2003

Taxonomy of synchronization and barrier as a basic mechanism for building other synchronization from it

Pauline Braginton

TAXONOMY OF SYNCHRONIZATION AND BARRIER AS A BASIC MECHANISM FOR BUILDING OTHER SYNCHRONIZATIONS FROM IT

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Pauline Braginton

March 2003

TAXONOMY OF SYNCHRONIZATION AND BARRIER AS A BASIC MECHANISM FOR BUILDING OTHER SYNCHRONIZATIONS FROM IT

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

by

Pauline Braginton

March 2003

Approved by:

[redacted]

Dr. Ernesto Gomez, Chair, Computer Science

3/14/2003

Date

[redacted]

Dr. Javier Torner

[redacted]

Dr. George Georgiou

ABSTRACT

A Distributed Shared Memory (DSM) system consists of several computers that share a memory area and has no global clock (since each machine is slightly slower of faster that the others). Therefore, an ordering of events in the system is necessary. For example: the messages in this system need to be delivered in the exact order in which they are sent since a message cannot be received before it is sent.

Synchronization is a mechanism for coordinating activities between processes, which are program instantiations in a system. To synchronize processes in a DSM system, ordering of events is a necessary task in order to ensure mutual exclusion (it makes sure that if one process is in the Critical Section then the other processes will be excluded from doing the same thing). Critical Section (CS) is the part of the program where the shared memory is accessed. In order to coordinate between processes in the system and allow them to broadcast and deliver messages in order, a consensus needs to be reached.

A consensus can be found in an area of memory that allows processes to reach a common decision despite

potential process failures. Failures can happen when a process stops participating in the algorithm (Benign failure) or when a process sends incorrect information (Byzantine failure). However, in the presence of faults, consensus can't be reached in an asynchronous system, where no upper bound on message delay is assumed. In asynchronous system, there is a possibility of non-termination; there is no way of knowing if a process has crashed or if it's just run very slow and will eventually send an answer.

Semaphore is a synchronization tool that helps to overcome difficulties of CS problems. The semaphore's atomic (uninterruptible) operations (wait and signal) guarantees mutual exclusion and the order in which processes are allowed to enter the CS does not matter. However, in a DSM system the order of events is essential and in this paper a deterministic (repeatable in the same sequence) semaphore algorithm is introduced.

Another common synchronization construct is a barrier. A barrier is a global synchronization point in a parallel program where the number of processes expected to arrive is known in advance. When the last process arrives, all processes execute, release, and reach the barrier again. A simple implementation of barrier synchronization can result in memory hot-spots, especially in large scale shared-memory multi-processors containing hundreds of processors and memory modules communicating through an interconnection network. Different solutions to solve this problem are introduced.

If a barrier is a form of synchronization, can the reverse also be true; can any form of synchronization be a barrier?

When considering the facts about semaphore and barrier synchronization, is it possible to reverse the synchronization operation? If a semaphore will be used as a barrier, and a barrier releases its processes all at once, we will get a chaotic result. All the processes will enter the CS all at once, and the operation will violate mutual exclusion in a semaphore. This paper considers these obstacles.

By controlling the way processes are released to CS, I show that a semaphore can be a barrier. For the purpose of proof, it needs to be stated that although there are many types of semaphores, one algorithm type is critical in defining this operation. A counter semaphore is needed to guarantee a deterministic operation (verifiable and repeatable algorithm in the same sequence).

My new algorithms are using a counter semaphore and they work as follows: each process identification (id) is added and accumulates to a list while waiting to enter CS. This is accomplished in a regimen order resembling a sergeant giving orders to his subordinates who are standing in a line; the sergeant gives out orders (the last process to arrive at the barrier) and the soldiers in line are the processes contained in the list. When the sergeant orders his subordinates to exit the line, they can leave the line in two ways: one by one (Algorithm2) or all at once (Algorithm1). The list (line of soldiers) forms a barrier, which deterministically releases the processes from it. Algorithm2 insures that each process is released after a signal or a message, forming a semaphore. Algorithm1 insures that all processes are released all at once, forming a barrier.

By the reduction of barrier and semaphore to each other, when two algorithms perform the same amount of work (steps), it can be proven that a barrier is a form of synchronization and the reverse (any form of synchronization is barrier) is also correct. Both algorithms: barrier and semaphore, will be theoretically proven to perform the same amount of work in an order of O(n) steps. Therefore, the reverse can also be true; if a barrier is a form of synchronization, any form of synchronization can be a barrier.

ACKNOWLEDGMENTS

I would like to begin by Thanking My Advisor, Dr. E. Gomez, Dr. J. Torner, and Dr. G. Georgiou for being the members of my committee. I am grateful for the guidance and helpful discussions by my advisor Dr. Gomez. I thank Dr. Torner for always been willing to provide encouragement, and for Dr. Georgiou for being a great instructor.

I am grateful to the Department of Computer-Science at California State University San Bernardino, for providing me with the MII scholarship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

BACKGROUND

1.1 Introduction

A Distributed Shared Memory (DSM) system consists of several computers that share a memory area. Processes (i.e. executing programs) in a DSM need to communicate with each other frequently. There are three issues concerning the need for communication between processes: how one process can pass information to another, how to make sure that two or more processes do not get into each other's way when engaging in critical activities (when two processes try to get hold of the same resource), and how to sequence properly when dependencies are present. Processes that act together in a group form a group communication.

A group is a collection of processes that act together in some system. When a message is sent to the group itself, all members of the group receive it. It is a form of "one-to-many" communication and is constructed with point-to-point communication where there is one sender and many receivers. Group communication is a communication mechanism in which a message can be sent to multiple receivers in one operation. It requires two properties: the first one is atomic broadcast, which insures that a message sent to the group arrives to all members of the group or to none of them and the second one is ordering, which means that messages are delivered in the exact order in which they are sent. For successful communication, processes need to cooperate and synchronize with one another.

Synchronization is a mechanism for coordinating activities between processes in the system: between sender and receiver, between joint activity of cooperating processes, and serialization of concurrent access to shared objects by multiple processes. Synchronization in a DSM system needs to coordinate between processes; although, no common clock or global time source exits.

A DSM system has no global clock since each machine is slightly slower or faster than the other. To synchronize a system without global agreement on time, Lamport (1995) suggested that all processes need to agree on the order in which events occur (happened-before) rather than trying to agree on real time. Ordering of events in the system is a necessary task in order to ensure mutual exclusion in a DSM system.

When a process has to read or update shared data, it first enters a Critical Section (CS), the part of the program where the shared memory is accessed. While a process is in the CS, mutual exclusion ensures that no other process will use the shared data (CS) at the same time. A process-coordinator, perform some special role, insures mutual exclusion in a DSM. Different algorithm designs on how the coordinator achieves mutual exclusion (Lamport, Richard and Agranta, and the Token Ring algorithm [Tanenbaum, 1995]) and how to elect the coordinator will be discussed in this paper. The coordinator, who grants permission to enter the CS, needs to be elected by the processes in the system. Two ways of electing a coordinator will be considered. One way is "The Bully Algorithm" which locates the process with the highest process number. The second way is the "Ring Algorithm" which builds an ELECTION message containing its own process number and sends the message to the processes next to it in a circular fashion. Eventually, the process receives an incoming message containing its own process number and the message type becomes the coordinator. At this point, the coordinator circulates once again to inform everyone else who the coordinator is (Tanenbaum, 1995).

In order to coordinate between processes in a system, there is a need to agree on something. For example: electing a coordinator, deciding whether to commit a transaction or not, synchronizing, and so on.

A consensus is an area in memory that allows processes to reach a common decision in order to communicate. A Consensus allows processes to broadcast and deliver messages in such a way that processes agree on the set of messages they deliver and on the order of message deliveries. In order to reach consensuses between processes, the order in which the events occur is essential in solving synchronization problems. For example: a message cannot be received before it is sent. When all processes agree on the order of events, mutual exclusion can be reached.

The general goal of distributed agreement algorithms is to have all the non-faulty processors reach consensus on some issues and to do that within a finite bounded time. There are two kinds of process failures: one kind is when a process stops participating in the algorithm

(Benign failure), and the other kind is when a process sends incorrect information (Byzantine failure). Consensus in the presence of faults is difficult. Therefore, the consensus algorithm design depends on the type of system that is considered. Is the system synchronous or asynchronous?

In a synchronous system where the message system is completely reliable, the following assumptions are made: restriction on time bound, only processes are subject to fail, any delivered message can arrive without errors, and the communication graph is connected. In a Benign failure, a death of a process is immediately detected since there is a finite time on a message delay.

In an asynchronous system, no assumptions can be made about upper bound on message delays; therefore, a slow process cannot be distinguished from a dead process. According to Fischer et al., (1985) a totally asynchronous system can't tolerate even a single unannounced process death. Also, according to Lynch et al. (1986), exact agreement in asynchronous system can't be reached. If assuming lower bound, only approximate agreement can be reached with the condition that the total number of processes is more than five times the number of possible faulty processes. In addition, Chandra D. and Toueg S. (1996), introduce failure detectors algorithm. This is a mechanism that maintains a list of processes that are suspected to have crashed and can be infinitely adding and removing suspected processes from the list. This implies an infinite upper bound on message delay, where exact agreement with guaranteed termination is not possible in an asynchronous system.

Semaphore synchronization protects critical sections by its two atomic (uninterruptible) operations: wait and signal. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or has blocked (wait). If one or more processes were unable to complete an earlier operation, one of them is chosen by the system and is allowed to complete its operation. However, the order in which processes are allowed to enter to CS does not matter.

Another common synchronization operation is a barrier. Upon reaching a barrier, a process must stall, (wait until all participating processes reach the barrier). After the last process reaches the barrier, all the processes are released. The barrier brings a group of processes to a known global state before proceeding to a new phase of computation. In a shared-memory system with multiple processors, typical implementations of a barrier are done with the use of spin on a variable. When a process spins on a variable, it spins on a loop until the shared variable "release" is read 1. In a large scale shared-memory, especially if multi-processors containing hundreds of processors communicate through a shared memory, this implementation results in memory hot-spots. For this reason, different solutions are introduced.

If a barrier is a form of synchronization, can the reverse apply? Is every synchronization a barrier? Can it be proven that a semaphore is a barrier? Can they perform the same amount of work (N steps)?

In focusing on a DSM software solution, there is a need to find a way, so that a semaphore will release its processes to a CS in sequential order. However, the standard semaphore is not deterministic (not repeatable in the same sequence). In addition, if we use a semaphore as a barrier, and a barrier releases its processes all at once, we will have bunch of processes enter into the CS all at once. Now, we'll get a chaotic result, which violates mutual exclusion in semaphore. Therefore, an algorithm that will take advantage of a Counter Semaphore can work in a proven theoretical solution.

To create a deterministic semaphore, a Counter semaphore can be used in the algorithm as follows: a process identification (id) is added and accumulated to a list while they are waiting to enter the CS. The list of processes forms a barrier, which waits for its last process to arrive. Once the last process arrives, the process can deterministically (in order) be released. By controlling the way the processes are released from the accumulated list (barrier), we can see that a semaphore and barrier are reversible.

If processes are released from the list (barrier) one by one, after a signal or a message, we can make a semaphore from a barrier with a work of O(N) steps. Also, if processes are released from the list (barrier) all at once with a "for loop", we can make a barrier from semaphore with a work of O(N)steps. When considering these facts it is possible to reverse the assumption and theoretically prove that any synchronization can be expressed as a barrier.

1.2 Purpose of the Thesis

The purpose of the thesis is: if a barrier is a form of synchronization, can we express any synchronization as a barrier?

1.3 Context of the Problem

The context of the problem was to reduce barrier and semaphore to each other.

1.4 Significance of the Thesis

Since a semaphore is non-deterministic, the significance of the thesis is to find a mechanism in a DSM system, so that a semaphore will release its processes to a CS in a sequential order. In addition, a barrier waits for the last processes to arrive and then release all its processes from a global point. If a semaphore operates like a barrier, there is a danger that all processes will enter CS all at once and will violate mutual exclusion. The way we can control this chaos, is bye regulating the way the processes are released from the list. Both, barrier and semaphore are reducible to each other and require the same amount of work of order O(n).

1.5 Assumptions

The following assumptions were made regarding the thesis:

1. The system is asynchronous.
2. Assume no failure.
3. Number of processes, N, that we are waiting for is known ahead.

1.6 Limitations and Delimitations

During the development of the thesis, a number of limitations and delimitations were noted. These limitations and delimitations are presented in the next section.

1.6.1 Limitations

The following limitations apply to the thesis:

1. The reduction of semaphore and barrier to each other can not work with a binary semaphore.

1.6.2 Delimitations

The following delimitations apply to the thesis:

1. The reduction of barrier and semaphore to each other can only work with a counter semaphore. This way a process can register and add its process id/priority to a list. The processes that are accumulated in the list form a barrier.
2. When processes are released all at once (Algorithm1), we can make a barrier from a semaphore by releasing the processes within a "for loop". Also, when the processes are released one by one, (Algorithm2) we can make a semaphore from a barrier by releasing the processes one at a time using a signal operation.

