the token back to i only after using its critical section. In the meantime, any other request to j will be forwarded to i.

5.2.1.3 Correctness Proofs :

The proofs for mutual exclusion and freedom from both deadlock and starvation are given below -

• **Mutual Exclusion is Achieved** -

In token-based algorithms, a site cannot enter the critical section if it does not possess the token. So mutual exclusion may be violated only if a site passes the token to another site while it is executing the critical section. In the algorithm, the token can be passed to another site either in the exit code of the process which invokes mutual exclusion or in the process which receives the request messages. A site executes its exit code only after it has finished executing its critical section. In the process which receives request messages, the token is passed only if holder_as_I_see_it = "self" and both requesting_CS and using_CS are false. Therefore, the algorithm guarantees mutual exclusion. □

For the proofs of following lemmas, the variable holder_as_I_see_it is represented at each node by suffixing the node's name to it. The nodes and the values of the variable holder_as_I_see_it at each node can be represented as a directed graph $G = (V,E)$, where $V =$ set of nodes, and $E = \{(i,j) \mid \text{holder\_as\_I\_see\_it}_i = j \land i, j \in V\}$. In this notation, the value "self" for a node is denoted as its own unique name. The loops formed from the values of the variable $\text{holder\_as\_I\_see\_it}_i = i$ do not have any effect on the algorithm because of the following reasons - (i)a node possessing the token does not send a request to itself, (ii)a node, on receiving a request, does not send that request to itself again, and (iii)a node never transmits the token to itself. Therefore, these loops are not considered in the proofs given below.

Lemma 1 - A node has at most one outgoing edge.

Proof - Since the variable holder_as_I_see_it at a node can hold only one value at a time, and an edge from this node is formed using the value of its holder_as_I_see_it variable, there can be only one outbound edge. Also, since it is possible for a site i to have $\text{holder\_as\_I\_see\_it}_i = i$, there is no edge from this node then. □

Lemma 2 - It is impossible to have a cycle in the directed graph G.

Proof - (1) Since the token always stays with the site that used it last, when there are no pending requests in the system, the variable holder_as_I_see_it for that site contains the value "self" . Therefore, one node in the system has no outgoing edge then.

(2) If there are requests floating in the network (that is, they have not yet reached their destinations), then it is possible to have more than one site with holder_as_I_see_it = "self". The graph is then disconnected. However, the algorithm in Figure 5.1 ensures that the requests are directed/going to be directed to all but one of these sites and that would change the variable holder_as_I_see_it at these sites in such a way that the graph is again connected with only one node having no outgoing edge.

(3) Assume that a cycle is formed. This implies that each site involved in the cycle has an outgoing edge. The other sites, not involved in the cycle cannot remain permanently isolated as the final graph is connected. Further, these rest of the sites can only point, directly or indirectly, to one of the sites involved in the cycle as a site can have only one outgoing edge (from Lemma 1). Thus, all the sites in the graph have an outgoing edge. But from (1) and (2), there

is one node which has no outgoing edge. This is a contradiction and therefore, the assumption made is wrong.

Hence, a cycle is never formed. □

Lemma 3 - A request to access the critical section will always reach a node which possesses or is going to possess the token and has who_to_pass_token = "none".

Proof - (1) The communication network is assumed to be reliable. So a request is never lost.

(2) From Lemma 2, a cycle is never formed in the graph of nodes. Therefore, a request does not keep circulating among nodes.

(3) Since a site cannot generate another request until one request is satisfied and the site with the token does not forward the first request it handles to another site, a request never reaches back to the node which generated it.

(4) Transmissions delays are assumed to be finite.

From (1), (2), (3), and (4), it follows that a request to access the critical section reaches a node, which has/is going to get the token and has the value of the variable who_to_pass_token equal to "none", in a finite amount of time. □

- **Deadlock is Impossible** -

Deadlock occurs when no node is in the critical section and there is at least one node trying to enter it and cannot do so.

Proof - Let R_i be the request from site i to use the critical section. If site i holds the token, then there is nothing that can prevent i from entering its critical section. (Of course, if site i generated R_i and it has the token, then it is not executing the critical section as multiple requests are serialized. Also, there are no other requests when R_i is generated and serviced.)

If site i does not hold the token, then it is guaranteed that R_i will reach a node, say m, which has/is going to get the token and it is the first request m is going to service (from Lemma 3). Now, according to the algorithm in Figure 5.1, if that node m is not using the critical section, it must immediately send the token to i; otherwise it sets its who_to_pass_token to i and upon finishing execution of the critical section, it will send the token to i. That is, it is impossible that m keeps the token forever when it has serviced a request from another node. Since the token takes only a finite amount of time to reach i and the possession of the token is equivalent to accessing the critical section, i then enters its critical section.

Hence, deadlock is impossible. □

- **Starvation is Impossible** -

Starvation occurs when one node waits indefinitely to enter its critical section while other nodes are entering and leaving their critical section. So we wish to prove that every request to enter the critical section is satisfied within a finite time.

Proof -

(1) From Lemma 3, we know that a request R reaches the node which possesses or is going to possess the token and has not serviced a request.

(2) We also know that a node, on seeing R, will not forward any future requests directed at it to the node to which it forwards R. Rather, it would now forward the next request to the node which generated R. Therefore, no new requests from that node (its own or on behalf of other nodes) can precede R after it has serviced R.

(3) Transmission delays are assumed to be finite.

From (1), (2), and (3), it follows that any request R is eventually satisfied in finite time. Hence, starvation is impossible.□

5.2.1.4 Cost of the Algorithm :

The cost of the algorithm is measured in terms of the number of messages required for one execution of the critical section.

Like other token-based algorithms, if a node has the token and there are no pending requests to be serviced, that node can enter the critical section without communicating with anybody and therefore, the number of message exchanges is 0.

When a node does not have the token, the best case would be when the request message is directed to the node which has the token and that node services this request first. In that case, only 2 messages are needed for mutual exclusion invocation, the request and the passing of the token.

The worst case occurs when the nodes arrange themselves in a straight line. This can happen because of the way requests to enter the critical section are satisfied. This is illustrated below. In this case, if the node at one end holds the token and if the node at the other end wishes to enter the critical section, it needs (N – 1) request messages and 1 token message to do so. Therefore, a total of N messages is required in the worst case.

Example to demonstrate the worst case -

Assume 6 people (1 through 6) share a book. Also, assume the book is with 3, and the state of the system is as given -

holder_as_I_see_it$_i$ = 3, for i = 1 and 2,
holder_as_I_see_it$_i$ = "self", for i = 3,
holder_as_I_see_it$_i$ = 1, for i = 4 , 5, and 6, and
who_to_pass_token = "none", for all i = 1 to 6.

