CONCURRENCY AND SYNCHRONIZATION ISSUES IN SHARED INFORMATION SYSTEMS

by

MADHU C. PATEL

B.S.M.E., University of Illinois, 1965

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

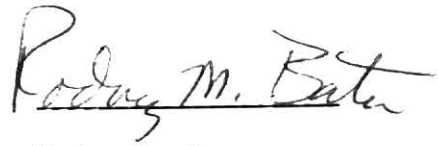
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

Approved by:



Major Professor

TABLE OF CONTENTS

Chapter			Page
I			
	1.0	INTRODUCTION	1
	1.1	STATEMENT OF PROBLEM	3
	1.2	OUTLINE OF THE PAPER	4
II			
	2.0	REVIEW OF LITERATURE	6
	2.1	HISTORICAL BACKGROUND	6
	2.2	EXECUTION-BASED CONTROL MECHANISMS	10
	2.2.1	SEQUENTIAL MACHINE LEVEL CONTROL	12
	2.2.2	OPERATING SYSTEM LEVEL CONTROL	15
	2.2.3	DBMS LEVEL CONTROL	16
	2.2.4	PROGRAMMING LEVEL CONTROL	17
	2.3	PROCESS SYNCHRONIZATION CONCEPT	17
	2.3.1	SYNCHRONIZATION BASED ON SHARED VARIABLES	19
	2.3.2	SYNCHRONIZATION BASED ON MESSAGE PASSING	27
	2.3.3	OTHER METHODS OF SYNCHRONIZATION	30
	2.4	CONCURRENCY CONTROLS IN DBMS	34
III			
	3.0	IMPLEMENTATION REVIEW IN COMMERCIAL SYSTEMS	42
	3.1	NCR 6600 INTELLEGENT I/O CONTROLLER	43
	3.2	INTERPROCESS COMMUNICATION IN UNIX	49
	3.3	SYNCHRONIZATION IN CONCURRENT CP/M-86	52
	3.4	SYNCHRONIZATION IN SDD-1 DDBMS	56
IV			
	4.0	EVALUATION	60
	4.1	COMPARISON	60
	4.2	LIMITATIONS	63
	4.3	IMPROVEMENTS	64
V			
	5.0	SUMMARY AND CONCLUSION	67
		BIBLIOGRAPHY	70

THIS BOOK CONTAINS NUMEROUS PAGES WITH THE ORIGINAL PRINTING BEING SKEWED DIFFERENTLY FROM THE TOP OF THE PAGE TO THE BOTTOM.

THIS IS AS RECEIVED FROM THE CUSTOMER.

LIST OF FIGURES

Figure		Page
1a	No Concurrency In System	7
1b	Concurrent I/Os	7
1c	Concurrency In Multi-Users system	8
1d	Concurrency In A Network	8
2	Control Components And Structures In Computer Systems	11
3	States Of A Process In An Operating System	16
4	NCR 6600 File Subsystem	44
5	Structure Of SDD-1 Global Time Layer	58
6	Major Synchronization Mechanisms In Commercial Products	62

ACKNOWLEDGEMENT

I wish to express my indebtedness to my advisor at KSU, Dr.Beth Unger,for her guidance during the preparation of this paper and she had been very helpful in giving me encouragement throughout the project. I am thankful to John Howell, Dave Moffett, and Darrell Woelk for their comments and reviews. I would also like to thank my wife, Rashmi, for her patience during the project.

My thanks to all of these people, to NCR Corp., and to the Department of Computer Science at Kansas State University for their support during this work.

CHAPTER I

1.0 INTRODUCTION

The concept of concurrency is widely exhibited in the day-to-day operation of industrial nations. A production line in a manufacturing organization is a simple structure in which information and goods flow rigidly along the production line. The goal of the organization is to minimize the cost of the operation. In order to achieve such a goal, the production line needs to keep everyone almost equally busy and to allow an adequate flow of information and resources to keep the line moving. Additionally, it is also a goal to ensure that resources are shared cost-effectively. Thus, a production line is a highly concurrent system that requires synchronization of the shared resources and information. We can find a similar situation in many other systems. Economic systems, legal systems, and traffic control systems are just a few examples of systems which require interacting components (process, information, resource) exhibiting concurrency or parallelism.

Computer systems exhibit characteristics similar to manufacturing systems, and many of the concurrent operation concepts developed in manufacturing systems have been applied to present-day computer systems.

Two processes are defined to be concurrent whenever the first operation of one of the processes is started before the last operation of a previously started process is completed, and the results of the two processes do not interfere with each other.

If the concurrent processes utilizing common resources interact with one another in a non-destructive way to maintain a

consistent state of the system, then the operations are synchronized.

Businesses over the past 40 years have used traditional Electronic Data Processing (EDP) techniques in applications to process data in batch operation at a specified time of day according to the schedule designed by Management Information System (MIS) administrators. All this has been changed rapidly over the last five years in favor of applications doing interactive processing of data. This change has come about as result of availability of low-cost and high-performance hardware such as storage devices, intelligent terminals, and networking peripherals to the end users. The new environment has allowed the users to share peripherals (keyboards, CRT, printers, and disks) and information (programs, data, files,and databases) in a more cost-effective manner and in a more productive way. However, the complexity of the interaction among the various resources introduces the necessity of finding ways to safely share the resources. This implies that a mechanism must be provided to permit the resources to be shared according to certain rules of operation in the computer systems which guarantee that the correct interactions take place.