1.7 Definition of Terms

The following terms are defined as they apply to the thesis:

Atomic broadcast - It designs in a way that when a message is sent to a group, it will either arrive correctly at all members of the group, or at none of them.

Atomic operation - It guarantees that once an operation has started, no other processes can interfere, until the operation has completed or blocked.

Benign/Fail-stop Failures - In this kind of failure, a faulty process crashes, stops operating, but does not perform wrong operations (deliver messages that were not sent).

Byzantine Failures - Byzantine failures can send messages when it is not supposed to, make conflicting claims to other processes, act dead for a while and then revive itself, etc.

Clock skew - It is the difference in time values.

Consensus - It has an area in memory that is identical in every process. All processes must agree on the same single value.

Critical Section (CS) - It is a part of the program where the shared memory is accessed.

Deterministic algorithm - It is verifiable and repeatable in the same sequence with the assumption of a given finite group of processes.

Diameter - This is the longest path between any two nodes.

Distributed Shared Memory (DSM) system - This is a collection of individual computing devices that can communicate with each other while sharing a memory address space.

Fault-tolerant system - It is a system that can maintain a reasonable number of process or communication failures.

Group Communication - These are processes that act together in a group.

Logical clock - It is not the actual clock in the usual sense. It agrees on the order of which event happed first.

Mutual Exclusion - This is a way of insuring that if one process is using a shared variable or file, the other processes will be excluded form doing the same thing.

Network - Collection of channels

carries its destination address inside it, and this address is used for routing.

Physical clock - It shows the real time.

Process - It is a program in execution.

Timeout - A period of time after, which an error condition is raised if some event hasn't occur. A common example is in sending a message. If the receiver does not acknowledge the message within some preset timeout period, a transmission error is assumed to occur.

T-resilient - When no more that t processes fail before or during execution.

1.8 Organization of the Thesis

The thesis is divided into four chapters. Chapter One provides an introduction to the context of the problem, purpose of the thesis, significance of the thesis, limitations and delimitations and definitions of terms. Chapter Two consists the taxonomy of synchronization. Chapter Tree documents the Methodology used in this thesis. Chapter Four presents the discussion from the thesis. Finally, the references.

CHAPTER TWO

TAXONOMY OF SYNCHRONIZATION

2.1 Introduction

Several techniques in solving synchronization problems in a DSM environment are described. Since there is no global clock in a DSM system, different computers have different frequencies. Therefore, Lamport (1995) suggested that all processes needs to agree on the order in which events occur rather then on the exact time. The use of a timestamp and the event of "happened before" relation meet the requirements for global time. Ordering of events in the system is a necessary task in order to ensure mutual exclusion in a DSM system. The following algorithm guarantees mutual exclusion: Lamport, Richard and Agranta, and the Token ring algorithm (Tanenbaum, 1995). A process-coordinator insures mutual exclusion in a DSM system. A coordinator is being elected by using the "The Bully Algorithm" and the "Ring Algorithm" (Tanenbaum, 1995). Reaching ordering in synchronization requires consensus between processes. A consensus in the presence of faults is difficult. Therefore, assumptions about the system (whether if it is synchronous or asynchronous) and on the kind of faults (Benign or Byzantine failure) that can occur will be discussed. According to Fischer et al, (1985) a totally asynchronous system can't tolerate even a single unannounced process death.

A barrier is a synchronization point where processes are forced to wait until all processes have arrived. Then they are all released simultaneously. The "Central Barrier" and the "Tree Barriers" algorithms are discussed (David & Singh, 1999). Different implementations of barrier synchronizations are also presented in this paper. Also, processes upon reaching a barrier are idle while waiting for other processes to reach the barrier. Thus, no useful work is done by the processes while waiting to synchronize at the barrier. The following algorithms: "Adaptive Combining Tree" and the "Fuzzy Barrier" by Gupta, (June 1989) are solutions to avoid latency at the barrier.

Semaphore synchronization is used to ensure that only one process at any time can be utilizing a resource. They can only be used in a system with a shared memory. If there is no shared memory, there is no CS problem and a semaphore is not needed.

2.2 Defining Synchronization

Processes in a DSM need to communicate with other processes by inter-process communication, exchange of data between one process and another, either within the same computer or over a network. There are three major components of a synchronization event. The first component is the acquire method: a method by which a process tries to gain the right to enter the critical section or proceed past the event synchronization. The second component is a waiting algorithm: a method by which a process waits for a synchronization to become available. For example: if a process tries to acquire a lock but the lock is not free, or to proceed past an event but the even has not yet occurred. The third component is the release method: a method for a process to enable other processes to proceed past a synchronization event. For example: an implementation of the Unlock operation, a method for the last process arriving at a barrier to release the waiting process, or a method for notifying a process waiting at a point-to-point event that the event has occurred.

Mutual exclusion ensures that certain operations on certain data are performed by only one process at a time.

In other words, when a process is using a shared variable or a file, the other processes are excluded form entering the CS. A semaphore is a synchronization algorithm that supports mutual exclusion. There are two ways in which a process is waiting to enter the CS: busy waiting and blocking. Busy-waiting means that the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another process changes the value of the variable and allows the waiting process to proceed. Unless there is a reasonable expectation that the wait will be short, busy-waiting wastes CPU time. Under blocking, the process does not spin but simply blocks (suspends) itself and releases the process if it finds that it needs to wait. It will be awakened and made ready to run again when the release it was waiting for occurs. Blocking has higher overhead since suspending and resuming a process involves the operating system, but it makes the processor available to other threads or processes that have useful work to do. Busy-waiting avoids the cost of suspension but consumes the processor and cache bandwidth while waiting.
Blocking is more powerful than busy-waiting because, if the process or thread that is being waited upon is not

allowed to run, the busy-wait will never end. Software implementations of synchronization constructs are usually included in system libraries.

Synchronization in a DSM system uses atomic exchange in message passing. A data transfer occurs whenever a data in one storage element is transferred into another. If one process sends a copy of a data (that is in the sender's address space) into a region of the receiver's address space, communication occur. A synchronization operation must take place to indicate that the value is ready to be read. The order of the event is of importance since we can't receive a message before it was sent. Events may be point-to-point, involving a pair of processes, or they may be global, involving all processes or a group of processes for example: a barrier [4].

A new definition about synchronization for deterministic and non-deterministic order of processes was established during a thesis discussion with Dr. E. Gomez. The new definition of synchronization arrives through the following process: assuming that there are two possible relations between two or more processes for any two given process. Processes can be executed at the same time and order, or they can be executed not in the same order. The time is defined as T, and two processes P and Q are in states j and k.

A semaphore is considered to be non-deterministic since the order of which the processes are allowed to enter to CS does not matter. And it may vary from one execution of the program to the next (what matters is that they do so one at a time). For example: process P_j can enter CS before process Q_k and can be expressed as $P_j < Q_k$. We need to measure time T such that for every event P_j, we can assign it a time value $T(P_j)$ and for every event Q_k we can assign a time value $T(Q_k)$. This must have the property that if $P_j < Q_k$, then $T(P_j) < T(Q_k)$. Also, Process Q_k can enter CS before processes P_j and can be expressed as $Q_k < P_j$. We can assign a time value $T(Q_k)$ and for every event P_j we can assign a time value $T(P_j)$. This must have the property that if $Q_k < P_j$, then $T(Q_k) < T(P_j)$. We can conclude that the time that process P can enter CS is not equal to the time that process Q can enter CS as follows: $T(P_j) \mathrel{!=} T(Q_k)$. If the order is not the same for each execution it is non-deterministic. In semaphore synchronization, processes can enter to CS in any order they wish, therefore it is considered a non-deterministic operation and it can be defined as follows: $T(P_j) \mathrel{!=} T(Q_k)$.

On the other hand, the ordering of processes can be considered to be deterministic if the event of happened before exists. Which means that if event of process P occurs before process Q, then $P_j < Q_k$ is true. Also, the time value for processes P assigned as $T(P_j)$ and the time value for process Q assigned as $T(Q_k)$. This time value must have the property that if $P_j < Q_k$, then $T(P_j) < T(Q_k)$. Therefore, order of events is deterministic and is defined as follows: $T(P_j) < T(Q_k)$.

A barrier is considered to be deterministic since all processes can be released at the same time or in order one after another. If process P is released at the same time as process Q, then $P_j = Q_k$. If time value T was assigned to P, and the same time value was assigned to Q, then $T(P_j)$ is equal to $T(Q_k)$. Therefore, barrier synchronization can be defined as follows: $T(P_j) = T(Q_k)$ or $T(P_j) < T(Q_k)$.

In a DSM system, there is no global time. However, time plays a major role in the synchronization of processes. A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension. However, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. Lamport showed that clock synchronization in a DSM system is possible by pointing out that what matters is that processes agree on the order in which events occur, rather then agreeing on the exact real time.

In order to synchronize (coordinate between processes) and insure ordering of events in the system, a set of formal rules describing how to transmit data, especially across a network is needed. A barrier is a point of global synchronization where all processes are released from that point. While some processes are waiting for all processes to arrive at the barrier, latency arises in the synchronization operation. Different algorithms to support synchronization problems of event ordering, mutual exclusion, consensus, barrier and semaphore will be introduced [20].

2.3 Event Ordering

It is sufficient that all machines agree on the same time despite the absence of a global clock (time) in a DSM system. According to Lamport, it is not essential that the time to agree on will be the real time. The concept of one event is happening before another in a DSM is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events.

2.3.1 Partial Ordering

The relation "happened before" is a partial ordering of the events in the system. Lamport define the expression a -> b as "a happens before b" which means that all processes agree that first event a occurs then event b occurs. If a and b are events in the same process, and a occurs before b, then a -> b is true. Also, if a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a -> b is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite amount of time to arrive. "Happens-before" is a transitive relation, so if a -> b and b -> c, then a -> c. If two events, x and y, happened in different process that do not exchange messages, then x -> y is not true, but neither is y -> x. These events are said to be concurrent, which means that nothing can be said about when they happed or which is first [20].

2.3.2 Total Ordering

To place a total ordering of events in the system, an abstract point of view of time is introduced. Since the definition cannot be based on a physical time, it must be based on the order in which events occur. This is done by assigning a number to an event. Where the number is thought of as the time at which the event occurred. Therefore, the clock condition must have the property that if $a < b$ then a time value $C(a) < C(b)$. If a is the sending of a message by one process and b is the reception of that message by another process, then $C(a)$ and $C(b)$ must be assigned in such a way that everyone agrees on the values of $C(a)$ and $C(b)$ with $C(a) < C(b)$. In addition, the clock time (C) must always go forward (increasing), never backward (decreasing). Corrections to the time can be made by adding a positive value, never by subtracting one. Each message carries the sending time according to the sender's clock. When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast-forwards its clock to be one more than the sender time. This operation is called a global clock. With one small time addition, this algorithm meets the requirements for global time. The time addition is such that between every two events, the clock must tick at least once. If a process sends or receives two messages in quick succession, it must advance its clock by one tick in between them. This description follows the illustration in figure 1:

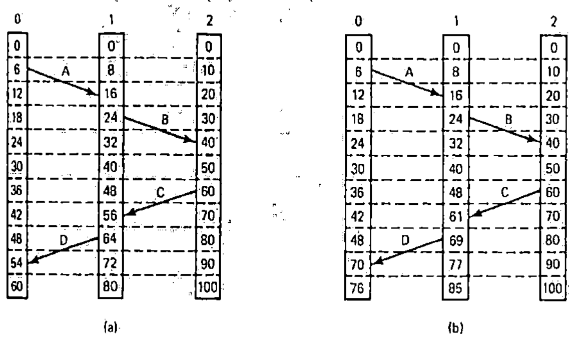


Figure 1. Lamport's Algorithm Corrects the Clocks

In some situations, an additional requirement is desirable. That is, no two events ever occur at exactly the same time. To achieve this goal, the number of the process in which the event occurs can be attached to the low-order end of the time, separated by a decimal point. Using this method, time events can be assigned in a distributed system, with the following conditions: if event a happens before b in the same process, C(a) < C(b). If a and b represent the sending and receiving of a message, C(a) < C(b).

For all events a and b, C(a) != C(b). This algorithm is a way to provide a total ordering for all events in the system [20].

2.4 Mutual Exclusion

Mutual exclusion is a way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. Four conditions are needed in order to maintain mutual exclusion:

1. No two processes may be simultaneously inside their CS.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its CS may block other processes.