Consider the case, when 2 needs the book and the state of the system is as given above. So 2 sends a request to 3. Now assume that 4, 5, 6, and 1 use the book in this order before 2's request reaches 3. The state of the system then is -

holder_as_I_see_it$_1$ = "self", holder_as_I_see_it$_2$ = "self",
holder_as_I_see_it$_3$ = 4, holder_as_I_see_it$_4$ = 5,
holder_as_I_see_it$_5$ = 6, holder_as_I_see_it$_6$ = 1,
who_to_pass_token$_i$ = "none", for all i = 1 to 6.
There is a request in the network from 2 to 3.

On receiving 2's request, 3 forwards it to 4, 4 forwards it to 5, 5 forwards it to 6, and 6 forwards it to 1. Also, 3, 4, 5, and 6 cannot use the book before 2 , once they have seen 2's request. So it takes 5 request messages (2→3, 3→4, 4→5, 5→6, 6→1) and one for transferring the book (1→2) for 2's request to be satisfied.

The example becomes more interesting when 2's request reaches 3 when 3 had just passed the book to 4, then 3's request on behalf of 2 reaches 4 when 4 had just passed the book to 5, and so on. But this does not affect the worst case analysis of the algorithm.

Since the proposed algorithm uses a dynamic information structure, the number of messages vary between 0 and N.

5.2.1.5 Failure Considerations :

This section addresses the effects of both link and site failures on the proposed algorithm and presents methods for recovery from these failures. The Byzantine failures [Lamport 1982] are not considered.

There are many states in which a system can be when it fails [Singhal 1989]. A crash recovery procedure should be able to pull the system back from all of these states. Such an exhaustive crash recovery procedure is not given here, but some of the more important cases are discussed.

- **Message Losses** -

It is assumed that a message is either delivered correctly or not delivered at all by the network communication subsystem. This can be ensured by using error detecting codes [Tanenbaum 1989]. Message loss can be detected using time-out mechanisms.

If a request message is lost, the sending site will have to make the request again. So loss of a request message is not a big problem. However, if the token is lost, it needs to be handled carefully as only one site must regenerate the token. An election algorithm [Garcia-Molina 1982; Peterson 1982; Hirschberg 1980] may be used to generate the new token.

- **Link Failures** -

It is assumed that the underlying network layer informs the sender if a message cannot be sent because of a link failure. There are two situations to be taken into account -

A) When the link failures cause network partitioning -

In this case, the network graph is divided into two different subgraphs. Following two situations may occur then -

(i)holder_as_I_see_it$_i$=j, where i and j belong to separate subgraphs.

(ii)who_to_pass_token$_i$=j, where i and j belong to separate subgraphs.

In the first situation, a request cannot be made by i or forwarded by i on behalf of the other sites which have holder_as_I_see_it variable value equal to j (transitive closure). Other sites can still enter the critical section and mutual exclusion condition is still maintained. In the second situation, the token cannot be passed. The mutual exclusion constraint is not violated as the token remains with one site only.

It is possible to have the token in the subgraph where all the sites have their holder_as_I_see_it variables set to sites from the other subgraph. In that case, critical section is inaccessible to all sites once the site with the token has found out that the token cannot be passed to the site in the other subgraph. This does not affect the mutual exclusion constraint; it only causes delay in execution of the critical section by a site. The sýstem jumps back to full activity once the connectivity is restored.

If the amount of parallelism is a big consideration and the estimated time to restore the system connectivity is large, the Recovery Procedure 2 given below may be used. The Recovery Procedure 1 must be followed anyway once the system is restored as the sites in the subgraph without the token will have no idea where the token is. These sites will have to issue a new request to access the critical section as the requests at the time of partition are not satisfied.

Recovery Procedure 1 -

When system connectivity is restored, the sites in the partition (say A) which did not have the token need to be notified which site holds the token in the other partition (say B). This can be achieved by having all the sites in partition A send a "recovery" message to all the sites in partition B. Sites in partition A cannot request for the critical section until an "inform" message reaches them. On receiving the "recovery" message, the site holding the token or going to have the token and having who_to_pass_token = "none" sends an "inform" message containing its identity to the site which sent that "recovery" message. Other sites ignore the "recovery" message. On receiving the "inform" message, a site sets its holder_as_I_see_it to contain the identity of the site which sent the "inform" message. The variable who_to_pass_token is set to "none" for the sites in partition A. Thus the connectivity of the graph formed by the sites and the values of the holder_as_I_see_it variable are restored.

Recovery Procedure 2 -

Once it is found by a site (say i) that token can't be passed to the site given by $\text{who\_to\_pass\_token}_i$, it sends an "attention needed" message containing its identity to all the sites. The sites which are still connected to site i in the physical network graph will receive this message and then change their holder_as_I_see_it variable value to i to reflect that change in the graph formed (by sites and the values of the variable holder_as_I_see_it) in the algorithm. This would enable the sites in the partition to access the critical section. Once the connectivity is restored, Recovery Procedure 1 is followed.

However, there is a problem - what if a request message was already sent to i by another site? This can be handled by using timestamps in the messages. i would neglect all request messages marked with a timestamp value smaller than the "attention needed" message. A site on receiving the "attention needed" message would generate a request message again.

B) **When the link failures do not cause network partitioning -**

Even if link failures do not cause the network graph to be disconnected, it is possible that the graph formed by the sites and the values of the variable holder_as_I_see_it gets partitioned. This does not pose problems as the strong connectivity feature can be exploited (network is assumed to be fully connected) to find alternate paths and complete message transmissions.

- **Site Failures** -

It is assumed that site failures can be detected by some kind of mechanism, such as time-outs. Once a site failure is detected, it is made known through messages to the other sites by the site detecting this failure. It is also assumed that a site does not malfunction on failing.

The following situations that require action may occur at the time a site (say j) fails -

(1) j had the token at the time of failure, or

(2) j is going to get the token as it had made a request to enter the critical section before it failed, or

(3) there is a request message directed at the failed site j, or

(4) j was recovering from a previous failure.

Each of these cases is handled separately. The recovery procedure for the failed site j is given first. This is common to all the four cases.

Recovery Procedure 3 (recovery from a failed state) -

In the recovery phase, the failed site (j) sends a "recovery" message to all the sites. On receiving this "recovery" message, the site holding the token and having who_to_pass_token = "none" sends an "inform" message containing its name to the recovering site j. After the receipt of this "inform" message, site j sets its variable holder_as_I_see_it to contain the name of the site which sent the "inform" message. Site j assigns "none" to its variable who_to_pass_token.