This paper is intended to examine the mechanisms of concurrency and synchronization used in shared information systems. The paper is organized to present a review of the available literature and to examine commercial systems that have implemented the ideas from the literature.

1.2 STATEMENT OF THE PROBLEM.

The terms concurrency and synchronization refer to the solution of two distinct but related problems:

a) Controlling and completing the joint activity of cooperating sequential processes.

b) Ensuring consistency of a system whenever multiple processes must access shared objects concurrently.

Conventional programming languages and operating systems provide solutions to specification and control problems by means of semaphores, mutual exclusion of critical regions, and monitors. Some of the newer approaches proposed are event-counts, synchronizers, path expressions, and object managers.

The problem of ensuring consistency of a system due to concurrent access to shared objects by multiple processes is more challenging as a result of movement toward database management systems, networking of distributed systems, and "non-stop" redundant resource systems. The solution to the the problem of ensuring consistency of a system is addressed via techniques such as those known as serialized schedules, timestamp ordering, two-phase locking, and majority consensus.

Two areas from which tools for solving the problems of concurrency and synchronization have come are programming languages and innovative computer architectures. The efforts of software engineers and computer scientists in developing these tools over the last two decades have led to a better formal understanding of languages and to a refining of hardware architectures; these improvements will continue

over the next two decades [Brinch 78].

Hardware architecture around the 1955 was developed to meet the need to handle asynchronously operating peripheral devices. The interrupt capabilty in hardware was developed to switch processes so that the slower devices could operate simultaneously. However, the programs developed in machine-level languages were too cumbersome to manage. The result was that the efforts of developers were directed toward creating higher level languages such as FORTRAN and ALGOL60.

However, keeping the level of concurrency down to two or three parallel operations was not sufficient. The new system programs, called operating systems, were developed in the 60's. This resulted in the creation of Multics(1965) and IBM360 type operating systems. These large single-site operating systems fell short of their original goal of ensuring efficient utilization. Thus, during the late sixties the main focus was on the development of higher levels of abstraction in programming languages. The end result was the development of languages like PASCAL, PL/I, CONCURRENT PASCAL, and recently the ADA language blessed by Department of Defense. During the eighties, the evolution of hardware to provide systems with multiple processors and VLSI data-flow architecture continues. We are assured that this will result in development of new concurrency schemes.

1.3 OUTLINE OF THE PAPER

This paper is composed of five chapters.

Chapter one presents an introduction and a statement of the problem to be discussed.

Chapter two discusses the development of hardware/scftware tools

to handle concurrent applications and discusses how they are used to solve the problems which arise in the synchronization of processes that use shared information. The chapter introduces concepts of concurrency and synchronization and traces their use from early computing systems managing I/O peripherals to the development of systems having complex operating systems. The chapter concludes by examining how programming techniques employing levels of abstraction have provided concurrency controls in database and network systems.

Chapter three discusses the implementation of sharing concepts in several commercial systems that employ some of the concurrency ideas discussed in Chapter two. The chapter introduces the general implementation of some early system of the 1950's and 1960's, followed by descriptions of the NCR File-Server known as MODUS™, Concurrent CP/M-86 operating system for microcomputers, UNIX™ operating system, and a distributed database system known as SDD-1.

Chapter four presents a discussion on limitations of concurrency ideas in present commercial systems. Future system improvements are examined by exploring dataflow architecture and multiple processors systems.

Chapter five presents a summary and suggests some areas such as deadlocks and data overhead management for further study.

CHAPTER II

2.0 Review of Literature

2.1 Historical Background

One extremely important development that has emerged during the post Von Neumann era is the expansion of the concepts of hardware parallelism and concurrency of operations. The concept of concurrency as it has developed since the 50's can be viewed as:

a) Concurrent operation of I/O peripherals.

b) Concurrent processing of multiple programs.

c) Concurrent access by multiple users.

d) Concurrent operations of multiple computers.

Although there are subtle differences among the above concepts as to how the mechanism of synchronization is implemented, the basic problem is always one sharing the resources in the most effective and non-destructive way.

Figures 1a through 1d show a simplistic historical view of hardware and software architecture evolution based on the above four concepts during the last thirty years.

Figure 1a shows a typical system of the early 1950's which has Input/Output devices operating under the control of a fixed set of instructions. The operation of the system is sequential in nature and sharing is not done. At no time is the processor freed to do other work while the I/O devices are active.

Figure 1b shows a typical system of the early 1960's which uses the concept of stored program and has the implementation of other hardware innovations such as memory buffers, registers, interrupts, and