4. No process should have to wait forever to enter its CS [21].

2.4.1 Test and Set Lock (TSL) Instructions

If we are given assistance by the instruction set of the processor we can implement a solution to the mutual exclusion problem. The instruction we require is called test and set lock (TSL). This instruction reads the contents of the memory word lock, stores it in a register and then stores a non-zero value at the address. This operation is guaranteed to be indivisible; no other process can access that memory location until the TSL instruction has finished. The first instruction copies the old value of lock to the register and then set lock to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Before a process enters its CS, a process calls enter_region, which does busy waiting until the lock is free. Then it acquires the lock and returns. After the CS, the process calls leave_region, which stores a 0 in lock [21].

2.4.2 Sleep and Wakeup

The pair Sleep and Wakeup are interprocess communication that block instead of wasting CPU time when they are not allowed to enter their CS. Sleep is a system call that causes the caller to block, that is, to be suspended until another process wakes it up. The Wakeup call has one parameter, the process to be awakened [21].

2.4.3 Semaphores

A semaphore S in an integer variable that can be accessed only through two atomic operations Wait and Signal. Meaning, when one process is testing or modifying the semaphore value, no other process can modify the value [18]. There are two types of semaphores: a binary semaphore and a counter semaphore.

2.4.3.1 Binary Semaphore. A binary semaphore is a semaphore with an integer value that can range only between 0 and 1 [9]. If each process does a wait (down) just before entering its critical region and signal (up) just after leaving it, mutual exclusion is guaranteed [21].

2.4.3.2 Counter Semaphore. In a counter semaphore, a process encounters a wait(s) command before it enters into the CS and a signal(s) when it exits the CS. In the wait(s) command, it checks to see if the CS is accessible. If it is, it blocks the other processes by decrement the semaphore value (s.value--;) and enters into the CS. If another process tries to enter CS, it adds its process id into the semaphore list and blocks itself, wait. The process, who exits the CS, performs signal(s) command. It increments the counter, removes one of the process's id from the semaphore list, and wake it up. There can be several ways of choosing a process from the s.list: FIFO, Priority, Size, etc [9]. For example,

```
Process 1                     Process 2
{...                          {...
}                             }
Wait(s)                       Wait(s)
CS                            CS
Signal(s)                     Signal(s)
```

2.4.3.3 Semaphore Counter Implementation. The semaphore operation can be defined as follows:

```
Type of struct
 {
   /*Each semaphore has an integer value and a list of processes */

   Int value;
   Int Sempahroe_List[max_proc];
} semaphore

Wait(S):
        S.Value--;
        If (S.Value < 0)
        {
          S.Semaphore_List <- pid;   /* enter process id into the semaphore list. */
          block;
        }

Signal(S):  /*A signal operation removes one process from the list of waiting processes,
              and awakens that process. */

        S.Value++;
        If (S.Vlaue =< 0)
        {
          remove process P from the S.L;
          wake up P;
        } [9].
```

Figure 2. Counter Semaphore

2.4.4 When We Don't Have Shared Resource

If there is no shared resource, there is no CS problem. Therefore, there is no need for a semaphore. When there is no shared resource, each processes has it's own memory space and the scheduler takes a processes, run it, and then make a context switch to a different process.

2.5 Mutual Exclusion in a Distributed Shared Memory System

To achieve mutual exclusion in a DSM system with a single processor, several algorithms are being introduced: One is the centralized algorithm, where one process is elected as a coordinator. Whenever a process wants to enter a CS, it sends a request message to the coordinator and the coordinator sends back a reply granting permission (if CS is available) to enter CS. Another one is the Lamport's algorithm. It presents total ordering of events in the system, where every process agrees on the order of the timestamps. And the last one is the token ring algorithm. When a process acquires the token from its neighbor, it can enter the CS, does its work, leaves the region, and pass the token to the next node.

2.5.1 A Centralized Algorithm

The centralized algorithm is the simplest way to achieve mutual exclusion. One process is elected as the coordinator. Whenever a process wants to enter a CS, it sends a request message to the coordinator stating which CS it wants to enter and asking for permission. If no other process is currently in that CS, the coordinator sends back a reply granting permission. When the reply arrives, the requesting process enters the CS. Now, if another process asks for permission to enter the same CS, the coordinator knows that a different process is already in the CS and cannot grant permission. In this case, the coordinator will either block the waiting process or reply "permission denied" while placing the request in a queue. When a process exits the CS, it sends a message to the coordinator releasing its exclusive access. The coordinator then, takes the first item off the queue (of the deferred request) and sends that process a grant message. This description follows the illustration in figure 3:

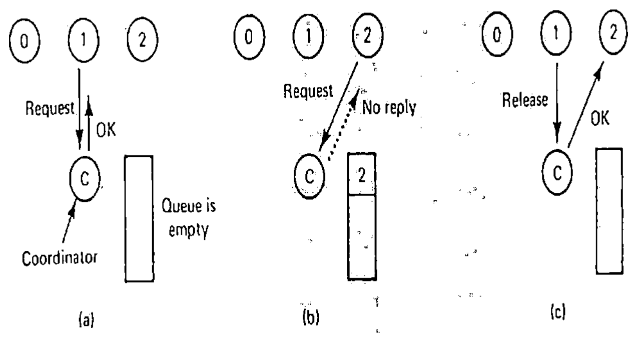


Figure 3. Centralized Algorithm

This algorithm guarantees mutual exclusion because the coordinator only lets one process at a time into each CS. The requests are granted in the order in which they are received and no process ever waits forever. This method requires only three messages to enter and leave a CS: a request, a permission to enter, and a release to exit CS.

The disadvantage of this algorithm approach is that the coordinator is a single point of failure. Therefore, if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordination from permission denied since in both cases no message comes back. In a large system, a single coordinator can become a performance bottleneck.

2.5.2 Distributed Algorithm

The distributed algorithm, by Ricart and Agrawala's, requires a total ordering of all events in the system. Which means, for any pair of events such as messages, it must be unambiguous which one happened first. When a process wants to enter a CS, it builds a message which contains the name of the critical region it wants to enter, its process number, and the current time. It then sends a message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable, that it, every message is acknowledged. When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. These cases have to be distinguished: first, if the receiver is not in the CS and does not want to enter it, it sends back an OK message to the sender. Second, if the receiver is already in the CS, it does not reply but instead it queues the request. Third, if the receiver wants to enter the CS but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If the receiver's message has a lower timestamp, the receiver queues the (incoming) sender's request and sends nothing.

After sending out requests asking permission to enter a CS, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the CS. When it exits the CS, it sends OK messages to all processes on its queue and deletes them all from the queue. In case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps. This description is follows the illustration in figure 4:

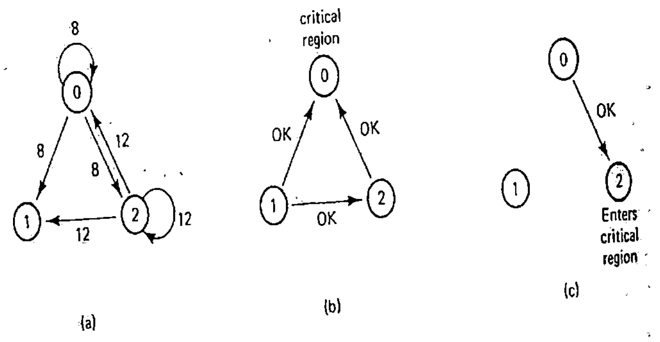


Figure 4. Distributed Algorithm

This algorithm required n - 1 request messages, one to each of the other process, and an addition n - 1 grant messages, for a total of 2(n - 1).

The disadvantage of this algorithm is that if any process crashes, it will fail to respond to requests. This silence will be interpreted, as denial of permission and the process cannot enter CS.

2.5.3 Token Ring Algorithm

Another approach to achieve mutual exclusion in a DSM system is the token ring algorithm. This logical ring is constructed in a way that each process is assigned a position in the ring by a numerical order of the network addresses or some other means. Each process has to know who is next in line after itself. When the ring is initialized, process 0 is given a token. The token circulates around the ring and is passed form process k to process k+1 in point-to-point messages.

When a process receives the token form its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the CS, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring. This description follows the illustration in figure 5:

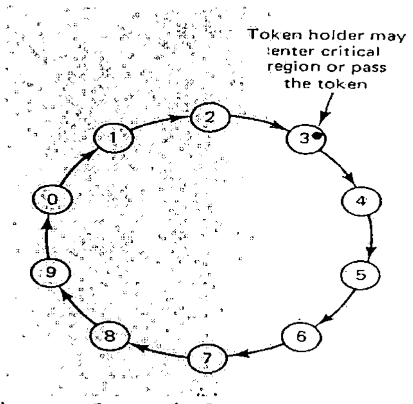


Figure 5. Token Ring Algorithm

In this algorithm, only one process has the token, so only one process can be in a CS. Since the token circulates among the process in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter and leave one critical region. The time varies from 0 (token just arrived) to n - 1 (token just departed).

Advantages: this algorithm allows only one process in CS at a time. A dead process will be detected when its neighbor tries to give it the token and fails. Also, a dead process can be removed from the group, and the token holder can throw the token to the next member down the line, or the one after that, if necessary.

Disadvantages: If the token is ever lost, it must be regenerated. However, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. If a token has not been spotted for an hour does not mean that it has been lost because someone may still be using it [20].

2.6 Election Algorithms in Distributed Shared Memory System

Many DSM systems require that one process will act as a coordinator. It is done by a group of processes that choose one among them to be the leader (coordinator). The existence of a leader is helpful among processes communication and is helpful in achieving fault-tolerance or in a deadlock situation. For example: when a deadlock is created, due to processes waiting in a cycle for each other, or in case of a non-responding process, this can be broken by electing one of the processes as a new leader and removing the faulty process from the cycle [2]. An election algorithms attempt to locate the process with the highest process number and designate it as the coordinator. Assume that every process knows the process number of every other process. The processes do not know which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all process agreeing on who the new coordinator is to be.

2.6.1 The Bully Algorithm

When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows: P sends an election message to all processes with higher numbers. If no one responds, P winds the election and becomes a coordinator. If one of the higher-ups answers, it takes over and P's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator. If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. This description follows the illustration in figure 6:

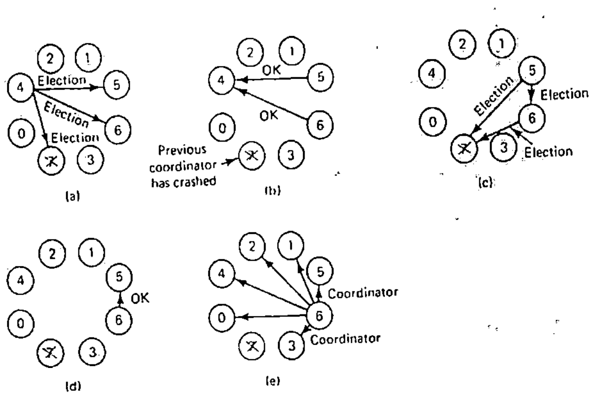


Figure 6. Bully Algorithm

2.6.2 Ring Algorithm

Another election algorithm is based on the use of a ring, but without a token. Rings are a convenient structure for message passing systems and correspond to physical communication system [2].

Assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an election message containing its own process number and sends the message to its successors. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds it's own process number to a list-message. Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At this point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members (from the list-message) of the new ring are. When this message has circulated once, it is removed and everyone goes back to work. The new coordinator does his job in achieving mutual exclusion when a process wants to enter CS [20].

2.7 Barrier Synchronization

A barrier is a synchronization point in a parallel program at which all processes participating in the synchronization must arrive before any of them can proceed beyond the synchronization point [20]. A software implementation of the barrier mechanism using shared variables, especially in a shared-memory system with multiple processors, has tow major drawbacks: one is the execution of the barrier results in hot-spot access. The second is that processes which are waiting for other processes to reach the barrier cannot do any useful work. While they are waiting, they are typically spin on a lock and waist CPU time.

A typical implementation of a barrier are created with the use of spin on a lock. A lock that uses busy waiting is called a spin lock. When a process spins on a lock (variable), it spins on a loop until the shared variable "release" is read 1. The problem is that in a large scale shared-memory, especially if multi-processors containing hundreds of processors communicate through a shared memory, many processes are spinning on the lock (variable), and the processes are continuously testing a variable until the value 1 appears in the release variable. Since only one process at a time can access the shared memory "release", this implementation results in memory hot-spot. Also, when a process is continuously spins on a variable (lock), it results in busy waiting, which wastes CPU time. This method is described as the central barrier in the next section follows the pseudocode in figure 7:

```
lock (counterlock); /*ensure update atomic */
if (count==0) release=0; /*first=>reset release */
count = count +1; /* count arrivals */
unlock (counterlock); /* release lock */
if (count==total) /* all arrived */
{
    count=0; /* reset counter */
    release=1; /*release processes */
}
else
{
    spin (release==1); /* wait for arrivals */

} [12].
```

Figure 7. Central Barrier

The central barrier algorithm uses busy wait.