It is possible that more than one "inform" message is received by a recovering node (due to unpredictable communication delay). But it is sufficient to have the recovering node process only the first "inform" message and ignore the rest, because processing of one such message connects the recovering site back into the dynamic graph formed in the algorithm.

The recovery procedures for each of the four cases are discussed below -

Case One - The token is definitely lost. So it needs to be regenerated. As mentioned earlier, detection of token-loss is not a trivial problem since the token may be considered to be lost when instead network connectivity is broken. Recovery Procedure 4 tries to bring back system activity in such a situation.

Recovery Procedure 4 -

(1) Run a token-recovery algorithm. An election algorithm may be employed to regenerate the token.

(2) Once the token is regenerated, all other sites have to be notified of the site which has the token (this is necessary if the token-recovery algorithm does not do so). The site which regenerates the token assigns "self" to its holder_as_I_see_it variable and all other sites set their holder_as_I_see_it variable to contain the name of the token regenerating site. The variable who_to_pass_token is set to "none" at all the sites.

(3) A site (except the failed site) can make a request to access the critical section only after it has performed the first two steps completely.

(4) When the failed site recovers, it follows the Recovery Procedure 3 given earlier.

Case Two - If the failure of site j is detected before the token is passed to it, then the token is not transmitted to j. Then a recovery procedure containing steps 2, 3, and 4 of Recovery Procedure 3 is employed. This may cause some sites to make a request again to enter the critical section. However, if the token is already on its way to j, it will be lost when it reaches the failed site j. It is then handled in a similar manner to the first case.

Case Three - When a request message is sent to the failed site, it is considered lost. Also, the sites, whose holder_as_I_see_it variables are equal to the failed site's identifier, cannot make a request. This problem appears in the second case also. It may be handled by waiting till the failed site recovers (at the cost of wasting parallelism). But these (dependent) sites can be allowed to reorganize themselves in the dynamic graph formed by the algorithm. The details are given in Recovery Procedure 5.

Recovery Procedure 5 -

There are two parts of this recovery procedure -

(1) Reorganization of the sites dependent on the failed site, and

(2) Recovery of the failed site (same as Recovery Procedure 3).

The sites whose holder_as_I_see_it variable contains the name of the failed site send a "help me" message to all other sites. On receiving this "help me" message, the site holding the token and having who_to_pass_token = "none" sends an "inform" message containing its name to the site which sent the "help me" message. Other sites ignore the "help me" message. On receiving this "inform" message, a site sets its variable holder_as_I_see_it to contain the name of the site which sent the "inform" message. Thus, these sites are connected back in the dynamic graph of the algorithm and they can now make a request to enter the critical section.

Case Four - When a site fails during its recovery from previous failure, it starts the recovery procedure again from the beginning. There is however one problem - an old “inform” message may reach this site during its current recovery procedure. This can be handled by using timestamps.

The recovery procedures given above demonstrate that the proposed algorithm allows dynamic reconfiguration of the network. A new node can be added to the system by following the Recovery Procedure 3. In fact, the dynamic nature of the algorithm makes it easier to handle these cases.

5.2.2 Mutual Exclusion in case of M instances of the Resource :

In practical systems, it happens quite often that there exists more than one resource of the same kind and each resource can be used by at most one process at any given time. A process does not care which resource it uses as long as it gets to use a resource. Example 2, given at the beginning of this chapter, fits this description very well.

The solution of this problem is built upon the mutual exclusion algorithm for one shared resource. Extensions are proposed to the Ricart-Agrawala algorithm, the Suzuki-Kasami algorithm, and the algorithm from the previous section to solve this problem.

In the following discussions, N is the number of sites and M is the number of resources available.

5.2.2.1 Extension to the Ricart-Agrawala Algorithm -

The following changes to the Ricart-Agrawala algorithm are proposed to solve this problem -

(1) Instead of waiting for $(N-1)$ "replies" in the entry code for access to the critical section, wait for $(N-M)$ "replies" only.

(2) Since nothing is assumed about the time taken for a message to reach its destination node, it is now possible (because of step 1 above) that a site receives a "reply" (from a site) to an old request while this site has made another request to access the critical section. This can be handled by having a site timestamp its "replies" like it timestamps its "requests". Then, on receipt of a reply, it can be decided, whether or not that "reply" pertains to the current request, by comparing the timestamp of the "request" made with that of the "reply" received.

5.2.2.2 Extension to the Suzuki-Kasami Algorithm -

In the Suzuki-Kasami algorithm, the token determines which site enters its critical section. Since there is only one token in the system, only one site can access the shared resource at a time. Two extensions are proposed to the Śuzuki-Kasami algorithm to achieve mutual exclusion in case of M copies of the shared resource. The basic idea in both of these extensions can be applied to any token-based algorithm and is given below -

(1) the token carries the number of available shared resources, which is represented here by M. In the Suzuki-Kasami

algorithm, M can be considered to carry the value 1, or

(2) there are M tokens in the system.

In the first extension, a site decrements M on getting the token and may retransmit it to another requesting site if $M > 0$. After a site finishes execution of its critical section, it broadcasts a "release" message, if it does not have the token; otherwise, it increments the value of M on the token. Special care has to be taken so as not to update M at different sites for the same "release" message, and also in the situation when the token is in transit at the time of broadcast of the "release" message. This can be handled by having each site maintain sequence number of the "release" message received from all the sites, like it does for the "request" message. In the Suzuki-Kasami algorithm, the token carries the sequence number of the last request satisfied for each site. So by comparing the sequence number of the last request satisfied and the sequence number of the "release" message, it can be found whether or not M has to be updated.

In the second extension, there are M tokens in the system, one for each instance of the resource. Assume these M tokens to be distributed among the nodes. Since in the Suzuki-Kasami algorithm, a request message is sent to all the nodes, it is possible that a site receives more than 1 token in response to its one request. This site must immediately pass the extra tokens to the other requesting sites so as not to waste parallelism. It is also possible for a site to receive a token when it is finished executing its critical section and has not made a request to enter the critical section again. Since a site sends a request message to all the sites, this does not pose any problems except for some extra message transmissions and loss of parallelism. This M tokens approach is more useful than the former scheme (where there is one token and it carries the value M) if the system involves processes like in the *readers-writers* problem. In that case, a writer (site) waits until it has obtained all tokens, where as a reader (site) can read with just one token.

5.2.2.3 Extension of the Proposed Algorithm -

The algorithm proposed in §5.2.1.2 is also token-based. Therefore, the same two extensions are possible - one involving only one token which carries the value M and the other involving M tokens.