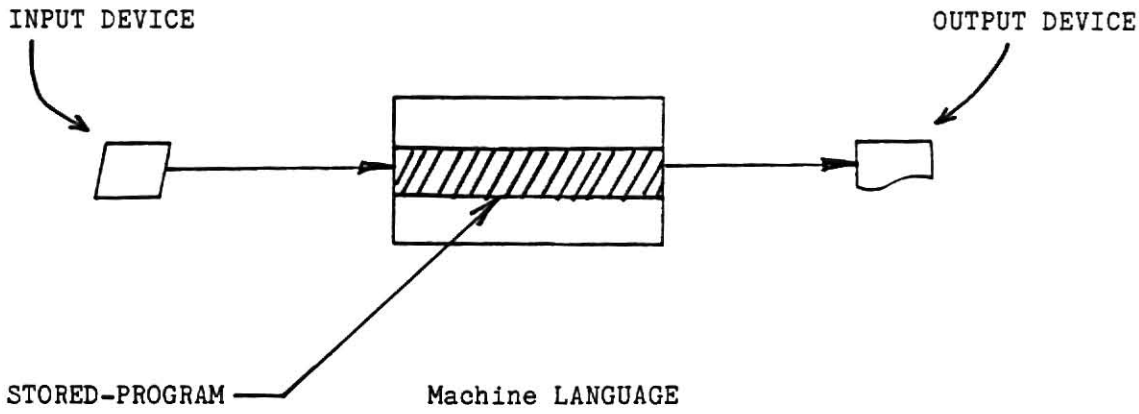


Figure 1a. NO CONCURRENCY IN SYSTEM late 50's-early 60's

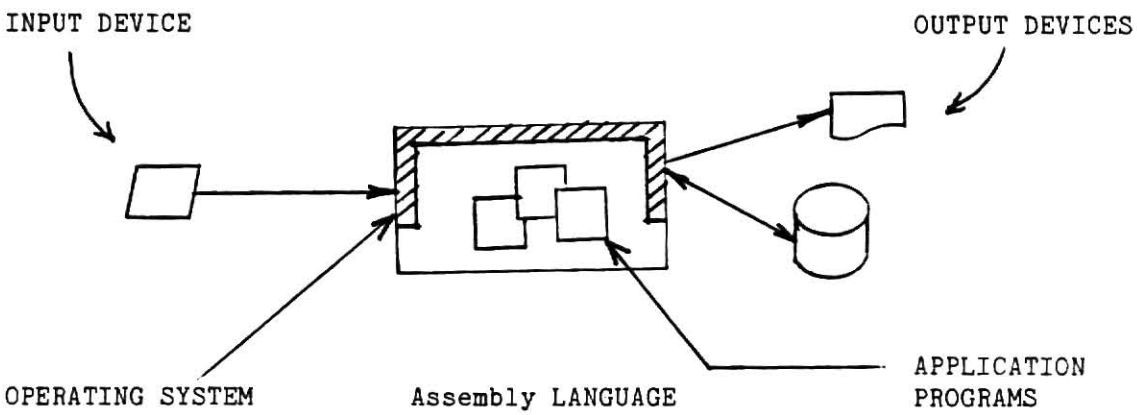


Figure 1b. CONCURRENT I/Os late 60's-early 70's

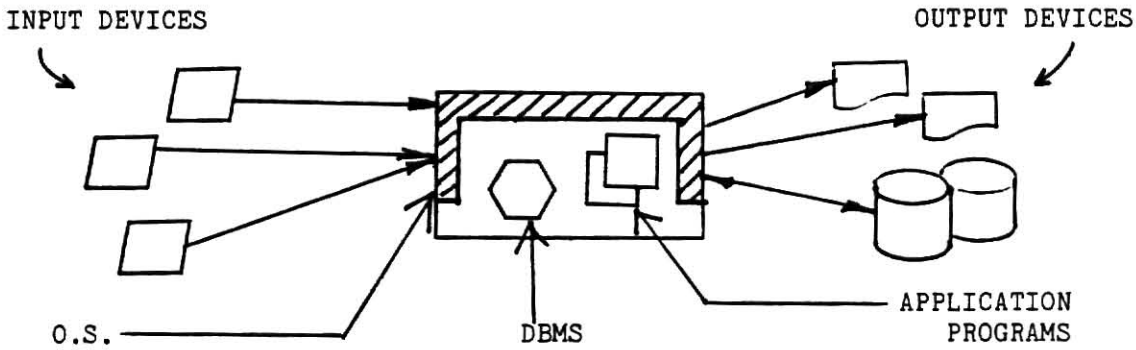


Figure 1c. CONCURRENCY IN MULTI-USERS SYSTEM late 60's-early 70's

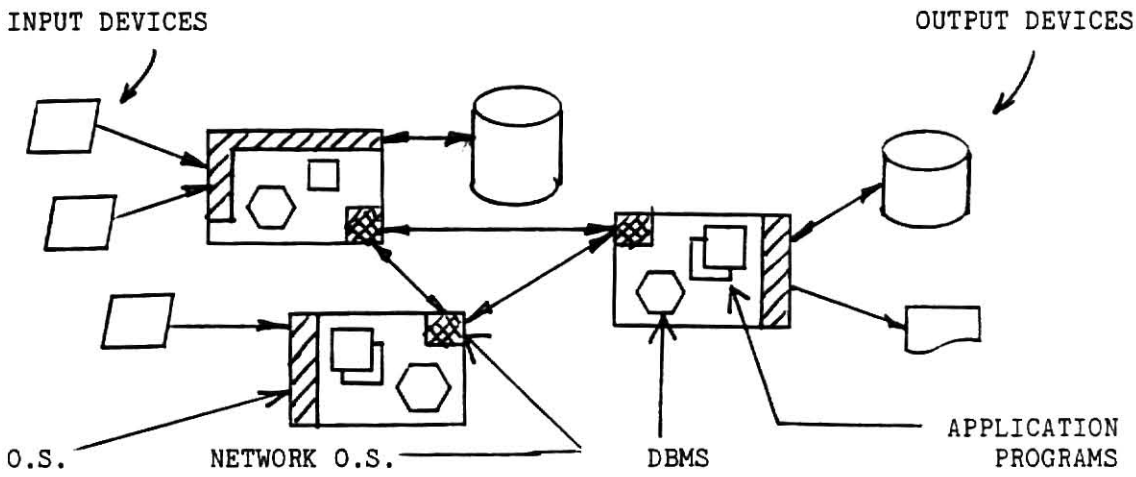


Figure 1d. CONCURRENCY IN A NETWORK SYSTEM late 70's-early 80's

polling techniques. The system is designed to operate with several sequential programs which seem to an observer or an user to be running "concurrently". If there is hardware parallelism present, the potential or virtual concurrency can be exhibited. Concurrency here is a virtual situation if only one CPU is available. That is there are several programs available in memory and each program is executed for a fixed quantum of time. Any concurrency which occurs is the overlap of I/O and CPU executions.