2.7.1 Central Barrier

Central barrier is a software algorithm typically implemented using a single lock, a single counter, and a single flag, its length is of O(N). In this algorithm, a shared counter maintains the number of processes that have arrived at the barrier and is incremented by every arriving process. These increments must be mutually exclusive. Assuming that there are N processes, the last process that has arrived at the barrier checks to see if the counter is equal to N. If not, it busy-waits on the flag (flag = 0) associated with the barrier. If the process is equal to N, it writes the flag to release the N - 1 waiting processes (flag = 1), exit the barrier, perform computation, and enter the barrier again. However, if a process didn't see the flag change form the first barrier before others have reentered the barrier for the second time, it will continue to wait for the flag to change to 1. It will never leave the spin loop form the first barrier while the other processes may have already entered the second instance of the barrier, and the first of this will reset the flag to 0. But, the flag will never reset to 1 since the previous processes is still spinning on the flag.

Therefore, it is important to insure that all the processes have to exit the previous barrier before entering a new instance of a barrier. A solution to use another counter to count the processes that leave the barrier will increase latency and contention. Sense reversal, which is described next, is a better solution for this problem.

2.7.2 Centralized Barrier with Sense Reversal

This algorithm prevents processes form re-entering the barrier before all processes have exited. It also uses spin on a release. The use of a second counter to count the leaving processes can incur latency and contention. An alternate solution would be to have the processes wait for different flag values on consecutive instances of the barrier. A private variable is used per process to keep track of which value to wait for in the current barrier instance. A process needs two values 0 and 1 to toggle between each time. And they can be toggled only when all processes reached the barrier. The value of the flag is only changed once when all processes have reached the new barrier instance [4]. This description follows the pseudocode in figure 8:

```
BARRIER (bar_name, p)
{
   local_sense=!(local_sense); /* toggle private sense variable */
   LOCK(bar_name.lock);
   Mycount = bar_name.couter++; /*mycount is a private variable */
   if (bar_name.count == p)
   {
      UNLOCK (bar_name.lock);
      Bar_name.counter = 0; /*reset counter for next barrier */
      Bar_name.flag = local_sence; /*release waiting processes */
   }
   else
   {
      UNLOCK (bar_name.lock);
      While (bar_name.flag != local_sense) {}; /*busy_wait for release */
   }
} [4].
```

Figure 8. Centralized Barrier with Sense Reversal

2.7.3 Fuzzy Barrier

The fuzzy barrier algorithm, by Gupta, R., avoids hot-spot accesses to a shared memory. It also avoids a waiting process from not doing a useful work until all participating processors reach the barrier

This algorithm works as follows. The barrier includes a region of statements that can be executed by a process while it awaits synchronization. Upon reaching the first instruction in the region, a process is ready to synchronize. However, it can continue to execute the remaining instruction in the region even if synchronization has not yet occurred.

The barrier regions are constructed by a compiler and consist of several instructions such that a process is ready to synchronize upon reaching the first instruction in this region and must synchronize before exiting the region. The barrier can continue to execute the remaining instructions in the region even if synchronization has not yet occurred. The larger the barrier region is, the more likely it is that none of the processes will have to stall.

Instruction streams consist of barrier regions and non-barrier regions. Where streams with no barrier regions have no barrier synchronization and the barrier region forces the processes to synchronize.

The fuzzy Barrier functions as follows: no processes can execute an instruction form its respective non-barrier region (U2) following the barrier region until all processes have executed the instructions in their respective non-barrier regions (U1) preceding the barrier region. This description follows the illustration in figure 9:

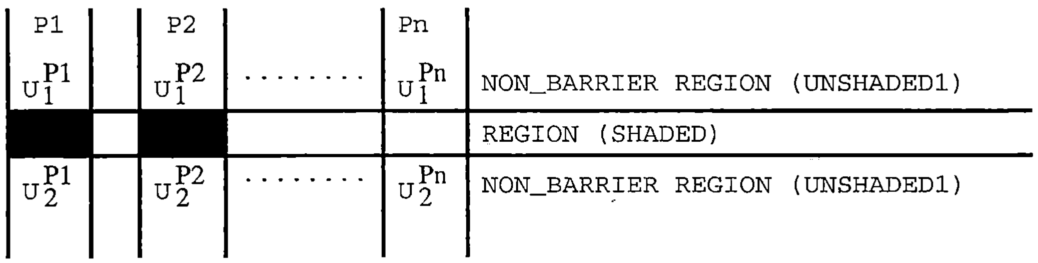


Figure 9. Fuzzy Barrier

There are a few conditions that hold in a fuzzy barrier for entering a region, exit a region, synchronize and stalling. A process is considered to have exited a region (barrier or non-barrier) of a stream if it has completed the execution of all the instructions in that region. A process is considered to have entered a region if it has started the execution of an instruction form that region. Processes can synchronize at the barrier if and only if they have all exited their respective non-barrier regions preceding the barrier region. A process can enter a non-barrier region following a barrier region if and only if synchronization has occurred. Thus, if the synchronization has not occurred when the process exits the barrier region, it is not allowed to enter the non-barrier region and must idle, and the execution of the stream is stalled. In short, for a process to exit a non-barrier region it must complete all instructions before it can enter the barrier region because of data dependency (that is forced by iterations). A process that enters a barrier region start executes instructions from this region and there is no data dependency. This region forces all processes to synchronize and then, only after the last processes arrived, the processes can enter the non-barrier region again.

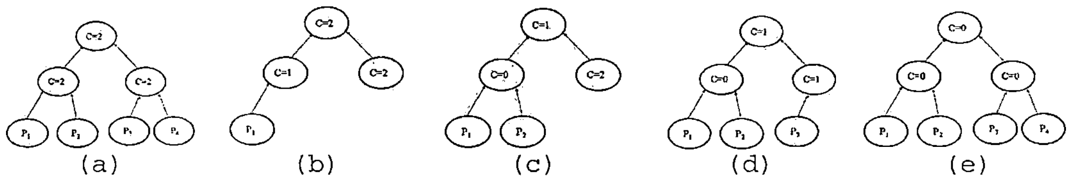
In order to construct the barrier and non-barrier regions the instructions that must be in the non-barrier regions are identified. These instructions are referred to as the marked instructions. All the instructions that are starting with the first marked instructions and ending at the last marked instruction are included in the non-barrier region, and the remaining instructors are form the barrier region.

Hot -spot accesses are avoided as the mechanism does not rely upon shared memory to achieve synchronization. This is done by implementing a mechanism in hardware, where instruction streams are detected by the hardware to ascertain when a process is ready to synchronize. All participating processes are simultaneously informed of this even, and when all of the processes have reached the barrier, they simultaneously recognize that synchronization has taken place [10].

A simple distributed way to coordinate the arrival or release of processes is through a tree structure. By distributing the variables among different memory modules in the system the problem of memory contention is greatly reduced. But, the process that is waiting for the last process to arrive is busy waits on the variable. Therefore, the use of spinning on a lock method is still being used, which waist CPU time. The different ways of tree structures are introduce as follows:

2.7.4 Software Combining Tree

Assuming a combining tree for synchronizing N processes. The nodes of the tree represent variables allocated from different memory modules in the system. Each node contains a parent pointer, a counter that is initialized to d, which is the number of children of each node in the tree, and a notify filed used during the notification of synchronization. The number of processes (N) synchronizing at the barrier is an integer power of d ($N = d^k$). A process upon arriving at the barrier goes to the leaf node assigned to it and decrements the counter. If the counter is not zero there are other processes that have not reached the barrier and the process remains at that node and busy waits on the notify filed. If the counter is zero, then it is the last process to arrive at the node and it goes to the parent node and repeats the above process until a process decrements the counter at the root node to zero. Then, barrier synchronization occurs [11]. The following example will illustrate a binary combining tree, where the order in which four processes P_1, P_2, P_3, P_4 arrive at the barrier respectively, as shown in figure 10(a):



(a) Initial state. (b) P_1 decrements the counter (c) P_2 decrement the parent node. (d) P_3 decrement the counter (e) P_4 decrement the root node to 0.

Figure 10. Combining Tree

When process P_1 arrives at the barrier, it goes to the leaf node assigned to it and decrement the counter to 1, as shown in figure 10(b). When process P_2 arrives at the barrier, it goes to the leaf node assigned to it and decrements the counter (to 0). Then, P_2 goes to the root node and decrement the counter to 1, as shown in figure 10(c). When P_3 arrives at the barrier, it goes to the leaf node assigned to it and decrement the counter to 1, as shown in figure 10(d). Finally, when the last processes P_4 arrives at the barrier, it decrements the counter at the root node to 0 and barrier synchronization occurs, as shown in figure 10(e).

Arcading to Gupta and Hil, the recursive algorithm of an arrival of processes to determine whether all processes have reached the barrier is called the recognition phase. When a process decrements the counter at the root node to zero, barrier synchronization has occurred and the notification through the notify field is carried out. During the notification phase all processes are notified about the occurrence of synchronization so that they can continue execution.

The recognition phase needs to be considered two cases. The first case deals with the situation in which all processes arrive at the barrier simultaneously. The second case arises when one of the processes arrives later then all the other processes (Simultaneous arrival is defined when one of the processes arrives before the other enters the busy waiting stage).

During simultaneous arrival, the time to achieve barrier synchronization is $O(d\log_d N)$. This is because d processes arriving at a node must decrement counter one at a time. Since there are $\log_d N$ levels in the tree the total time spent in synchronizing is $O(d\log_d N)$.

During non-simultaneous arrival, if all but one of the processes has already arrived at the barrier, then the last process must decrement the counter from the lower most level to the root of the tree to detect synchronization. This takes $O(\log_d N)$ in the detection of barrier synchronization after the last process has arrived at the barrier.

For the notification phase it takes $O(d\log_d N)$ time. This is because the processes that reaches the root of the tree has to go through $\log_d N$ levels notifying the processes and at each level it ensures that each on the d - 1 processes receives the notification. In the pseudocode (figure 11) the synchronization instructions has the following form: <syncvar; test; oper>. The syncvar is an integer synchronization variable allocated in shared memory, test is a condition that is tested prior to performing the operation on the synchronization variable. If the test fails the operation is not formed and the outcome of the test is sent to the process. Process can issue the same instruction again or proceed with execution. A star on the test condition (test*) is used to indicate that the process will continue to issue the instruction till it succeeds.

```
typedef struct node
{
         int counter;             //syncvar initial value = d
         int notify;              //syncvar initial value = 0
         struct node *node;
}node;

ProcedureBarrier(node);
{
<node->counter; Null; Fetch(last)&Decrement>
if(last == 1)
{
/* d processes have arrived at node */
if(nod != root) Barrier(node.parent);
/* all processes have arrived at the barrier-begin
  notification */
node->notify = d-1;  /* notify siblings */
/* wait for all siblings to notice */
while(node->notify != 0);
/* reinitialize the current node */
node->counter = d;
}
else
{
   /* wait for notification and indicate receipt of
     notification */
   <node->notify; (>0)*; Decrement>
   /* wait for all siblings to notice */
   while(node->notify !=0);
   }
} [11].
```

Figure 11. Combining Tree-Pseudocode

This approach of combining tree also requires busy-waiting at the nodes. Assuming that four processes are arriving at the barrier in an order of P_1, P_2, P_3, P_4 respectively. The nodes of the tree are labeled by the processes busy-waiting at the nodes. When process P_4 arrives, it has to go from the bottom of the tree, in order to go to the root of the tree to recognize synchronization. The adaptive combining tree is a way to minimize latency at the barrier.

2.7.5 Adaptive Combining Tree

According to Gupta and Hil, the adaptive combining tree achieves the appropriate tree structure dynamically. The combining tree is originally organized as a binary tree so that it can exploit maximum parallelism during synchronization if the processes arrive simultaneously. The adaptive tree is organized so that no process has to visit multiple levels in the tree. When process P_4 arrives, it recognizes synchronization immediately after decrementing the counter at the root node. (It will go directly to the root instead of going from the bottom of the tree to the root, to announce that synchronization occurred). This description follows the illustration in figure 12:

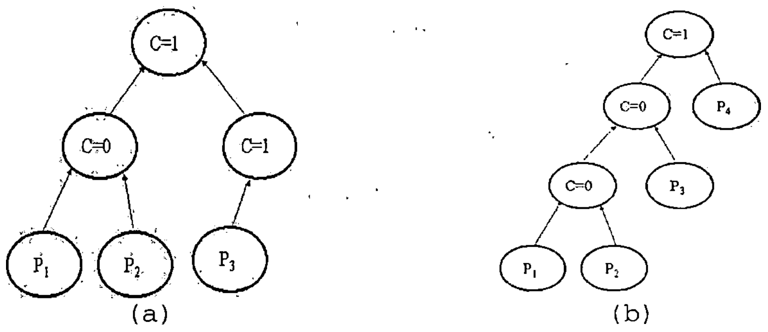


Figure 12. Adaptive Combining Tree

Figure 12(a) is one step before the last process (P_4) arrives at the barrier. This step is the same step as in figure 10(d). In figure 12(b) the last process to arrive at the barrier (P_4) goes directly to the root and recognize synchronization.