In the first extension, assume that initially the token with value M is at site 1. In the algorithm in Figure 5.1, the token is not transmitted by a site if it is in its critical section. Since more than one site can be in its critical section, the token is allowed to be transmitted by a site even if it is executing the critical section, but with the condition that $M > 0$.

On receiving the token, a site decrements the value of M by 1. The token stays at that site if $M = 0$, otherwise it is transmitted to a requesting site. When a site finishes execution of its critical section, M needs to be incremented by 1. If the site has the token, the task of incrementing is no problem. If the site does not possess the token, it needs to be handled with care. Two methods are given here for handling this -

(1) Broadcast a "release" message to all the sites. The site with the token increments M by 1 after receiving a "release" message. Broadcast of "release" message introduces the need for each site to maintain sequence numbers like in the Suzuki-Kasami algorithm.

(2) Transmit the "release" message to the site given by its holder_as_I_see_it variable. On receiving a "release" message, the site holding the token increments M by 1; a site not possessing the token forwards it to its holder_as_I_see_it site. This method reduces the number of messages but there is a risk involved - the "release" message may end up chasing the token. This is not very likely to occur unless the critical section is very short or there is a heavy demand to use the shared resources. Parallelism could be lost (by saving on the number of messages) as the "release" message may visit many nodes before reaching the node with the token.

In the second extension, there are M tokens in the system. Assume the tokens are initially distributed among all the sites. Also assume that each site has a set which contains the names of the sites which it thinks have the token. Initially, each such set is initialized to contain the names of the token holding sites. A reader site picks a site from its set of sites (can be random) as the one which it thinks has the token and then follows the protocol as given in Figure 5.1. A writer site follows a "greedy" approach by sending a request message to all the sites in its set. A site may update its set according to the following rules -

(i) When a site, say i, sends a request for itself to another site (one from its set), say j, it removes j from the set and adds its own name i to the set.

(ii) When a site, say k, sends a request on behalf of i to another site (one from its set), say j, it removes j from the set and adds i to the set.

(iii) The site, which transmits the token, removes its own name from its set and adds the name of the site to which it sends the token.

This extension has a serious problem - what if two writers try to capture (all M) tokens at the same time? This can be solved by using timestamps with the request messages. Then the writer with the lower timestamp has precedence over the other writer and thus, deadlock is avoided.

5.3 Summary :

In this chapter, a new distributed mutual exclusion algorithm which requires between 0 and N message exchanges is proposed. Ideas are presented for extending one resource mutual exclusion algorithms to solve the mutual exclusion problem in the case where there is more than one copy of the resource.

CHAPTER VI

CONCLUSIONS

The goal of this work was to consider various methods of implementing mutual exclusion in both centralized and distributed systems.

Most of the currently available computer systems provide at least one of the mechanisms of Chapter II at the hardware level. So the mutual exclusion problem, local to a computer system, can be solved efficiently using the hardware mechanism available on that system. Algorithms in Chapter III implement the required synchronization within a system using the standard operators of a high-level programming language. These solutions are important not only from a historical point of view but also because they illustrate how concurrent programs behave.

Since the solutions of Chapters II and III are dependent on the existence of a shared memory, they cannot be used in distributed systems. Chapter IV discusses mutual exclusion solutions based on message-passing. The main characteristic of these solutions is the multiplicity of decision-making centers. And the major source of problems is the unpredictability of transmission delays along communication channels.

It is shown in Chapter V that new solutions to the problem of mutual exclusion can be formed by using heuristics. These heuristics are developed by considering real-world situations which require mutual exclusion. An algorithm, which is shown to be more economical than most of the other existing algorithms, is proposed. It requires between 0 and N message exchanges per critical section execution. The token size in this algorithm is smaller in comparison to that in other token-based algorithms. The effects of both site and link failures on the algorithm are considered in detail and procedures for recovery from these failures are also given.

In the proposed algorithm, requests are ordered based on their time of arrival at a site. Since all requests cannot be serviced at the same time, there does exist an order among the requests and this is made use of. Raymond uses a similar ordering scheme in his mutual exclusion algorithm [Raymond 1989]. The algorithm in Chapter V does not grant access to the critical section in a first-come-first-served order like [Lamport 1978] and [Ricart 1981]. But those two algorithms do so at the cost of more message exchanges.

The algorithm assumes that each site is equally likely to access the critical section and each access to the critical section is equally important. These assumptions may be relaxed a little bit. An example of why this would be desirable is given in terms of Example 1 from the beginning of Chapter V - "What if one person has an exam and the others don't? So that person needs the book more than anybody else".

This situation can be handled by introducing an "urgent" message and making all sites respect this "urgent" message. Of course, it is assumed that there are no false "urgent" messages. This gives rise to new problems. What if two sites issue an “urgent” message? Timestamps may be used by a site to determine the “more urgent” message of the two.

It was also assumed in the algorithm that a site serializes its multiple requests. This may be relaxed when a site possesses the token. If an internal request is generated to access the critical section and there is a request from another site pending to be serviced, the internal request may be satisfied first to save on the number of message exchanges. Theoretically speaking, this can cause starvation. But in practice, it is unlikely. (Based on this assumption, Lamport gave a “fast” mutual exclusion algorithm for shared memory systems [Lamport 1987].)

The distributed mutual exclusion solutions for one shared resource are extended in Chapter V to solve the problem of mutual exclusion in the case where there is more than one instance of the shared resource. It is assumed in these extensions that availability of any shared resource (from that pool of shared resources) satisfies a request. These extensions can be modified to include specific resource demands, if any.

The solutions to the problem of mutual exclusion in this work assume presence of only one critical section. The problem when processes have more than one critical section, which overlap with each other, needs to be considered in future.

Due to time constraints, we could not do a performance evaluation of the proposed algorithm. Gravey and Dupis proposed a modeling method for performance evaluation of distributed mutual exclusion protocols. They analyzed the performance of two such algorithms implemented in a distributed system consisting of two nodes only as the cardinality of the state space of the Markov Chain used grows rapidly with the number of nodes in the system [Gravey 1987]. A complete analytic study of the proposed algorithm is a topic of research in itself.

To sum up, the problem of mutual exclusion is something which cannot be overlooked by a system designer. A variety of solutions to this problem are available. Each solution has its own advantages and disadvantages - one has to choose a suitable solution for the problem depending on what factors (availability of shared memory, centralized or distributed control, cost, network topology, reliability, etc.) need to be emphasized.

BIBLIOGRAPHY

[Agrawal 1991] D. Agrawal, and A. E. Abbadi: *An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion*, ACM TOCS, 9(1), Feb. 1991, 1-20.