The sequencing of different programs in a time-share mode and scheduling of I/O devices in some orderly fashion requires an overall program which is known as an operating system (O.S.). The instructions of the program are written in symbolic form and translated into machine language by the machine for the native system. This operating system concept had a tremendous impact on the development of the computing systems for the next twenty years.

In retrospect it can be seen that the ability to view processes conceptually in symbolic form and to be able to implement the concept in hardware to run the processes is by far the most important innovation in the computer industry. The further enhancement of the symbolic form languages to higher level programming languages have provided the ability to view processes running in parallel.

Figure 1c shows a typical system of the early 1970's which provides sharing of a system's resources to multiple users by means of operating systems that are specialized for I/O devices. The concepts of sharing data and files are also introduced by means of programs written in higher level languages in some organized manner such as Data Base Management Systems (DBMS).

Figure 1d shows a typical system of the late 1970's which has several processors that are remotely located and connected by communication hardware to allow computational capability to be shared among the different sites. Again, a system of this type evolved with a specialized operating system for network capability and possibly a DBMS to manage shared data distributed over the network.

It can be seen that the architecture of each development over the thirty-year period has drawn upon the innovations of its predecessor, and the central theme of the development has been to increase the sharing of the computing resources. The formidable challenge for the task has been to provide potential and real concurrency and to provide synchronization of operations such that only non-destructive uses of data occur. The next section presents a more detailed discussion on control mechanisms that are available for concurrent operations.

2.2 Execution-Based Control Mechanisms

In order to understand the implementation of concurrency concepts in a computer system, it necessary to analyze the basic structure of such a system.

For analysis purposes, the basic structure is viewed as a hierarchy of four levels and only the control characteristics of each level is analyzed. Figure 2 illustrates four levels at which a computer system can be described.

Level one is a Sequential Machine which is based on physical things and hardware technology; level two is an Operating System (OS)

LEVEL	COMPONENTS	STRUCTURES
Sequential Machine	Transistors, Gates, Registers, Capacitors	Latches, Counters, Clocks, Decoders, Sequencers
Operating System	Sequential Machine, Instruction execution, CPU, ALU, Machine states, Memory blocks	Memory allocation, Instruction Set, MACRO, File system, I/O programs, queues,monitors, schedulers
DBMS	queues,File system	OS,Function routines
Programming	Utility programs, Function routines I/O Programs	Compilers, Programming languages

Figure 2. Control Components and Structures in Computer Systems

which utilizes level one structures and uses abstract and machine-dependent structures to manage physical resources; level three is a Data Base Management System (DBMS) which uses the structures of the Operating system and the constructs of programming level to control data; and level four is a Programming level which is based on expressing structures of the other three levels in a very compact fashion. Each level is further characterized by a set of components and a set of structures. The structures are composed of a combination of the components. There is recursion in describing the features of the structures. That is, a structure composed of components at one level might be a component at some other level.

2.2.1 Sequential Machine Level Control

The sequential machine level consists of the lowest level of electrical hardware such as transistors, capacitors, registers, and gates. The combination of these components provide a structure that can represent logic functions such as TRUE and FALSE. These structures, whose outputs are directly related to the inputs at any instant of time and can hold the values of the logic states over time, are sequential circuits.

The classical Von Neumann stored-program machine, which utilizes the sequential machine-level components and structures in combination, has been a fundamental building block for computer systems. The operation of a such a machine can be described as follows. The machine considers the program to be a set of instructions which are stored into contiguous storage locations. One register, called the Program Counter (PC), holds the information about the location at which an instruction

of a program is being processed. The other register called the Instruction Register (IR), contains the instruction to be executed.

In one particular state of the controlling machine, which is called Fetch-Next-Instruction (FNI), a series of operations are carried out in a cycle. The FNI cycle can be represented by:

```
repeat
    IR -- M[PC]
    PC -- PC+1
    {execute instruction in IR}
until machine STOP
```

The M[PC] instruction stands for the content of the memory at the location described by the value in "[ ]". The range of this value depends on the physical size of the memory and value that can be represented by the registers.

The sequencing of the machine continues until it encounters the STOP instruction, which is decoded as a state of halt. However, if the executed instruction calls upon a different value of the PC so that a different program execution can be started from some other location in the memory then we can run two programs in an alternate mode. In this case, if the previous value of PC before the 'jump' is saved in some register then the previous program can be continued by reloading the saved PC value into the PC register. This mechanism is extensively used at different levels of a system configuration.

Another mechanism used by the sequential machine is the periodic examination or "polling" of the status of an I/O device. The status can be checked by examining whether a bit is on or off in a status register of an I/O controller. Here the central processor and I/O controller of

the device operate in parallel. If the I/O device is slow in operation relative to the speed of the central processor then many executions of central processor instructions can be performed between the two consecutive pollings.

The polling mechanism is not very efficient because the central processor has to give up its 'working cycle' for the periodic check of the I/O status. To resolve this problem the idea of an interrupt is used, in which the I/O controller sends back its status or need of attention to the central processor when the I/O device is ready for more work.