This barrier implementation eliminates the latency for barrier recognition in the non-simultaneous arrival case. A binary tree is used to minimize the recognition time in the simultaneous arrival case to $O(\log_2 N)$. The notification process is also modified, resulting in a barrier implementation that requires $O(\log_2 N)$ time each for recognition in the simultaneous arrival case and performing the notification. The barrier is correctly reinitialized, thus calling its repeated use in synchronization [11]. In the next tree barrier with local spinning, the processes are also busy-wait on a loop until the last process arrives. However, the flag to spin on can be allocated in the local memory of the spinning processor rather than the one that goes up to the parent level.

2.7.6 Tree Barriers with Local Spinning-Tournament Barrier

This is a binary combining tree. In this case, the barrier is performed without any atomic operations like fetch & increment. It uses simple reads and writes as follows: one process that arrives at each node is simply spins on an arrival flag associated with that node. The other processes that associates with that node, simply write the flag when it arrives. The process whose role was to spin now simply spins on the release flag associated with that node while the other processes now proceeds up to the parent node. This binary tree is called a "tournament barrier", since one process can be thought of as dropping out of the tournament at each step in the arrival tree.

Another way to ensure local spinning is to use P-node trees to implement a barrier among P processes, where each tree node (leaf or internal) is assigned to a unique process. The arrival and wake-up trees can be the same, or they can be maintained as different trees with different breaching factors. Each internal node (process) in the tree maintains an array of arrival flags, with one entry per child, allocated in that node's local memory. When a process arrives at the barrier, if its tree node is not a leaf, then it first checks its arrival flag array and waits until its children have signaled their arrival by setting the corresponding array entries. Then it sets its entry in its parent's arrival flag array and busy-waits on the release flag associated with its tree node in the wake-up tree. When the root process arrives and when all its arrival flag array entries are set, this means that all processes have arrived. The root then sets the release flags of all its children in the wakeup tree. These processes break out of their busy-wait loop and set the release flags of their children, and so on until all process are released [4]. This description follows the pseudocode in figure 13: (with the assumption of an arrival tree of branching factor 4).

```
Struct tree_node
{
    struct tree_node *parent;
    int parent_sense = 0;  /* set flag to 0 */
    int wkup_child_flags[2];  /* flags for children in wake-up tree */
    int child_ready[4];  /*flags for children in arrival tree */
    int child_exists[4];
}
                            /*nodes are numbered form 0 to P-1 level-by-level starting from the root */
struct tree_node tree[P];  /* each element (node) allocated in a different memory */
private int sense =1, myid;
private me = tree[myid];

barrier()
{
    while (me.child_ready is not all TRUE) {} ;  /* busy-wait */
    set me.child_ready to me.child_exits;  /* reinitialize for next barrier call */
                                        /* set parent's child_ready flag, and wait for release */
if (myid != 0)  /*  if process is not the root node */
    {
        tree[$\left\lfloor \frac{myid-1}{4} \right\rfloor$].child_ready[(myid-1) mod 4] = true;  /* find the parent and set arrival
                                                        flag array to true */
        while (me.parent_sense != sense) {};  /*still busy wait */
    }
me.child_pointers[0].parent_sense = me.child_pointers[1].parent_sense = sense;  /* release */
sense = !sense;
}[4]
```

Figure 13. Tree Barriers with Local Spinning-Tournament Barrier-Pseudocode

The above code will be followed by an the illustration in figure 14:

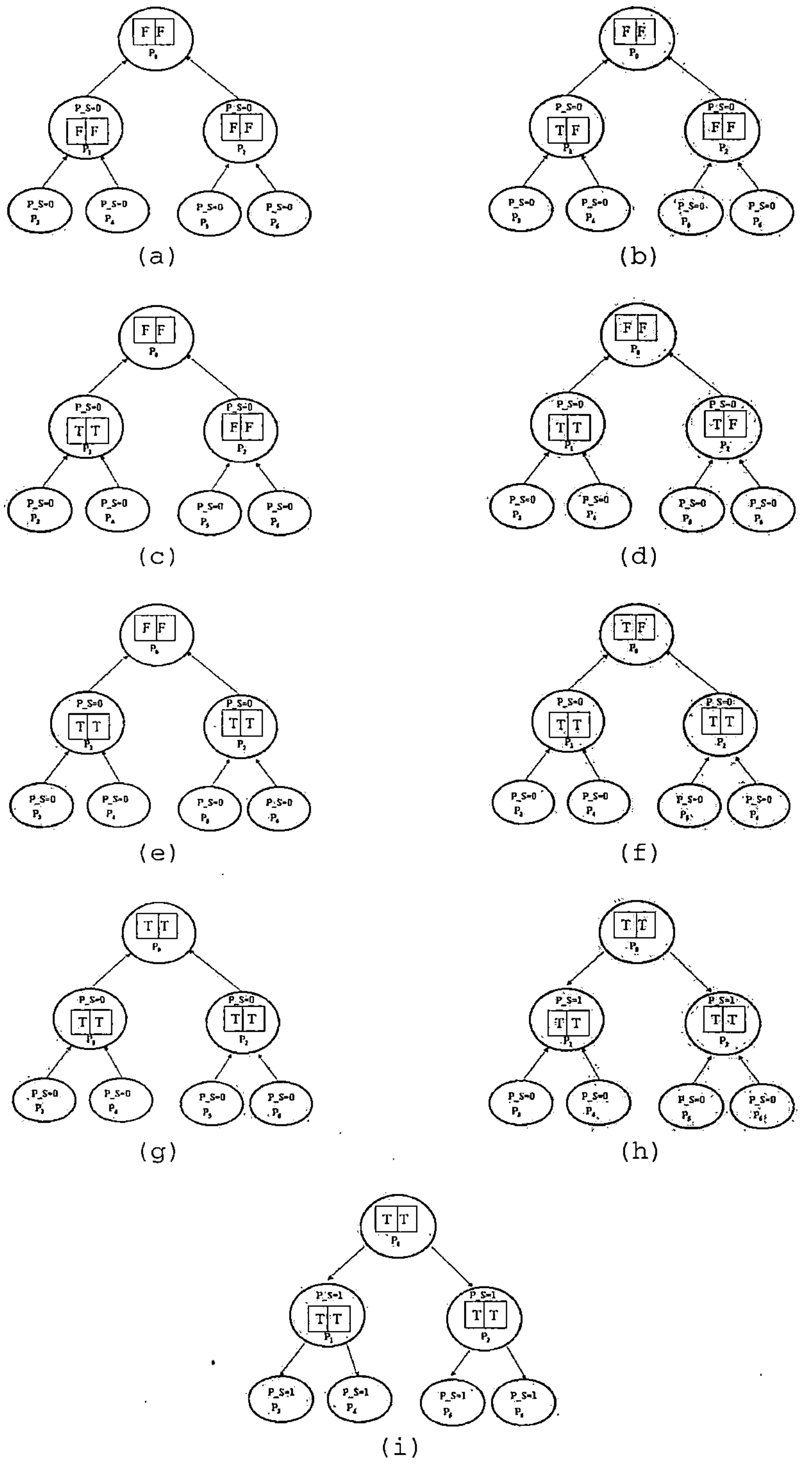


Figure 14. Tree Barriers with Local Spinning-Tournament Barrier

Figure 14(a) is the initial state. 14(b) process P_3 arrives at the barrier. 14(c) process P_4 arrives at the barrier. 14(d) process P_5 arrives at the barrier. 14(e) process P_6 arrives at the barrier. 14(f) process P_1 arrives at the barrier. 14(g) process P_2 arrives at the barrier. 14(h) the root (P_0) releases its children. 14(i) P_1 and P_2 release their children.

2.7.7 My Algorithm

In my algorithm's implementation instead of using a busy-wait on the lock, I use the block operation. Each process that arrives at the barrier registers itself into a list, which I called it a barrier-list. If a process is not the last one to arrive, it goes to a block state and "go to sleep". The last process that enters the barrier-list wakes up the processes from the list and they are released in an orderly manner from the barrier. In this case, the block operation does not waist CPU time, and the control is transferred to the CPU scheduler, which selects another process to execute.

2.8 Consensus

In order to investigate if a barrier can be a semaphore and if a semaphore can be a barrier, we need to investigate whether a barrier will work under fixed number of processes and whether or not a barrier is a consensus problem. Also, we need to know if we can arrive at consensuses in a synchronized network. Consensus and synchronization are related because we can synchronize if we can arrive on consensus. However, if we can agree on synchronization can we get consensus?

2.8.1 What is Consensus?

When a system is free of failures, an agreement can easily be reached among processes. A consensus protocol is correct if it meets the following conditions: consistency, validity and termination. The consistency condition exists if all processes agree on the same value and all decisions are final. The validity condition exists if the input value is valid (exist). The termination condition will take place if each process decides on a value within a finite number of steps. For example: assuming that two people are communicating with each other through an e-mail, trying to make an appointment. The consistency condition would be violated if one of them will wait alone. The validity condition would be violated if neither of them went to the meeting place, and the termination condition would be violated if they never agreed.

Reaching an agreement requires that each process has its own initial value and all non-faulty processors must agree on a single common value. For example: if the initial value of every non-faulty process is v, then they agree upon a common value v [18]. Another example is: that all process must agree on a binary value, based on the votes of each process. They must all agree on the same value, and that value must be the vote of at least one process (they can't decide on 1 when they all voted for 0).

In a DSM system, each process may vote on whether to commit a particular transaction, and if a single process votes no, then the decided value must be no and all processes must abort the transaction [2].

However, consensus in the presence of faults is difficult because the systems have different levels of synchrony or different kinds of failures. In considering a synchronous message passing systems (message system is completely reliable) two kinds of failures can occur: one kind is the Benign (Fail-stop), which causes a process to die at any time and stop participating in the algorithm.

The second kind are the Byzantine failures, where a process sends incorrect information possibly according to a malevolent plan. They can also send conflicting values to other processes and preventing them from reaching an agreement. Byzantine failures in asynchrouns systems are either equivalent to those in synchronous systems (if the malevolent process sends messages) or equivalent to fail-stop failures [2].

A protocol that can tolerate up to t Byzantine failures processes is said to be t-Byzantine resilient and is sometimes called a Byzantine protocol. In the absence of good ways of characterizing the kinds of failures that can occur, protecting against Byzantine failures is a conservative approach to a reliable system design [7].

In a DSM system, all non-faulty processes should be able to reach a common agreement even if certain components in the system are faulty. Therefore, non-faulty processes need to be free from the influence of faulty processes. If faulty processes are dominant in number, they can prevent non-faulty processor from reaching a consensus [18].

2.8.2 Discussion

In a synchronous system we can arrive at consensus if we know the diameter of the network or the number of processes. In this case, the process will eventually terminate (finish the execution), and we can also determine its failure. However, if the network is synchronized, but we have no knowledge of the network we don't know how long it will take for a process to arrive. Another question to consider is how can we halt on a network with an arbitrary and a finite size? If at some point we get no feedback, we don't know if it is a faulted system or not. It could be that there is a nod that is further, and the message didn't arrive yet. Therefore, we need to know something about the network in order to arrive at consensus. If we know the diameter of the network, we can count the network. Assuming that the number of processes is known, then the system is synchronized and there is no process failure. Then, all processes that agree to send messages and to wait for confirmation eventually we will get the answer. If one process fails, then we don't get consensus.

In order to determine what kind of necessary assumptions are needed in order to reach consensus, several algorithms will be summaries in a table. The table will present how consensus can be reached in synchronous and in asynchronous systems with the necessary assumption for each algorithm. In addition, the final results are summaries for each algorithm. A detailed algorithm description will follow the summary table.