[Andre 1985] F. Andre, D. Herman, and J. -P. Verjus: *Synchronization of Parallel Programs*, North Oxford Academic, 1985.

[Andrews 1982] G. R. Andrews, D. P. Dobkin, and P. J. Downey: *Distributed Allocation with Pools of Servers*, in ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, Aug. 18-20, 1982, 73-83.

[Andrews 1983] G. R. Andrews, and F. B. Schneider: *Concepts and Notations for Concurrent Programming*, ACM Computing Surveys, 15(1), March 1983, 3-43.

[Andrews 1991a] G. R. Andrews: *Paradigms for Process Interaction in Distributed Programs*, ACM Computing Surveys, 23(1), March 1991, 49-90.

[Andrews 1991b] G. R. Andrews: *Concurrent Programming - Principles and Practice*, The Benjamin/Cummings Publishing Co., Inc., California, 1991.

[Axford 1989] Tom Axford: *Concurrent Programming - Fundamental Techniques for Real-Time and Parallel Software Design*, John Wiley & Sons, 1989.

[Bagrodia 1989] R. Bagrodia: *Process Synchronization - Design and Performance Evaluation of Distributed Algorithms*, IEEE Transactions on Software Engineering, 15(9), Sept. 1989, 1053-1065.

[Beauquier 1990] J. Beauquier: *Fault-Tolerant Naming and Mutual Exclusion*, in Lecture Notes in Computer Science, Vol. 469, Springer-Verlag, Berlin, 1990, 50-61.

[Belpaire 1975] G. Belpaire: *Synchronization - Is a synthesis of the problems possible?*, in Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, 1975, 3-10.

[Ben-Ari 1990] M. Ben-Ari: *Principles of Concurrent and Distributed Programming*, Prentice Hall International, 1990.

[Bernstein 1981] P. A. Bernstein, and N. Goodman: *Concurrency Control in Distributed Database Systems*, Computing Surveys, 13(2), June 1981, 185-221.

[Birman 1987] K. Birman, and T. Joseph: *Reliable Communication in Presence of Failures*, ACM TOCS, 5(1), Feb. 1987, 47-76.

[Brown 1989] G. M. Brown, M. G. Gouda, and C. -L. Wu, *Token Systems that Self-Stabilize*, IEEE Transactions on Computers, C-38(6), June 1989, 845-852.

[Burns 1982] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson: *Data Requirements for Implementation of N-process Mutual Exclusion Using a Single Shared Variable*, Journal of the ACM, 29(1), Jan. 1982, 183-205.

[Burns 1987] J. E. Burns, and G. L. Peterson: *Constructing Multi-reader Atomic Values from Non-atomic Values*, in Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Aug. 1987, 222-231.

[Campbell 1974] R. H. Campbell, and A. N. Habermann: *The Specification of Process Synchronization by Path Expressions*, in Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, Berlin, 1974, 89-102.

[Carvalho 1982] O. S. F. Carvalho, and G. Roucairol: *On the Distribution of an Assertion*, in ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, Aug. 18-20, 1982, 121-131.

[Carvalho 1983] O. S. F. Carvalho, and G. Roucairol: *On Mutual Exclusion in Computer Networks*, CACM, 26(2), Feb. 1983, 146-147.

[Chandy 1984] K. M. Chandy, and J. Misra: *The Drinking Philosophers Problem*, ACM TOPLAS, 6(4), Oct. 1984, 632-646.

[Chandy 1985] K. M. Chandy, and L. Lamport: *Distributed Snapshots - Determining Global States of Distributed Systems*, ACM TOCS, 3(1), Feb. 1985, 63-75.

[Chandy 1988] K. Mani Chandy, and J. Misra: *Parallel Program Design - A Foundation*, Addison-Wesley Publishing Co., 1988.

[Chen 1975] R. C. Chen: *Representation of Process Synchronization*, in Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, 1975, 37-42.

[Chern 1989] S. E. Chern: *The Generalized Mutual Exclusion Problem in a Computer System*, Ph.D. Dissertation, The University of Texas at Dallas, Aug. 1989.

[Cornafion 1985] -: *Distributed Computing Systems - Communication, Cooperation, Consistency*, Elsevier Science Publishers B.V., Amsterdam, 1985.

[Courtois 1971] P. J. Courtois, F. Heymans, and D. L. Parnas: *Concurrent Control with Readers and Writers*, CACM, 14(10), Oct. 1971, 667-668.

[deBruijn 1967] N. G. deBruijn: *Additional Comments on a Problem in Concurrent Programming Control*, CACM, 10(3), March 1967, 137-138.

[Deitel 1990] H. M. Deitel: *An Introduction to Operating Systems*, Addison-Wesley Publishing Co., 1990.

[Dijkstra 1965] E. W. Dijkstra: *Solution of a Problem in Concurrent Programming Control*, CACM, 8(9), Sept. 1965, 569.

[Dijkstra 1968] E. W. Dijkstra: *The Structure of "THE" Multiprogramming System*, CACM, 11(5), May 1968, 341-346.

[Dijkstra 1968a] E. W. Dijkstra: *Cooperating Sequential Processes*, in Programming Languages, F. Genuys (ed.), Academic Press, London, 1968, 43-112.

[Dijkstra 1971] E. W. Dijkstra: *Hierarchical Ordering of Sequential Processes*, Acta Informatica, 1(2), 1971, 115-138.

[Dijkstra 1972] E. W. Dijkstra: *Information Streams Sharing Finite Buffer*, Information Processing Letters, 1, 1972, 179-180.

[Dijkstra 1974] E. W. Dijkstra: *Self-stabilizing Systems in spite of Distributed Control*, CACM, 17(11), Nov. 1974, 643-644.

[Dijkstra 1980] E. W. Dijkstra, and C. S. Scholten: *Termination Detection for Diffusing Computations*, Information Processing Letters, 11(1), Aug. 1980, 1-4.

[Dijkstra 1983] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, *Derivation of a Termination Detection Algorithm for Distributed Computations*, Information Processing Letters, 16(5), June 1983, 217-219.

[Dijkstra 1986] E. W. Dijkstra: *A Belated Proof of Self-stabilization*, Distributed Computing, 1, 1986, 5-6.

[Doran 1980] R. W. Doran, and L. K. Thomas: *Variants of the Software Solution to Mutual Exclusion*, Information Processing Letters, 10(4,5), July 1980, 206-208.

[Dupis 1986] A. Dupis, and G. Hebuterne: *On the Use of Quantitative Evaluation to Assess and Study Distributed Algorithms Properties*, in Proceedings of the IFIP WG 6.1 Fifth International Workshop on Protocol Specification, Testing, and Verification, France, June 10-13, 1985. Also in, Protocol Specification, Testing, and Verification V, M. Diaz (ed.), North-Holland, Amsterdam, 1986, 363-373.