If the exchange of information between the central processor and the I/O controller happens to be at a byte level then much time is wasted in handling overheads of registers and status checking. To improve on this situation, hardware capabilty is provided to access memory directly without bothering the central processor. Now the I/O controller can move a block of information such as 256 bytes or more and continue to service several I/O devices.

The next obvious mechanism to improve speed of an operation is to design an I/O controller that acts like a central processor with its own complete instruction set, including instructions for program control, looping, and testing. The central processor can create an I/O program in memory that the I/O controller can block move and execute. The central processor is interrupted only after the entire I/O program is complete (for example, IBM System/360 channel processor).

2.2.2 Operating System Level Control

The sequential machine mechanisms cannot be implemented in isolation nor can they be used without some comprehensive procedures to make them more effective. The conceptual process of "checking a bit condition in the I/O status register" requires several operations of the sequential circuits and these steps are identical every time the "process" is called upon. Thus, a process can be defined as a program module that consists of a data structure and a sequence of operations that operates on it. Based on this "process" concept, a designer can represent several processes operating in parallel by a single program module.

The operating system consists of a collection of processes represented by a set of program modules that provide control for communicating user requests to utilize hardware resources and processing information. Each program module executes sequentially but the execution of one or more modules may overlap, hence giving a system consisting of concurrent processes.

An operating system is organized to provide resource management and process management, for which the system requires some scheduling and communicating policies. In a single-processor system all processes cannot be executed at one time. Therefore, an operating system provides some reasonable schemes to handle other processes waiting to be serviced. There are at least three states through which a process could pass before it is completed. Figure 3 shows a transition diagram of the states of a process:

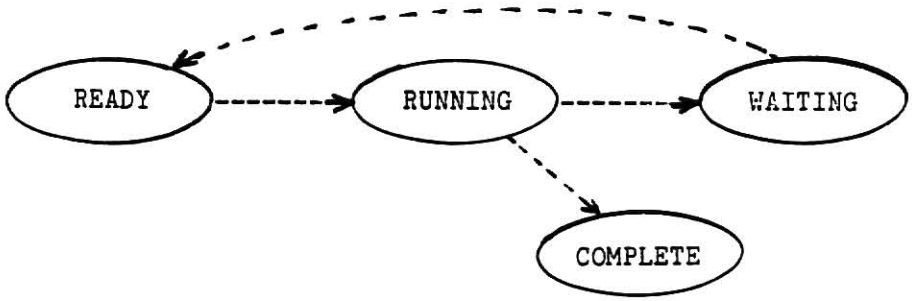


Figure 3. States of A Process in An Operating system

When a process is created it becomes "ready" for the CPU if some other process is already "running". When a running process has to wait for a request to be serviced then it is "blocked" and the ready process is running. The operating system maintains these transitions by means of mechanisms of sequential machines such as interrupts, queues, buffering, and stacks. The literature on operating systems offers many examples on how these mechanisms actually work [Brinch, 1973], [Habermann,1976], and [Holt et al., 1978].

2.2.3 DBMS Level Control

The DBMS level utilizes control for data of the system by means of the abstract view of programming languages. In contrast to operating systems which depend on resources and processes being in correct order, the DBMS is concerned with information, which is managed by the resources and processes, being in a correct state. The biggest problem is to prevent interference among users who are simultaneously updating information. The problem of concurrency control is similar to that of coordination of resources in operating systems. The control mechanisms

are software modules specifically designed to track transactions of the DBMS operations.

2.2.4 Programming Level Control

The programming level is a conceptual extension of other levels described previously. This concept is provided by means of high-level programming. The programming level applies the principle of data abstraction which requires using a high-level interface to hide the details of managing processes. This abstraction creates an integrated view of an operating system with a set of software to control an abstract machine. The operating system designed with programming control can offer the flexibility to affect only a small portion of software when a change in the hardware is required. This provides portability of application programs that can run on different systems.

The unique feature of the programming level is that it provides a representation for users to specify concurrent operations without describing details of the execution. This representation is created by combining components and structures of other levels.

2.3 Process Synchronization concepts

Synchronization is a general term for any constraint on ordering of operations in time to accomplish a common goal.

Sharing of resources in a computing system requires coordination and cooperation of processes. If processes are executing concurrently they may be either disjoint (process variables are not common) or joint (accessing common variables).

The result of the execution of a process is independent of the

progress of a process from which it is disjoint. Shared variable processes, on the other hand can affect the result of other processes. The interaction between two processes may be due to either sharing by competing processes or communication by cooperating processes. An example of competing processes is a process requiring system memory used by a another process. An example of cooperating processes is the familiar producer-consumer pair of programs.

An example of disjoint and concurrent processes requiring updates in a file A of a database x and in a file B of a database y can be shown in the form of a concurrent program as follows:

Assume File A= product info; File B= personnel info;

```
1.      Cobegin;
2.          open database x and add a new product
                                    type to file A;
3.          open database y and add 10% to all
                                    salaries in file B;
4.      Coend;
5.          read file A;
6.          read file B;
7.          close database x,y;
```

Since statements 2 and 3 operate on different files that do not affect the outcome in the statements 5 and 6, the files A and B will be read with correct values regardless which statements among 2 and 3 is executed first to update files A and B.