Table 1. Consensuses in Asynchronous System

Algorithm Name	Authors	Assumptions	Results
"Impossibility of Distributed Consensus with One Faulty Process"	Fisher et al.	No upper bound on message delay Reliable message system N processes Processes do not have access to synchronized clocks (can't use time out algorithm) Cannot detect a death of a process (can't tell if its dead or just runs very slow) Processes are modeled as automata Atomic broadcast Every message is eventually delivered as long as the destination makes infinitely many attempts to receive	Possibility of non-termination Can't tolerate even a single unannounced process death Stopping of a single process at an inappropriate time can cause failure to reach agreement

Algorithm Name	Authors	Assumptions	Results
"Reaching Approximate Agreement in the Presence of Faults"	Lynch et al.	Lower bound No upper bound on message delay assume Each process input a real value rather than a binary value All processes must eventually decide on a real values within ε of each other. Fixed number of N processes t-maximum number of faulty processes conditions holds: 1.all correct process eventually halt with output values within ε of each other. 2.the value output by each correct process must be in the range of initial values of the correct processes Each process waits for n-t messages at each round All correct processes can halt at different times	Exact consensus is impossible ε can be chosen to be arbitrarily small to get as close to consensus as desired convergence is guaranteed when n>5t

Algorithm Name	Authors	Assumptions	Results
"Unreliable Failure Detectors for Reliable Distributed Systems"	Chandra and Toueg	Arbitrary time bound No bound on message latency Abstract properties N processes Every process p is equipped with a failure detector Four classes of failure detectors: P, S <>P, and <>S. Where, P $\geq$ S and <>P $\geq$ <>S. Have two properties: completeness and accuracy. Completeness: failure detector eventually suspect every process that crashes Accuracy: restricts the mistakes that a failure detector can make Solving consensus for failure detectors of class S: Satisfies strong completeness and week accuracy, at least one correct process is never suspected Solving consensus using failure detectors of class <>S: Satisfy strong completeness and eventual week accuracy, there is a time after which some correct process	If there are no restrictions on upper bound and it takes infinite time to wait for reply, therefore the results are un-computable.

Algorithm Name	Authors	Assumptions	Results
		is never suspected. f < $\lceil n/2 \rceil$ Faulty processes are are less than half at least $\lceil (n+1)/2 \rceil$ processes are correct coordinator has a priori knowledge during round r: c=(r mod n)+1 Solving consensus of class <>W: It satisfied the weak completeness and the eventually week accuracy (which means that eventually some conditions must hold for a sufficiently long period of time until termination) Weak Completeness- there is a time after which every process that crashes is permanently suspected by some correct process	

Table 2. Consensuses in Synchronous System

Assumptions	Authors	Assumptions	Results
"Reaching Approximate Agreement in the Presence of Faults"	Lynch et al.	Byzantine failures Each process input a real value All processes must decide on a real values within ε. of each other. Fixed number of N processes t-maximum number of faulty processes conditions holds: 1. all correct process eventually halt with output values within ε of each other 2.. the value output by each correct process must be in the range of initial values of the correct processes Default value is chosen for a faulty process All correct processes can halt at different times	ε can be chosen to be arbitrarily small to get as close to consensus as desired convergence is guaranteed when $n>3t$

2.8.3 Consensus in Asynchronous System

In a fully asynchronous model, no assumptions are made about relative speed of the processes or the delay time in delivering messages. Therefore, there is no way to tell whether the sender has failed or is just running very slowly [7]. In asynchronous system, computations do not proceed in lock steps; a process can send and receive messages and perform computation at any time [18].

2.8.3.1 Impossibility of Reaching Consensus. Fischer et al., assumes unbounded message delay in delivering a message, unbounded processes' speeds, and processes do not have access to synchronized clocks. Since asynchronous system has no upper bound restrictions on message delay, during execution a delay of a process can cause the entire algorithm to wait indefinitely and there is a possibility of non-termination. Every process that waits for a response might wait forever and we don't know if it's ever going to answer; the stopping of a single process at an inappropriate time makes it impossible for a process to tell whether another process has died (stopped entirely) or just running very slowly.

In the asynchronous system, every message might eventually delivered as long as the destination process makes infinitely many attempts to receive. Also, messages can be delayed, can be arbitrarily long, and can be delivered out of order. Base on these assumptions, in asynchronous distributed system there is no protocol that can guarantee consensus even if a single process can fail by stopping [8].

2.8.3.2 Reaching Approximate Agreement. Lynch et al. says that the article that introduces the approximate agreement algorithm contradicts the result of Fischer et al. (which say that consensus in the present of a faulty process cannot be achieved). However, this article actually supports Fisher's theory. Not only that Lynch et al. develop a consensus algorithm with no upper bound on time delay, they assuming a lower bound only on a convergence rate to reach consensus. This means that the time of a message delay can be arbitrarily long. In order to reach consensus in asynchronous system, we need to add some restriction on time bound. If we add an upper bound on message delay we get synchronous system.

The goal of Lynch et al. is to reach an approximate agreement rather than an exact agreement. This algorithm works by successive approximation, with provable convergence rate that depends on the ratio between the faulty processes and the total number of processes.

The algorithm worked with the assumption is that each process inputs a real value rather than a binary value. Also, assuming a fixed, pre-assigned $\varepsilon > 0$ (as small as desired). The approximate agreement algorithm must satisfy the agreement and Validity condition. The agreement condition satisfies that all correct processes eventually halt with output values that are within ε of each other. For example, if two processes in T (subset with non-faulty processes) enter halting states with values r and s, respectively, then $|r - s| \leq \varepsilon$. The validity condition satisfies that the value output by each non-faulty process must be in the range of initial values of the non-faulty processes. For example, if a process in T enters a halting state with value r, then there exists processes in T having x and y as initial values, such that $x \leq r \leq y$.

The algorithm works by successive approximation. At each round, until termination is reached, each process sends its latest value to all processes (including itself). Each process only waits for n - t messages (where t are faulty processes and n are the total number of processes). For example, at round h, each non-faulty process p performs the following steps: process p labels its current value with the current round number h, and then broadcasts this labeled values to all processes, including itself. Process p waits to receive exactly n - t round h values and collects these values into a multiset V. Since there can be at most t faulty processes, process p will eventually receive at least n - t round h values.

On receipt of a collection V of values, the process computes a certain function f(V) as its next value. The function f is a kind of averaging function. In this way, every round gets closer to the goal with a guaranteed convergence rate. In the asynchronous system, convergence is guaranteed when n > 5t [5].

Would like to note that the approximate agreement algorithm works like the fuzzy barrier synchronized algorithm. The fuzzy barrier doesn't have an exact synchronization point. The barrier region has a range of statements in which synchronization can take place. Where in the approximate agreement, we don't have an exact consensus value; the output value halts within a range of ε.

2.8.3.3 Unreliable Failure Detector. This article also says that it contradicts the theory of Fisher et al. (exact agreement with guaranteed termination is not possible in an asynchrony system even with only one faulty process). On the contrary, it supports Fisher's theory. This article presents a mechanism of failure detectors that maintain a list of faulty processes. However, the failure detectors can make mistakes by entering a correct process to the list. This consensus algorithm allows up to infinite number of mistakes. This applies that if there can be infinite number of failure there is also an infinite time to wait for a reply. Therefore, there is an applied assumption that there is an arbitrary time bound; there are no restrictions on upper bound for message delay and the results are un-computable.

The algorithm of Chandra and Toueg unreliable failure detectors, works as follows: unreliable failure detectors can be characterize in terms of two properties: completeness and accuracy. In general, completeness requires that a failure detector eventually suspect every process that actually crashes, while accuracy restricts the mistakes that a failure detector can make.

Certain failure detectors allows any number of process failures, while other failure detectors require a majority of correct processes. This depends on the hierarchy that the failure detectors form. The failure detectors allow different number of mistakes; a correct process can erroneously be added to the list of processes that are suspected to have crashed. Base on mistakes and repentance (the mistaken process is removed form the list of suspected processes) a hierarchy of failure detector is being chosen.

Four classes of failure detectors P, S, <>P, and <> S satisfies the strong completeness are needed to be presented. The strong completeness property satisfies the condition that eventually, every faulty process is suspected by every correct process. Where, $P \geq S$ and $<> P \geq <>S$. Therefore, algorithm that will solve consensus for S will solve for P, and algorithm that will solve consensus for <> S will solve for <> P. The consensus algorithm that uses S tolerates any number of failures. In contrast, the one that uses <>S requires a majority of correct processes, the use of them depend on the number of mistakes that the failure detectors can make.

One of the central assumptions about the asynchronous system is that a death of a process cannot be detected and therefore we cannot distinguish a dead process from a slow one. The failure detectors keep a list of processes, which it thinks has crashed, and could regularly inspect each process to update its list. Since failure detector can make an infinite number of mistakes, each local failure detector module can repeatedly add and then remove correct processes form its list of suspect. all processes may be erroneously added to the lists of suspects at one time or another. With maximum number of mistakes that a failure detector it can solve consensus using class <>S.

Class <>S is a class of failure detectors that satisfy only strong completeness and eventual weak accuracy property such that: (strong completeness) eventually, every faulty process is suspected by every correct process and (eventual weak accuracy) there is a time after which some correct process is never suspected by any correct process [14].

To solve consensus using <>S, the assumption is that the maximum number of faulty processes (f) is less than half, $f < \lceil n / 2 \rceil$. That at least $\lceil (n + 1) / 2 \rceil$ processes are correct. Also, assuming that all processes have a priori knowledge of the list of (potential) coordinators; during round r, the coordinator is process c = (r mod n) +1. All messages are either to or from the "current" coordinator. Every time a process becomes a coordinator, it tries to determine a consistent decision value. If the current coordinator is correct and is not suspected by any surviving process, then it will succeed, and it will broadcast this decision value. At each round every process sends its current estimate of the decision value times-stamped with the round number in which it adopted his estimate, to the current coordinator (c). The coordinator gathers $\lceil (n + 1) / 2 \rceil$ such estimates, selects the one with the largest timestamp, and sends it to all the processes as their new estimate. If c is a correct process, every process sends an acknowledgement (ack) to c to indicate that it adopted the new estimate. If the processes suspects that the c crashed, they send nack, not acknowledgement to c. The coordinator c waits for $\lceil (n+1)/2 \rceil$ replies (acks or nacks). If all replies are acks, then c knows that a majority of processes changed their estimates to the new one and c broadcast the new decision value.

Consensus using failure detector of class S, satisfies strong completeness and weak accuracy, at least one correct process is never suspected. This algorithm tolerates up to n - 1 faulty process (in asynchronous systems with n process), any number of process failures.

The algorithm runs through three phases. In phase one, processes execute n - 1 asynchronous rounds during which they broadcast and send their proposed values. Each process p waits until it receives a round r message from every process that is not in D_p (failure detector list) before proceeding to round r + 1 (if message q is added to D_p, while p is waiting for a message from q, p stops waiting for q's message and proceeds to round r + 1). In phase two, correct processes agree on a vector V_p, based on the proposed values of all processes. The i_{th} element of this vector either contains the proposed value of process p_i or null. This vector contains the proposed value of at least one process. In phase three, every correct process decides on the first non-null value and this satisfies termination of consensus. No process decides more than once.

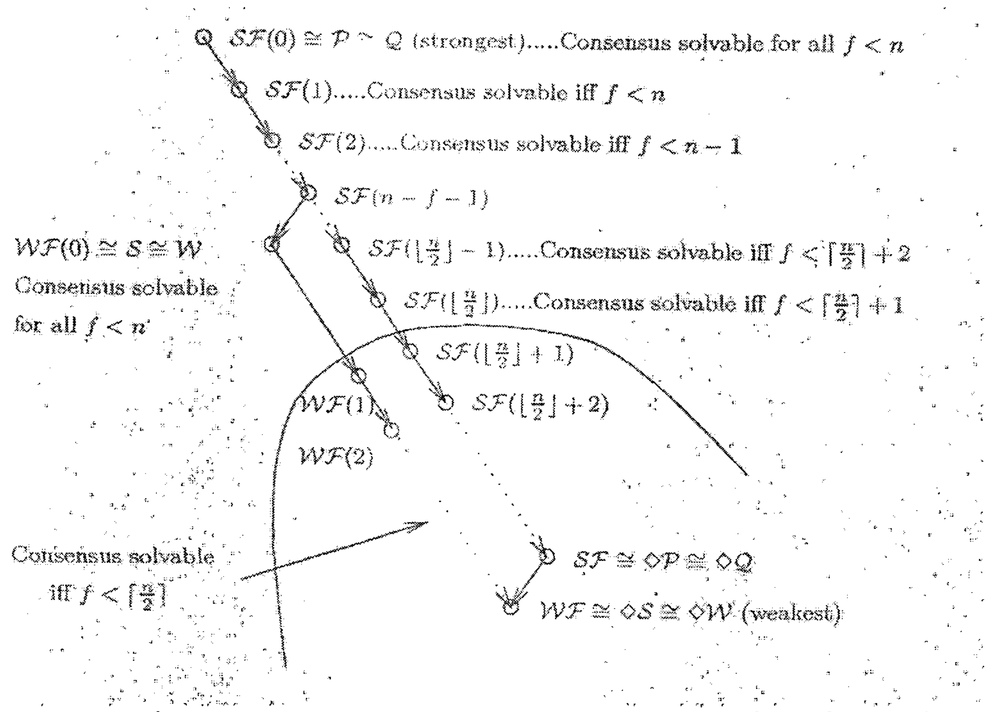
According to Chandra and Toueg consensus is also solvable using class <> W, the weakest class of failure detectors, in asynchronous systems with $f < \lceil n/2 \rceil$ (maximum number of faulty processes is less than half).