[Eisenberg 1972] M. A. Eisenberg, and M. R. McGuire: *Further Comments on Dijkstra's Concurrent Programming Control Problem*, CACM, 15(11), Nov. 1972, 999.

[Elshoff 1988] I. J. P. Elshoff, and G. R. Andrews: *The Development of Two Distributed Algorithms for Network Topology*, Technical Report No. 88-13, Department of Computer Science, University of Arizona, Tucson, Arizona, 1988.

[Enslow 1978] P. H. Enslow: *"What is a distributed data-processing system?"*, IEEE Computer, Jan. 1978, 13-21.

[Eswaran 1976] K. P. Eswaran, J.-N. Gray, R. A. Lorie, and I. L. Traiger: *The Notions of Consistency and Predicate Locks in a Database System*, CACM, 19(11), Nov. 1976, 624-633.

[Faulk 1988] Stuart R. Faulk, and David L. Parnas: *On Synchronization in Hard Real-Time Systems*, CACM, 31(3), March 1988, 274-287.

[Francez 1986] Nissim Francez: *Fairness*, Springer-Verlag, 1986.

[Freisleben 1989] B. Freisleben, and J. L. Keedy: *Priority Semaphores*, The Computer Journal, 32(1), 1989, 24-28.

[Garcia-Molina 1982] H. Garcia-Molina: *Elections in a Distributed Computing System*, IEEE Transactions on Computers, C-31(1), Jan. 1982, 48-59.

[Garcia-Molina 1985] H. Garcia-Molina, and D. Barbara: *How to Assign Votes in a Distributed System*, Journal of the ACM, 32(4), Oct. 1985, 841-860.

[Gifford 1979] D. K. Gifford: *Weighted Voting for Replicated Data*, in Proceedings of the 7th Symposium on Operating System Principles, ACM, New York, 1979, 150-162.

[Gottlieb 1983] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir: *The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer*, IEEE Transactions on Computers, C-32(2), Feb. 1983, 175-189.

[Gottlieb 1987] A. Gottlieb: *An Overview of the NYU Ultracomputer Project*, in Experimental Parallel Computing Architectures, J. J. Dongarra (ed.), Elsevier Science Publishers B.V., North-Holland, 1987, 25-95.

[Gravey 1987] A. Gravey, and A. Dupis: *Performance Evaluation of Two Mutual Exclusion Distributed Protocols via Markovian Modeling*, in Proceedings of the IFIP WG 6.1 Sixth International Workshop Protocol Specification, Testing, and Verification, Montreal, Qubec, Canada, June 10-13, 1986. Also, in Protocol Specification, Testing, and Verification VI, edited by B. Sarikaya, and G. V. Bochmann, North-Holland, Amsterdam, 1987, 335-346.

[Habermann 1972] A. N. Habermann:- *Synchronization of Communicating Processes*, CACM, 15(3), March 1972, 171-176.

[Hansen 1972a] Per Brinch Hansen: *Structured Multiprogramming*, CACM, 15(7), July 1972, 574-578.

[Hansen 1972b] Per Brinch Hansen: *A Comparison of Two Synchronizing Concepts*, Acta Informatica, 1(3), 1972, 190-199.

[Hansen 1973a] Per Brinch Hansen: *Operating System Principles*, Prentice-Hall, 1973.

[Hansen 1973b] Per Brinch Hansen: *Concurrent Programming Concepts*, ACM Computing Surveys, 5, 1973, 223-245.

[Hansen 1977] Per Brinch Hansen: *The Architecture of Concurrent Programs*, Prentice Hall, 1977.

[Hansen 1978] Per Brinch Hansen: *Distributed Processes - A Concurrent Programming Concept*, CACM, 21(11), Nov. 1978, 934-941.

[Hehner 1981] E. C. R. Hehner, and R. K. Shyamasundar: *An Implementation of P and V*, Information Processing Letters, 12(4), Aug. 1981, 196-198.

[Helary 1988] J. M. Helary, N. Plouzeau, and M. Raynal: *A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network*, The Computer Journal, 31(4), 1988, 289-295.

[Hirschberg 1980] D. S. Hirschberg, and J. B. Sinclair: *Decentralized Extrema-Finding in Circular Configuration of Processors*, CACM, 23(11), Nov. 1980, 627-628.

[Hoare 1969] C. A. R. Hoare: *An Axiomatic Basis for Computer Programming*, CACM, 12(10), Oct. 1969, 576-580.

[Hoare 1974] C. A. R. Hoare: *Monitors - An Operating System Structuring Concept*, CACM, 17(10), Oct. 1974, 549-557.

[Hoare 1985] C. A. R. Hoare: *Communicating Sequential Processes*, Prentice Hall International, 1985.

[Howard 1976] J. H. Howard: *Proving Monitors*, CACM, 19(5), May 1976, 273-279.

[Hwang 1985] K. Hwang, and F. A. Briggs: *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1985.

[Hyman 1966] H. Hyman: *Comments on a Problem in Concurrent Programming Control*, CACM, 9(1), Jan. 1966, 45.

[Intel 386^{TM}] -, 386^{TM}SX Processor Programmer's Reference Manual.

[Intel 486^{TM}] -, i486^{TM} Microprocessor Programmer's Reference Manual.

[Intel 80286] -, Introduction to the iAPX 286^{TM}, Intel Corp.

[Joseph 1989] T. A. Joseph, and K. P. Birman: *Reliable Broadcast Protocols*, in Distributed Systems, S. Mullander (ed.), ACM Press Frontier Series, Addison-Wesley Publishing Co., 1989, 293-317.

[Keedy 1979] J. L. Keedy, K. Ramamohanarao, and J. Rosenberg: *On Implementing Semaphores with Sets*, The Computer Journal, 22(2), 1979, 146-150.

[Keedy 1982] J. L. Keedy, J. Rosenberg, and K. Ramamohanarao: *On Synchronizing Readers and Writers with Semaphores*, The Computer Journal, 25(1), 1982, 121-125.

[Keedy 1985] J. L. Keedy, and B. Freisleben: *On the Efficient Use of Semaphore Primitives*, Information Processing Letters, 21(4), 1985, 199-205.

[Kessels 1977] J. L. W. Kessels: *An Introduction to Event Queues for Synchronization in Monitors*, CACM, 20(7), July 1977, 500-503.