When the concurrent processes are changing data in a common file, then the result can be unpredictable. For the above example, assuming that both databases have identical copies of file A in the beginning:

```
                File A= personnel info;

1.      Cobegin;

2.          open database x and add 10% to all salaries
                                    in file A;

3.          open database y and subtract 10% from all
                                    salaries of file A;

4.          read file A;

5.      Coend;

6.          close database x,y;
```

In the above example, if statements 2, 3, and 4 are to be done simultaneously then file A will be read with either a no change in salaries, or increase in salaries by 10%, or decrease in salaries by 10% of all personnel associated with file A; depending on the relative speed of the execution of these statements.

A process executing at an unpredictable speed must perform some action that other process can detect in order to synchronize the two processes. The action can be either setting the value of a shared variable or sending a message. However, this synchronization can only work if 'perform action' such as setting or sending; and 'detect' action such as checking or receiving, are constrained to happen in the order 'perform action' followed by 'detect' [Andrews 83].

2.3.1 Synchronization Based on Shared Variables

When shared variables are used for interprocess communication,

two types of synchronization are used: 1) mutual exclusion ensures that execution is interleaved when the two processes are accessing shared variables. For an example, a sequence of statements called a critical section when executed as an indivisible operation would have to be mutually exclusive for the processes to access the critical section. 2) condition synchronization requires a delay of a process that needs a shared object which is being used by another process. For an example, a process accessing a memory buffer which is a shared object should be delayed if the buffer has no space.

Historically, the first solution to the mutual exclusion problem was achieved by hardware interrupt. An interrupt is used to indicate when an entry to a critical section is being attempted by a process. Thus, when a process wishes to enter a critical section, it disables further interrupts and on exit it enables interrupts for other processes. The advantage of such a technique is simplicity. However, the ability to disable/enable capability is spread throughout the system and has to be programmed very carefully; otherwise, processes with large critical sections can delay other processes indefinitely.

The early software solution of the mutual exclusion is based on the assumption that if two particular operations could be performed without interruption, such as exchanging the value of a location with a local register then mutual exclusion can be programmed more easily. To implement this idea a global variable 'exclusion' can be set to a unit value. When a process enters a critical section, the process must acquire the unit value stored in 'exclusion' and set the value of 'exclusion' to zero in a single indivisible operation. The process restores 'exclusion' to an unit value when the process leaves the

critical section. Any process that is unable to enter the shared resource must loop continually, this is called the Busy-Waiting. This method of solution is illustrated as follows:

```
var exclusion :0..1;  {shared }
    exclusion := 1;   {initialized}

        { Process x }

var local : 0..1;
    local:= 0;
    while local = 0  DO exchange (local,exclusion);

    { enter critical section x }

    exclusion := 1;
```

where

```
exchange (local,exclusion)        {this operation is
                                   indivisible  }
BEGIN
    local := exclusion;
    exclusion := o;
END;
```

This technique may be acceptable if there is little demand for the resource and the critical sections are short. Otherwise, a long critical sections will result into a long delay of processes which are trying to enter critical sections and will waste processors time in a system. Also, it is possible, due to electrical timing conditions and many other processes trying to test the condition of the shared variable, that some process may always find "exclusion" to be zero, and hence may never enter the critical section. To improve upon this situation of busy-wait, it is possible to put a process in the wait-mode and then call the process back when an entry is available to access the critical section.

Dijkstra was one of the first to appreciate the difficulties of using low-level mechanisms for process synchronization, and this

prompted his development of semaphores [Andrews and Schneider, 1983]. Since a process that is denied access to some resource is only interested in receiving a timing signal when the resource possessed by another process has relinquished it, it is sufficient to send a signal or a message that can be counted each time a resource is acquired or relinquished.

A primitive operation on semaphores is implemented by software under the protection of disabled interrupts. A queuing technique is used to avoid the busy form of waiting with the insertion of processes in the queue and removal of processes from the queue is done under disabled interrupts. Instead busy wait, a process relinquishes the processor and blocks itself as a result of switching mechanism implemented in the operating system. This mechanism consist of a semaphore that is implemented as a protected variable and a queue in which processes can wait for SIGNAL operations. When a process attempts a WAIT operation on semaphore and if it cannot proceed then it blocks itself to be placed in the queue.

A semaphore (S) is an integer value on which three operations have to be defined:

(1) Initialization to a non-negative value

```
S := >0;   {usually S set to value 1}
```

(2) WAIT(S)

```
IF S <> 0 THEN S:=S-1 ELSE put process to sleep in queue
   {decrement and test}
```

(3) SIGNAL(S)

```
IF queue empty THEN S:=S+1 ELSE awake a process from queue
     {increment and test}
```

An implementation of a semaphore mechanism in a system is done by indivisible operations at the hardware level. That is, the entire operation is performed in a single store cycle. WAIT and SIGNAL operations are used in a program as single, elemental operations. For example, semaphores are generally used without making any assumptions about which process will be waked-up first. This simplicity provides the process coordinator some flexibility to implement a process priority scheme suitable to a system.

In the above implementation, logically a value of a semaphore variable can become negative, indicating a number of halted processes and it is also specified in the implementation that semaphores take only non-negative values. However, the synchronization implied by this implementation is identical with that implied by the logical definition because a process never passes a wait operation if the value of the semaphore is zero (or less). For example, if the value of the semaphore S is initialized to be

```
IF S >= 0 THEN  S  ELSE S=0
```

This provides a semaphore invariant as follows:

```
Value(S) = Init Value(S) - NSIGNAL(S) - NWAIT(S) >= 0
           { NSIGNAL is number of signals and NWAIT is number of
           completed waits}.
```

This statement guarantees the synchronization property to be correct.