Failure detectors could make mistakes and can be used to solve consensus despite such mistakes. A mistake occurs when a correct process is erroneously added to the list of processes that are suspected to have crashed. If a process learns that its failure detector module made a mistake, it takes a corrective action, repentance. For example, suppose failure detector module at process p erroneously adds q to D_p at time t. Then, p sends a message to q and receives a reply. This means that q had not crashed at time t and p knows that the module made a mistake about q and the failure detector module at p takes the corrective action of removing q from D_p. The property for repentance is defined as follows: if a correct process p eventually knows that q had not crashed at time t, then at some time aft t, q is removed.

The hierarchy of repentant failure detectors differ by the maximum number of mistakes they can make and defined as flows:

1. SF(k)-the class of strongly k-mistaken failure detectors (D makes at most k mistakes).
2. SF: the class of strongly finitely mistaken failure detectors (D makes a finite number of mistakes).
3. WF(k): the class of weakly k-mistaken failure detectors (there is a correct process p such that D makes at most k mistakes about p).
4. WF: the class of weakly finitely mistaken failure detectors (there is a correct process p such that D makes a finite number of mistakes about p)

The hierarchy above is summaries as follows: SF(0) ≥ SF(1) ≥...SF(k) ≥ SF(k+1) ≥...≥ SF. A similar order holds for the WFs where SF≥ WF [3]. This description follows the illustration in figure 15:


$\mathcal{SF}(0) \cong \mathcal{P} \sim \mathcal{Q}$ (strongest).....Consensus solvable for all $f < n$
$\mathcal{SF}(1)$.....Consensus solvable iff $f < n$
$\mathcal{SF}(2)$.....Consensus solvable iff $f < n - 1$
$\mathcal{SF}(n - f - 1)$
$\mathcal{SF}(\lfloor \frac{n}{2} \rfloor - 1)$.....Consensus solvable iff $f < \lceil \frac{n}{2} \rceil + 2$
$\mathcal{SF}(\lfloor \frac{n}{2} \rfloor)$.....Consensus solvable iff $f < \lceil \frac{n}{2} \rceil + 1$
$\mathcal{SF}(\lfloor \frac{n}{2} \rfloor + 1)$
$\mathcal{SF}(\lfloor \frac{n}{2} \rfloor + 2)$
$\mathcal{WF}(0) \cong \mathcal{S} \cong \mathcal{W}$
Consensus solvable for all $f < n$
$\mathcal{WF}(1)$
$\mathcal{WF}(2)$
Consensus solvable iff $f < \lceil \frac{n}{2} \rceil$
$\mathcal{SF} \cong \diamond\mathcal{P} \cong \diamond\mathcal{Q}$
$\mathcal{WF} \cong \diamond\mathcal{S} \cong \diamond\mathcal{W}$ (weakest)
Figure 15. Hierarchy of Maximum Number of Mistakes that can be Made

2.8.4 Consensuses in Synchronous System

Systems in which there is a finite bounded delay on the operations of the processes and on their intercommunication are said to be synchronous. In such systems, unannounced process deaths, as well as long delays, are considered to be faults [5].

In synchronous computation, processes in the system run in lock step manner, where in each step, a process receives messages, performs computation, and sends messages to other process. A step of synchronous computation is also referred to as a round. In synchronous computation, a process knows all the messages it expects to receive in a round.[18].

2.8.4.1 Approximate Agreement. According to Lynch et al., the algorithm of the approximate agreement is about reaching an agreement rather then exact agreement. This algorithm works with the assumption that processes are allowed to terminate at different times, meaning that there is no upper bound on message delay. But, synchronous system must restrict an upper bound and should guaranty termination in a bounded time, which Lych et al. fail to reach. If we don't have an upper bound on message delay the system can be looked at as asynchronous system. Since this algorithm doesn't guarantee termination in a bounded time, the proof is incomplete and the algorithm is un-computable. Although they don't prove the approximate algorithm, it seems that it could be proven correctly with an upper bound limit.

The algorithm works as follows: each process inputs a real value rather than a binary value. Also, assuming a fixed, pre-assigned $\varepsilon > 0$(as small as desired). The approximate agreement algorithm must satisfy the following two conditions: all correct processes eventually halt with output values that are within ε of each other, and the value output by each non-faulty process must be in the range of initial values of the non-faulty processes.

The algorithm works by successive approximation. At each round, until termination is reached, each process sends its latest value to all processes (including itself). This algorithm is the same as in the asynchronous system except that if a faulty process does not send a value, then some default value, say 0, is chosen. For example, at round h, each non-faulty process p performs the following steps: process p broadcasts its current value to all processes, including itself. Then, process p collects all the values sent to it at that round into a multiset V. If p does not receive exactly one correct value from some particular other process (meaning the other process is faulty), then p simply picks some arbitrary default value to represent that process in the multiset. The multiset V, therefore, always contains exactly n values.

On receipt of a collection V of values, the process computes a certain function f(V) as its next value. The function f is a kind of averaging function. In this way, every round gets closer to the goal with a guaranteed convergence rate. In the synchronous system, convergence is guaranteed when $n > 3t$. The function f is chosen to eliminate the lowest and highest t values from the list and take the average of the rest. The faulty processes are unable to affect the convergence of the values [5]. Because of the algorithm assumption, which contradicts the assumption of synchronous system, this algorithm is un-computable.

2.8.5 Conclusion

Consensus allows processes to reach a common decision based on the votes of each process. In order to reach consensus, we need to make assumptions about the system; we need to know whether the system is synchronous or not. If there are restrictions on upper bound on massage delay and if we know the number of participating processes in the system, the system is synchronous.

A distributed system is asynchronous if there is no upper bound on message delay or on the time necessary to execute a step. The impossibility result to reach consensus, by Fisher et al., for example has difficulty of determining whether a process has actually crashed or is only running "very slow." Asynchronous system depends oh how we define time. If we choose an arbitrary long time as an upper bound, we have the possibility of a process never terminate. A process can wait indefinitely, but eventually in a million years will terminate which is just as good as non-termination. Therefore, asynchronous system is un-computable.

To arrive at consensus and get a response from another processes we need a restricted upper bound on time. If we restrict a time bound on an asynchronous system we get synchronous system. Then, we can successfully exchange messages and receive acknowledgments and consensus can be reached. Once there is a successful message exchange, synchronization occurs. Therefore, if we can't reach consensus we can't synchronize.

2.9 Summary

Chapter Two consists of a discussion of the relevant literature to the taxonomy of synchronization in DSM. In a DSM system there is no global clock or a set of perfectly synchronized clocks. According to Lamport's algorithm, the use of logical clocks can determine the exact time relation in which two events occur. A further definition for synchronization in discussions with Dr. E. Gomez is introduced. Where there can only be two kinds of time relation between events: deterministic or non deterministic. In a deterministic time relation between events, a process with a lower timestamp can be released before a process with a later timestamp, or processes can be released all at once in the same timestamp. On the other hand, in the non-deterministic approach, processes are not released in an orderly manner.

Mutual exclusion in a distributed environment can be implemented in a variety of ways. In a centralized approach, one of the processes in the system is chosen to coordinate the entry to the CS. In the distributed algorithm approach, the decision-making is distributed across the entire system and is based on the event-ordering scheme. A token ring algorithm is a distributed algorithm, which is applicable to ring-structured network with the approach of token passing. Many distributed algorithms require a coordinator. Therefore, two ways of electing a coordinator were presented, the bully algorithm and the ring algorithm. Where, in the bully algorithm, the process with the higher number holds the election. In the ring algorithm, at each step along the ring, the sender adds its own process number to a list while sending a message to its successor. The coordinator is the list member with highest number.

A semaphore is a non-deterministic synchronization tool that can be used to solve mutual exclusion problem. The relation between times of events in a semaphore is the same as the one in the non-deterministic synchronization. In a counter semaphore, S is an integer variable that is accessed only through two standard atomic operations: wait and signal. The wait and signal operations are executed indivisibly. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait and block itself into a waiting queue. A blocked process should be restarted when some other process executes a signal operation, which changes the waiting state to the ready state. The process is then placed in the ready queue. A binary semaphore is a semaphore with an integer value that can range only between 0 and 1, where the initial value is 1. Binary semaphores are used to create mutual exclusion, because at any given time only one process can get past the wait operation.

Counting semaphores are used to synchronize access to a shared resource by several concurrent processes, which allow control on how many processes can concurrently perform an operation. This is relevant for my thesis because I need so see if I can make a barrier out of semaphore, and with the use of the list in the counter semaphore I can create a barrier. I will also need to see what can be done to make a semaphore deterministic in order to investigate if semaphore can become a basic synchronization mechanism.

Barriers are another synchronization mechanism, which are intended for groups of processes. They have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. Once the processes are release, they can be released all at once or in an orderly manner, which become the same definition as the deterministic synchronization. This information is relevant in order to make a further investigation if a barrier is indeed the basic synchronization mechanism unit, or if we can make any synchronization from any synchronization.

This chapter has explored many designs aspects of barrier synchronization algorithms through software. In the centralized barrier, all processes use the same lock to increment the same counter when they signaled their arrival, and all waited on the same flag variable until they were released. On a large machine, the allowing for all processes to access the same lock and to read and write the same variable can lead to a lot of traffic and contention and the use of a tree structure reduces hot spots. Also, a tree structure is a distributed way to coordinate the arrival or release of processes. It is another message-based mechanism that is used to synchronize between pairs of processes.

Some barrier algorithms were presented as solutions to reduce latency of the waiting processes at the barrier. For example, the software-combining tree reduces memory contention by replacing the single lock and counter of the centralized barrier by a tree of counters. In the adaptive software-combining tree, the processes that arrive early at the barrier adapt the combining tree so that it has a structure appropriate for reducing the latency for the processes that arrive later. In addition, the fuzzy barrier mechanism reduces the idling of processes at the barrier by allowing the processes to execute useful instructions while they are waiting at the barrier.

In the consensus problem, a group of processes has to arrive at a common decision; a set of processes must all agree on a decision based on their initial states. Consensus can be easy or difficult to achieve depending on the kind of the system (synchronous or asynchronous) and the algorithm assumptions. The assumptions have a large impact on what can be achieved in practice. If an upper bound on time is defined in a system, the system is synchronous and consensus can be reached. If the upper bound on message delay is arbitrary long, and only lower bound is assumed, then the system is asynchronous and consensus can't be reached.

Techniques such as "reaching approximate agreement in the presence of faults" by Lynch at el. and "Unreliable failure detectors for reliable distributed system" by Chandra and Toueg suggested that it is possible to reach consensus in asynchronous system. However, they do not assume an upper bound on message day, only an arbitrary upper bound which actually supporting Fisher et al.'s theory, that consensuses can't be achieve in asynchronous system. Since an upper bound is not declared, we cannot tell if a process has died or if it is just very slow in sending its message. Moreover, in reaching consensus in synchronous system, Lych at el., in the approximate algorithm, arrive at incomplete proof because there is no restriction on upper bound. However, it seems like it could be proven with an upper bound and therefore could guarantee fault tolerant. This algorithm also opens the door to a new area in synchronization. It could apply that different relation pairs of processes might interact with each other during one synchronization; this could be defined as partial synchronization and might reach approximate consensus. Also, this algorithm resembles the Fuzzy barrier because it doesn't have an exact synchronization point. Instead, the Fuzzy barrier has an approximate synchronization point.

Since a barrier is a form of synchronization and a semaphore is a form of synchronization, a further investigation is needed in order to find out if a semaphore can be considered a barrier. In other word, a further investigation is needed to find out if a barrier and a semaphore are reducible to each other. This will be discussed in Chapter Three, Methodology.

CHAPTER THREE

METHODOLOGY

3.1 Introduction

Chapter Three documents the steps used in the Methodology of the thesis. Specifically, showing that a barrier can be used as a semaphore and a semaphore can be used as a barrier.

3.2 Obstacle

First, the standard semaphore is non-deterministic. Which means, the order in which processes are allowed to enter to CS does not matter. But, in a DSM system, the order of events is important since we can't receive a message before it is sent. How can we make the semaphore deterministic?

Second, for instance, let assume that three processes p1, p2, and p3 want to enter into CS and a barrier replaces semaphore. By doing so, we receive a chaotic result. This is because the processes that are waiting for each other at the barrier are released all at once into the CS; this operation violates mutual exclusion. Therefore, there must be some mechanism to coordinate processes during the release stage, before entering CS.

My new algorithms are a solution for this problem. They combine some features of barrier and some of the counter semaphore and it works as follows. If we place a barrier instead of a semaphore, before the CS, and modify the way processes are released from the barrier list, the same concept as the list semaphore counter, mutual exclusion is not been violated. What actually happens is that processes are waiting for each other at the barrier point, and are released sequentially (one after another). Processes can be released one after another according to FIFO, priority, or size, etc.