[Kessels 1979] J. L. W. Kessels, and A. J. Martin: *Two Implementations of the Conditional Critical Region Using a Split Binary Semaphore*, Information Processing Letters, 8(2), Feb. 1979, 67-71.

[Kessels 1982] J. L. W. Kessels: *Arbitration Without Common Modifiable Variables*, Acta Informatica, 17, 1982, 135-141.

[Kleinrock 1985] L. Kleinrock: *Distributed Systems*, CACM, 28(11), Nov. 1985, 1200-1213.

[Knuth 1966] D. E. Knuth: *Additional Comments on a Problem in Concurrent Programming Control*, CACM, 9(5), May 1966, 321-322.

[Kohler 1981] W. H. Kohler: *A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems*, ACM Computing Surveys, 13(2), June 1981, 149-183.

[Kosaraju 1973] S. R. Kosaraju: *Limitations of Dijkstra's Semaphore Primitives and Petri Nets*, Operating Systems Review, 7(4), Oct. 1973, 122-126.

[Kruijer 1979] H. S. M. Kruijer: *Self-Stabilization in spite of Distributed Control in Tree-Structured Systems*, Information Processing Letters, 8(2), 1979, 91-95.

[Kruskal 1988] C. P. Kruskal, L. Rudolph, and M. Snir: *Efficient Synchronization on Multiprocessors with Shared Memory*, ACM TOPLAS, 10(4), Oct. 1988, 579-601.

[Lakshman 1986] T. V. Lakshman, and A. K. Agrawala: *Efficient Decentralized Consensus Protocols*, IEEE Transactions on Software Engineering, SE-12(5), May 1986, 600-607.

[Lamport 1974] Leslie Lamport: *A New Solution of Dijkstra's Concurrent Programming Problem*, CACM, 17(8), Aug. 1974, 453-455.

[Lamport 1976] Leslie Lamport: *The Synchronization of Independent Processes*, Acta Informatica, 7, 1976, 15-34.

[Lamport 1977] Leslie Lamport: *Concurrent Reading and Writing*, CACM, 20(11), Nov. 1977, 806-811.

[Lamport 1977b] Leslie Lamport: *Proving the Correctness of Multiprocess Programs*, IEEE Transactions on Software Engineering, SE-3(2), March 1977, 125-143.

[Lamport 1978] Leslie Lamport: *Time, Clocks, and the Ordering of Events in a Distributed System*, CACM, 21(7), July 1978, 558-565.

[Lamport 1978b] Leslie Lamport: *The Implementation of Reliable Distributed Multiprocess Systems*, Computer Networks, 2, 1978, 95-114.

[Lamport 1979] Leslie Lamport: *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers, C-28(9), Sept. 1979, 690-691.

[Lamport 1980] Leslie Lamport: *The 'Hoare Logic' of Concurrent Programs*, Acta Informatica, 14(1), 1980, 21-37.

[Lamport 1982] L. Lamport, R. Shostak, and M. Pease: *The Byzantine Generals Problem*, ACM TOPLAS, 4(3), July 1982, 382-401.

[Lamport 1986a] Leslie Lamport: *The Mutual Exclusion Problem Part I - A Theory of Interprocess Communication*, Journal of the ACM, 33(2), April 1986, 313-326.

[Lamport 1986b] Leslie Lamport: *The Mutual Exclusion Problem Part II - Statement and Solutions*, Journal of the ACM, 33(2), April 1986, 327-348.

[Lamport 1987] Leslie Lamport: *A Fast Mutual Exclusion Algorithm*, ACM TOCS, 5(1), Feb. 1987, 1-11.

[Lamport 1990] Leslie Lamport: *Concurrent Reading and Writing of Clocks*, ACM TOCS, 8(4), Nov. 1990, 305-310.

[Lauesen 1975] Soran Lauesen: *A Large Semaphore Based Operating System*, CACM, 18(7), July 1975, 377-389.

[Lawrie 1975] Duncan Lawrie: *Access and Alignment of Data in an Array Processor*, IEEE Transactions on Computers, C-24, 1975, 1145-1155.

[Le Lann 1980] G. Le Lann: *Consistency, Synchronization, and Concurrency Control*, in Distributed Databases, edited by I. W. Draffan, and F. Poole, Cambridge University Press, 1980, 195-222.

[Liskov 1972] B. Liskov: *The Design of Venus Operating System*, CACM, 15(3), March 1972, 144-149.

[Liu 1986] Y. C. Liu, and G. A. Gibson: *Microcomputer Systems - The 8086/8088 Family, Architecture, Programming, and Design*, Prentice Hall, Second Edition, 1986.

[Lorin 1972] H. Lorin: *Parallelism in Hardware and Software - Real and Apparent Concurrency*, Prentice Hall Inc., 1972.

[Lynch 1987] N. A. Lynch, and M. R. Tuttle: *Hierarchical Correctness Proofs for Distributed Algorithms*, in Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, Aug. 10-12, 1987, 137-151.

[Maekawa 1985] M. Maekawa: *A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems*, ACM TOCS, 3(2), May 1985, 145-159.

[Maekawa 1987] M. Maekawa, A. E. Oldehoeft, R. R. Oldehoeft: *Operating Systems - Advanced Concepts*, The Benjamin/Cummings Publishing Co., Inc., 1987.

[Misra 1982a] J. Misra, and K. M. Chandy: *Termination Detection of Diffusing Computations in Communicating Sequential Processes*, ACM TOPLAS, 4(1), Jan. 1982, 37-43.

[Misra 1982b] J. Misra, K. M. Chandy, and T. Smith: *Proving Safety and Liveness of Communicating Processes with Examples*, in ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, Aug. 18-20, 1982, 201-208.

[Morgan 1985] C. Morgan: *Global and Logical Time in Distributed Algorithms*, Information Processing Letters, 20, May 1985, 189-194.

[Morris 1979] J. M. Morris: *A Starvation-Free Solution to the Mutual Exclusion Problem*, Information Processing Letters, 8(2), Feb. 1979, 76-80.

[Mullender 1989] S. Mullender(ed.): *Distributed Systems*, ACM Press Frontier Series, Addison-Wesley Publishing Co., 1989.

[Natarajan 1986] N. Natarajan: *A Distributed Synchronization Scheme for Communicating Processes*, The Computer Journal, 29(2), 1986, 109-117.

[Nishio 1990] S. Nishio, K. F. Li, and E. G. Manning: *A Resilient Mutual Exclusion Algorithm for Computer Networks*, IEEE transactions on Parallel and Distributed Systems, 1(3), July 1990, 344-355.

[Owicki 1976a] S. Owicki, and D. Gries: *An Axiomatic Proof technique for Parallel Programs*, Acta Informatica, 6(4), 1976, 319-340.