The following example of the READERS and WRITERS problem using semaphores is taken from Theaker and Brookes [22].

The READERS/WRITERS problem is a fairly typical problem that can be compared to an application of an airline reservation system where many inquiries on a central database are done by travel agents and booking offices. However, only one such inquiry can be allowed to

change the database to reserve a seat.

The problem is considered for a case where readers have priority over writers in accessing a common file. The problem arises in the following situations: 1) several concurrent processes wish to access a common file; 2) some wish to read the file, while some wish to write to the file. 3) shared accesses to the file is allowed for reading, but exclusive access is required for writing.

Exclusive access to write to the file is accomplished with a single semaphore W.

Writing

```
Initially W=1;

WAIT(W)
      exclusive access to
      change the file.
SIGNAL(W)
```

A variable READCOUNT, initialized to zero, is needed to keep track of how many processes are currently reading the file. The variable READCOUNT is accessed by all readers and must be protected as a critical region by controlling it with a semaphore X.

Reading

```
Initially x=0

WAIT(X)
     READCOUNT := READCOUNT + 1;
     IF READCOUNT = 1 THEN   WAIT(W)
SIGNAL(X)

     read the file

WAIT(X)
     READCOUNT := READCOUNT - 1;
     IF READCOUNT = 0 THEN  SIGNAL(W)
SIGNAL(X)
```

If a writer process is writing to the file, then the first reader process will halt on semaphore W.

As long as there is at least one process reading the file, any other reader processes can also access the file concurrently but must pass through the critical region protected by X one at a time.

The critical regions are difficult to manage, since one has to study the entire concurrent program to keep track of statements performing operations on resource variables which are dispersed throughout the processes.

In 1973 Dijkstra suggested that by combining all operations on a shared data structure into a single program module, one can simplify the understanding of process interaction [Brinch, 1973]. Hoare refined the concept by suggesting a single program module called Monitor which is an encapsulation of a definition of data structure and the operation on it in such a way that the components of the structure can be accessed only from within the procedure that define operations on it.

A monitor can be conceptualized in terms of a "fence" around critical data where all sequences of statements that manipulate the data are located within the "fence". This provides mutual exclusion in accessing shared data, since only one process at a time may be executing code within a given monitor "fence". This allows a resource, subject to concurrent access, to be viewed as a module. Consequently, a programmer can ignore the implementation details of the resource when using it, and can ignore how it is used when programming the monitor that implements it.

This can be illustrated with an example of a monitor, having two procedures P and V, which simulates a binary semaphore operation.

```
MONITOR semaphore;
VAR mutex : 0..1;
    positive : condition;


PROCEDURE P;
  BEGIN
     IF mutex = 0 THEN positive.WAIT;
     mutex := mutex - 1;
END;   (procedure P)


PROCEDURE V;
  BEGIN
     mutex := mutex + 1;
     positive.SIGNAL
END;   (procedure V)


BEGIN
    mutex := 1;                (initialization)
END;
```

```
( * Process 1 *)                    ( * Process 2 *)
   semaphore.P;                        semaphore.P;
      critical section;                   critical section;
   semaphore.V;                        semaphore.V;
```

In the above, if a process which calls the P procedure finds the value of semaphore to be zero, then it joins the queue associated with the condition variable 'positive'. It will remain there until some other process calls the V procedure, which changes value of the variable 'mutex' to one and then signals the first process on the queue 'positive' to resume. As with the semaphore solution previously presented, each critical section must be bracketed by calls to monitor procedures P and V.

The monitor construct provides a systematic way to isolate synchronization, i.e., processes no longer need to share data explicitly but shared data is gathered together and accessed within the monitor and complex process interactions are specifically ordered within

the monitor and thus easier to understand.

Other forms of control-passing are provided by condition variables along with the operations DELAY and CONTINUE (Hoare calls these WAIT and SIGNAL). A condition variable has no visible value, although it does have an initially empty queue associated with it. When a process executes the statement DELAY('condition') in the body of a monitor procedure, the process name is placed on the queue for 'condition'; the process is blocked from executing further, and the control of the monitor is released. When a process executes the statement CONTINUE('condition'), this process is temporarily blocked (unless the queue for 'condition' is empty), and one of the processes on the queue for 'condition' is resumed. Once this reawakened process leaves the monitor procedure, the process which executed the CONTINUE('condition') statement is resumed. The Solo operating system of Brinch Hansen is written in Concurrent Pascal, an extended version of Pascal, supports the type of monitors described [Brinch, 1977].

2.3.2 Synchronization Based on Message Passing

A different outgrowth from synchronization by shared variables is the method of synchronization by message passing, which can be viewed as extending semaphores and monitors to convey data as well as to implement synchronization. Message passing systems are devised in an attempt to deal with the problem of inadvertently altering variables that are shared by processes. If a system is not properly designed, then a process can destroy the operation of another process by interfering with those variables that are shared. On the other hand, when synchronization requirements become complex, their programming becomes

quite difficult. The underlying idea in a message passing scheme is to keep processes from sharing synchronizing variables and provide them with alternative means of communication by passing messages to one another.

In a message passing, scheme an operating system provides a message-passing process that is an exception in that it is the only process in the system that explicitly accesses shared data, and the only one that performs synchronization. Other processes, requiring shared data, synchronize indirectly via the message-passing process. Communication is accomplished because the receiving process gets data from the sending process. Synchronization is accomplished because a message can be received only after it has been sent, which constrains the order in which these two events can happen.