3.3 My Algorithm

My two new algorithms, Algorithm1 and Algorithm2 are presented. In Algorithm1, all processes are released all at once at the barrier point. In Algorithm2, processes are released sequentially, one after another, using a signal. In Algorithm1 I am making a barrier from a semaphore. In Algorithm2 I am making a semaphore from a barrier. The pseudocode for these algorithms is located in page 95 and in page 97 of this thesis.

3.3.1 Making a Barrier from a Semaphore

In Algorithm1, figure 16, each process that enters the barrier increments the counter (which counts the number of processes arrive at the barrier point), enters its process id into a local (barrier) list, and block itself. This step is repeated until the Nth process (last process) arrives at the barrier. Once the last process (N) arrives, it clears the counter for the next use and wake up the processes that are blocked, waiting in the barrier. It does so by calling the wake_up_processes function.

The wake_up_processes function, in Algorithm1, works as follows: all processes are released at once within a "for loop" on the barrier list in an orderly manner. The processes can be removed in an order of FIFO, by their Priority, by Size, etc.

Assuming that the list can be a fixed or linked list; when a process needs to register in a list (before it fall a sleep) it adds the process id and its priority, to the list. In this algorithm, a barrier can be mad out of semaphore and the behavior of the system would be different. We get a list of processes that are being released all at once within a "for loop". For example:

```
 p1                  P2

barrier()           barrier()/* the semaphore are inside

                                the barrier/*
```

This description follows the pseudocode in figure 16:

```
/*****************************************************************************************************************
                    Algorithm 1:  Making a Barrier from Semaphore
*****************************************************************************************************************/
barrier()
{
  if(counter < N)  /* processes are accumulated in a list forming a barrier */
  {
    wait(S)
    counter++;  /* counts the number of processes that arrive at the barrier list.  It's protected
                   by wait() and signal(). */
    Signal(S)
    barrier_list <- pid;  /* register process into the barrier list */
    go_to_sleep();  /*  processes are in a block state (one of the O.S. services), waiting at the
                        barrier list until they are released. */
  }
  else
  {
    counter = 0;  /* clear counter for the next barrier. */
    wake_up_processes(mode);  /* last process that enters the barrier wakes up the
                                 processes from the barrier list and they are released. */
  }
}

/*****************************************************************************************************************
This function is called by the last process that enters into the barrier.  It removes the process
from the list and wakes them up.
*****************************************************************************************************************/
wake_up_processes()
{
  for(i = 0; i < (N-1); i++)
  {
  pid = get_&_remove_next_process_from_list();  /* in an order of FIFO, priority, etc. remove
                                                   the process from the list */
  wake_up(pid);  /* processes from the barrier list go to the ready Q. */
  }
}
```

Figure 16. Making a Barrier from a Semaphore

3.3.2 Making a Semaphore from a Barrier

In Algorithm2, the barrier() function replaces the wait(S) function that is in the counter semaphore. Each process registers itself into the barrier list, and all the processes are in a block state waiting for each other at the barrier point. The last process to arrive enters CS. When it exits the CS, it signals and wake up a process from the barrier list (according to an order of FIFO, Priority, etc.).

Each process is released from the barrier list after a signal or a message. So, when the first released process exits the CS, it sends a signal to release the next process from the barrier; then, it can enter the CS. This operation is repeated as long as there are processes in the list. The behavior of the system would be as the following example: assume three processes, P1, P2 and P3.

```
p1:                 p2:                 p3:
barrier()           barrier()           barrier()
critical s.         critical s.         critical s.
signal_b()          signal_b()          signal_b()
```

This description follows the pseudocode in figure 17:

```
/*****************************************************************************************************************
                        Algorithm 2:  Making a Semaphore from a Barrier
*****************************************************************************************************************/
barrier()
{
  if(counter < N)  /* processes are accumulated in a list forming a barrier */
  {
    wait(S);
    counter++;  /* counts the number of processes that arrive at the barrier list.  It's protected
                    by wait() and signal() */
    Signal(S);
    barrier_list <- pid;  /* register process into the barrier list */
    go_to_sleep();  /*  processes are in a block state (one of the O.S. services), waiting at the
                        barrier list until they are released */
  }
  else
  {
    counter = 0;  /* clear counter for the next barrier */
  }  /* the Nth process will go into CS */
}

/* the process that exits CS, will send a signal to a process in the barrier list and it will be
  released according in an order of FIFO, priority, etc. */
wake_up()  /* signal */
if(barrier_list != EMPTY)
  pid = get_&_remove_next_process_from_list;  /* (pid <- list; get and remove from the list */
  wake_up(pid);
}
```
Figure 17. Making a Semaphore from a Barrier

3.4 Summary

The solution in reducing semaphore and barrier to each other is by forcing all the processes to accumulate in a list. This is done by using a counter; then each process registers and blocks itself in the list. This collection of waiting processes in a list forms a barrier. All the processes at the barrier point are waiting for the last process to arrive and wake them up; then the processes can be released from the barrier.

By manipulating the way the processes are released they can be waken up all at once, using a "for loop", or by using a signal. After the processes are released, they can enter CS without violating mutual exclusion. Also, the processes are being released in an orderly manner and the semaphore becomes deterministic (just like a barrier). Therefore, we can make a barrier from a semaphore and a semaphore from a barrier.

CHAPTER FOUR

DISCUSSION

4.1 Introduction

Included in Chapter Four is a presentation of the findings of this thesis. It was found that if a barrier is a form of synchronization, the reverse could also be true (any form of synchronization can be a barrier). Initially we can discuss that the reason we can reduce semaphore to a barrier is because the barrier is the basic mechanism for synchronizing processes. Therefore, from a barrier we can create variations of synchronizations and no other form of synchronization can be used as a basic building mechanism. It could be argued that there could be new synchronization methods to be used as the basic form of synchronization (other then a barrier). Then, might we not need to start with a barrier? The answer lies in the basic definition of synchronization in which there are two possible relations between two or more processes. They can be deterministic or non-deterministic. Obviously, we cannot rely on a non-deterministic relation since the output will be unpredictable. Therefore, we cannot build a new non-deterministic synchronization mechanism. So, given a basic synchronization unit, we need to look at a deterministic relation only. This means that the processes will be executed at the same time and in the same order. If processes are executed at the same time, it leads us to the basic definition of what a barrier is; that is, all processes have to meet in a global point, and execute at the same time. Therefore, a barrier is the basic unit of synchronization and as its been shown, we can build any form of synchronization from a barrier. However, according to the discussion in section 4.2, the amount of work it takes for a barrier and a semaphore to release its processes in an orderly manner is O(n). Therefore, it can be argued that if we can build a semaphore from a barrier and a barrier from a semaphore (since they require the same amount of work); then, they are both basic mechanism units of synchronization. Which means that we can build any synchronization mechanism from any synchronization mechanism. In other words, any synchronization can be a basic mechanism for synchronizing processes. This discussion is relevant only under the assumptions that processes are released in a deterministic order in a shared memory system and the number of processes is known ahead of time.

4.2 Amount of Work-Order Required

It was found that a barrier and a semaphore could be reducible to each other where we can express a barrier as a semaphore and a semaphore with a barrier. The amount of work it takes for a semaphore and a barrier to release processes in the same order is being compared.

In a Algorithm1, building barrier synchronization from semaphore synchronization, the processes are released in a "for loop": for (I = 0; I < (N - 1); I++), which takes O(n) work. Therefore, to form a semaphore from a barrier and a barrier from a semaphore takes equal amount of work to deterministically release the processes in order of O(n)steps.

In Algorithm2, when building semaphore synchronization from barrier synchronization, processes are released one at a time. Eventually, all processes are removed form the list sequentially. Therefore, the amount of work it takes to build a semaphore from barrier is of order of O(n).

4.3 Summary

In a Distributed Shared Memory (DSM) environment, processes need to be synchronized and communicate with each other in an orderly meaner. Using ordering of time relation between two or more processes is a way of achieving synchronization. Also, when a system has an upper bound on time, processes can successfully exchange messages and receive acknowledgments. If no process needs to wait indefinitely for a reply, consensus can be reached and synchronization occurs. Therefore, consensus and synchronization are related because if we can arrive at consensus we can synchronize, and if we can agree on synchronization we can arrive at consensus. In addition, barrier synchronization requires consensus and ordering. To synchronize a barrier, we need to know in advanced the number of processes to arrive at the barrier and the order in which they arrive. Once all the processes arrive at the barrier point, they all released in order, and deterministic synchronization occurs. Because barrier is deterministic and by definition it is the same as the basic definition of synchronization, I investigated if indeed the barrier is the basic mechanism unit of all synchronizations or we can build any synchronization from any synchronization. For the proof I compared barrier to semaphore. To prove that semaphore in a DSM can be a barrier I had to find a way to show that semaphore can become deterministic; therefore, some ordering needs to be provided. For the purpose of the proof, I used semaphore counter as a way to know the number of processes that arrive at the barrier_list ahead of time. Then, each process added its process ID and priority into the list. Also, the release of processes from the barrier_list was done in an orderly manner, in the order of O(n). From these results we can see that barrier and semaphore require the same amount of work to deterministically release processes. Also, we can say that a semaphore is deterministic and it arrives at consensus. I have shown that we can make barrier from semaphore and semaphore from barrier. Furthermore, it can be concluded that we can build any synchronization mechanism from any synchronization mechanism.

REFERENCES

1. Aiken, A., and David, G., "Barrier Inference", EECS Department, University of CA, Berkeley, 1998

2. Attiya, H., and Welch, J., Distributed Computing, McGraw-Hill Publishing Comapany, 1998, pp.91-123

3. Chandra, T.D., and Toueg, S., "Unreliable Failure Detectors for Reliable Distributed Systems", ACM, Vol.43, No. 1, March 1996, pp.225-267

4. David, E.C., and Singh, J.P., Parallel Computer Architecture- A Hardware/Software Approach, Morgan Kaufmann Publishers, Inc. San Francisco, CA., 1999, pg.57, 334-337, 353, 542-547, 704-705

5. Dolev, D., and Lynch, N.A., "Reaching Approximate Agreement In The Presence of Faults," Journal of the Association of Computing Machinery, Vol.33, No.3, July 1986, pp.499-516

6. Eichenberger, A., and Abraham, G., "Impact of Load Imbalance on the Design of Software barriers", Proceedings of the 1995 International Conference on Parallel Processing, August 1995, pp. 62-72

7. Fischer, M.J., "The Consensus Problem in Unreliable Distributed Systems", YaleU/DCS/TR-273, February 2000, pp.1-16

8. Fischer, M.J., et al, "Impossiblity of Distributed Consensus with One Faulty Process", Journal of the Association of Computing Machinery, Vol.32, No.2, April 1985, pp. 374-382

9. Galvin, S., Operating system concepts, Forth Edition, Addition-Wesley publishing company, 1995

10. Gupta, R., "The Fuzzy barrier: A Mechanism for the High Speed Synchronization of Processors," 3rd International Conference on Architectural Support for Programming languages and Operating Systems, 1989, pp. 54-63

11. Gupta, R., and. Hil, C.R., "A Scalable Implementation of Barrier Synchronization Using an Adaptive combining tree", International Journal of Parallel Programming, Vol.18, No3, June 1989, pp. 161-180

12. Hennessy, J.L., and Patterson, D.A., Computer Architecture, Morgan Kaufmann Publisher, Inc. San Francisco, CA. Second Edition 1996, pp. 108-113, 694-706

13. Hill, J. M., "Practical Barrier Synchronization", The computing Laboratory Oxford Universtiy, Oxford, UK, Oxf 3QD, 1997

14. Keidar, T., and Rajsbaum, S., "On the Cost of Fault-Tolerant Consensus When There Are No Faults-A Tutorial", Distributed Computing Column, June 200, pages 45-63

15. Lynch, N. and Dwork, C., "Consensus in the Presence of Partial Synchrony", Journal of the Association for Computing Machinery, Vol. 35, No. 2, April 1988, pp. 288-323

16. Raynal, M, et al, "Consensus-Based Fault-Tolerant Total Order Multicast", IEE, Vol. 12, No.2, February 2001, pp. 147-156

17. Sangman M, et al., "Four-Ary Tree-Based Barrier Synchronization for 2D Meshes Without Nonmemeber Involvement" ,IEE Transactions on Computers, vol.50, no.8, August 2001, pp.811-823

18. Singhal, M., and Shivarati, N.G., Advanced Concepts in Operating Systems: distributed Database, McGraw-Hill, Inc., 1994, pp.13-45

19. Sivaram, R, et al, "A Reliable Hardware Barrier Synchronization Scheme", Depart. of Computer and Information Science, The Ohio State University Columbus, Oh., 1998

20. Tanenbaum, A.S., Distributed Operating Systems, Prentic-Hall, Inc, 1995, pp.118-166,134-143

21. Tanenbaum, A.S., Modern Operating Systems, second edition, Upper Saddle River, New Jersey, 2001, pp. 110-113