[Owicki 1976b] S. Owicki, and D. Gries: *Verifying Properties of Parallel Programs-An Axiomatic Approach*, CACM, 19(5), May 1976, 279-285.

[Owicki 1982] S. Owicki, and L. Lamport: *Proving Liveness Properties of Concurrent Programs*, ACM TOPLAS, 4(3), July 1982, 455-495.

[Page 1989] I. P. Page, and R. T. Jacob: *The Solution of Mutual Exclusion Problems which can be Described Graphically*, The Computer Journal, 32(1), Feb. 1989, 45-54.

[Paker 1983] Y. Paker, and J. -P. Verjus (ed.), *Distributed Computing Systems - Synchronization, Control, and Communication*, Academic Press, 1983.

[Paker 1987] Y. Paker, J. P. Banatre, and M. Bozyigit: *Distributed Operating Systems - Theory and Practice*, Springer-Verlag, Berlin, 1987.

[Parnas 1975] D. L. Parnas: *On a Solution to the Cigarette Smoker's Problem (without conditional statements)*, CACM, 18(3), March 1975, 181-183.

[Patil 1971] S. S. Patil: *Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes*, Computation Structures Group Memo 57, Project MAC, M.I.T., Cambridge, MA, Feb. 1971.

[Peterson 1979] G. L. Peterson: *Concurrency and Complexity*, Technical Report 59, Department of Computer Science, The University of Rochester, Rochester, New York, Aug. 1979.

[Peterson 1980] G. L. Peterson: *New Bounds on Mutual Exclusion Problem*, Technical Report 68, Department of Computer Science, The University of Rochester, Rochester, New York, Nov. 1980.

[Peterson 1981] G. L. Peterson: *Myths about the Mutual Exclusion Problem*, Information Processing Letters, 12(3), June 1981, 115-116.

[Peterson 1981b] J. L. Peterson: *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[Peterson 1982] G. L. Peterson: *An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem*, ACM TOPLAS, 4(4), Oct. 1982, 758-762.

[Peterson 1983a] G. L. Peterson: *Concurrent Reading While Writing*, ACM TOPLAS, 5(1), Jan. 1983, 46-55.

[Peterson 1983b] G. L. Peterson: *A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables*, ACM TOPLAS, 5(1), Jan. 1983, 56-65.

[Quinton 1986] P. Quinton, and J. -P. Verjus: *Distributed Synchronization of Parallel Programs - Why and How?*, in Parallel Algorithms and Architectures, edited by M. Cosnard, Y. Tobert, P. Quinton, and M. Tchuente, Elsevier Science Publishers B.V., North-Holland, Amsterdam, 1986, 109-125.

[Rana 1983] S. P. Rana: *A Distributed Solution of the Distributed Termination Problem*, Information Processing Letters, 17, 1983, 43-46.

[Raymond 1989] K. Raymond: *A Tree-Based Algorithm for Distributed Mutual Exclusion*, ACM TOCS, 7(1), Feb. 1989, 61-77.

[Raynal 1986] Michel Raynal: *Algorithms for Mutual Exclusion*, North Oxford Academic, 1986.

[Reed 1979] D. P. Reed, and R. K. Kanodia: *Synchronization with Eventcounts and Sequencers*, CACM, 22(2), Feb. 1979, 115-123.

[Rettberg 1986] R. Rettberg, and R. Thomas: *Contention is no Obstacle to Shared-Memory Multiprocessing*, CACM, 29(12), Dec. 1986, 1202-1212.

[Ricart 1981] G. Ricart, and A. K. Agrawala: *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, CACM, 24(1), Jan. 1981, 9-17.

[Ricart 1983] G. Ricart, and A. K. Agrawala: *Author's Response*, CACM, 26(2), Feb. 1983, 147-148.

[Sanders 1987] B. A. Sanders: *The Information Structure of Distributed Mutual Exclusion Algorithms*, ACM TOCS, 5(3), Aug. 1987, 284-299.

[Schmid 1974] H. A. Schmid: *On The Efficient Implementation of Conditional Critical Regions and the Construction of Monitors*, Acta Informatica, 6, 1976, 227-249.

[Schneider 1982] F. B. Schneider: *Synchronization in Distributed Programs*, ACM TOPLAS, 4(2), April 1982, 179-195.

[Silberschatz 1988] A. Silberschatz, and J. L. Peterson: *Operating System Concepts*, Addison-Wesley Publishing Co., 1988.

[Silberschatz 1991] A. Silberschatz, J. L. Peterson, and P. B. Galvin: *Operating System Concepts*, Third Edition, Addison-Wesley Publishing Co., 1991.

[Singhal 1988] M. Singhal: *A Dynamic Information Structure Mutual Exclusion Algorithm for Distributed Systems*, Technical Report No. OSU-CISRC-3/88-TR9, The Ohio State University, 1988.

[Singhal 1989] M. Singhal: *A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed Systems*, IEEE Transactions on Computers, 38(5), May 1989, 651-662.

[Singhal 1989b] M. Singhal: *Deadlock Detection in Distributed Systems*, IEEE Computer, Nov. 1989, 37-48.

[Spector 1984] A. Spector, and D. Gifford: *The Space Shuttle Primary Computer System*, CACM, 27(9), 1984, 874-900.

[Stark 1982] E. W. Stark: *Semaphore Primitives and Starvation-Free Mutual Exclusion*, Journal of the ACM, 29(4), October 1982, 1049-1072.

[Stone 1984] Harold S. Stone: *Database Applications of the Fetch-and-Add Instruction*, IEEE Transactions on Computers, C-33(7), July 1984, 604-612.

[Stone 1989] Harold S. Stone: *High-Performance Computer Architecture*, Addison-Wesley Publishing Co., 1989.

[Suzuki 1985] I. Suzuki, and T. Kasami: *A Distributed Mutual Exclusion Algorithm*, ACM TOCS, 3(4), Nov. 1985, 344-349.

[Tanenbaum 1988] A. S. Tanenbaum: *Computer Networks*, Second Edition, Prentice Hall, 1988.

[Thomas 1979] R. H. Thomas: *A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases*, ACM Transactions on Database Systems, 4(2), June 1979, 180-209.

[van de Snepscheut 1987] J. L. A. van de Snepscheut: *Fair Mutual Exclusion on a Graph of Processes*, Distributed Computing, 2, 1987, 113-115.

[Vantilborgh 1972] H. Vantilborgh, and A. van Lamsweerde: *On an Extension of Dijkstra's Semaphore Primitives*, Information Processing Letters, 1, 1972, 181-186.

VITA

[redacted]