A minimum set of operations needed are a sender and receiver. Taken together, the sender and receiver define a communication channel. There are two types of channel-naming schemes that are used most often. One is referred to as direct-naming and second is referred to as global-naming or mailboxes. The direct-naming is the simplest scheme and easy to implement. In this scheme, process names serve as source and destination designators. Thus a message sent is executed by

SEND expression-list TO destination.

The message contains the value of data in expression-list at the time SEND is executed and the destination gives the programmer control over which process receives the message. A message to receive is executed by

RECEIVE variable-list FROM source.

Here the source designation gives the programmer control over where the message came from and first, get receipt of data from variable-list and,

second, subsequent destruction of the message. An example of how a card reader and a printer can interact to print card images is shown below:

```
PROCESS  reader;
    VAR card: cardimage;
    CYCLE
        read card;
        SEND card TO holder;
    END-cycle;
END (card-process);


PROCESS holder;
    VAR card:cardimage; line:lineimage;
    CYCLE
        RECEIVE card FROM reader;
        line:=card;
        SEND line TO printer;
    END-cycle;
END (holder-process);


PROCESS printer;
    VAR line : lineimage;
    CYCLE
        RECEIVE line FROM holder;
        print line;
    END-cycle;
END (printer-process);
```

In the above example, information flows analogously to the way liquid flows in a pipeline. Here information flows from the 'reader' process to the 'holder' process and then from the 'holder' process to the 'printer' process.

Another version of process interaction is a CLIENT/SERVER relationship. A SERVER process renders service to requests from several CLIENT processes. For example, on a network with only one printer that is shared by several users, then the jobs created by the users have to be synchronized to be printed on the printer. In a case like this, RECEIVE in print-server should allow receipt of a message from any user process. If there is only one user, then the direct-naming method of

communication can work; but difficulty arises due to many client-processes which at the very least require one RECEIVE for each processes from a server-process. To resolve this one to many communication situation use of global naming or mailbox scheme is used. Here a mailbox is assigned as a destination designator in any process' SEND statements and as a source designator in any process' RECEIVE statements. Thus, clients send their requests to a single mailbox from which a server receives service requests.

Source and destination designations may be fixed at compile time, which provides static channel naming, or they may be computed at run time, which is called dynamic channel naming. Static channel naming presents the problem of not being able to communicate along channels not known at compile time. In many applications, such as file systems it is more desirable to allocate resources dynamically.

2.3.3 Other methods of synchronization.

Mutual exclusion is a mechanism that forces the time-ordering of execution of pieces of code called critical section. Furthermore, the explicit programming of critical sections using semaphores and monitors requires programmers to specify the implementation and constraints of operations of synchronization at one place in programs. The language-based approach to process synchronization aims to facilitate designing systems by defining the semantics of a programming language in such a way that the language cannot express time-dependent errors.

A class of synchronization mechanisms called Path Expressions [Campbell and Habermann, 1974] provides a way to specify all constraints of execution at one place in each module of each processes. Moreover,

code to enforce the constraints is generated by a compiler. The path expression mechanism describes synchronization at the level of procedures. That is, when two actions need to be synchronized, each action must be provided by a separate procedure invocation. Thus, this mechanism allows statement of what action sequence is permissible, in contrast to mutual exclusion schemes in which the main function is to delay actions. The new mechanism specifically describes the synchronization permissible between procedures and prohibits all others.

The idea underlying the path expression mechanism can be described as follows: A path expression names the procedures whose execution by processes are to be synchronized. The naming of procedure is done by a specification which tells exactly the way in which the synchronization is to take place. Each path expression is implemented by a "controller" which decides when the procedure execution should be allowed to proceed, and therefore process to continue. The controller operates as follows: Each procedure starts with a 'begin' and finishes with an 'end'. A process executing the 'begin' of a synchronized procedure checks with the controller to determine whether it may proceed. Based on the synchronization specifications, the controller either delays the process for the execution of requested procedure or the process is allowed to continue. Finally, when the process executes the end condition of the procedure then the process notifies the controller. Now the controller may be able to release other delayed processes.

There are four basic synchronization schemes which in combination can provide more complex path expressions. The notation used for describing these schemes simplifies the construction of

controllers. These four schemes are 1) Sequence of Actions 2) Selection from A Set 3) Repetition and 4) Simultaneous Execution. Sequence of actions (action means execution of procedures by a process) permits each action to occur in the order specified. If three procedures OPEN, READ, and WRITE are to be sequentially synchronized, then

path OPEN ; READ ; WRITE end

is an example for a path expression in which notation ';' signifies that the three procedures are to be executed one after the other in the sequence given. The procedures may have been invoked by separate processes, in a different order, and with possible delays. If the invocation of READ occurs first, the invoking process will be delayed until procedure OPEN has been executed.

A selection from set of action scheme permits only one action to occur from the set of procedure specified by a path expression. For example,

path READ , WRITE end

will allow the select on one of the procedures from the set {READ and WRITE}. Here the notation ',' in the path expression signifies selection. The process attempting to execute the procedure selected is allowed to continue while processes attempting to execute those procedures not selected are delayed until a new selection is made from the set. In this example the path expression specifies a series of executions by processes of the procedures READ and WRITE in unpredictable order, none of which overlap in time.

Repetition permits a path expression once completed to be repeated. This is represented by enclosing a path expression between the key words 'path' 'end'. However, repeated path expressions are now

allowed to be embedded within other path expressions. All previous examples are of repetition type.

Simultaneous execution permits several processes to execute given procedures concurrently. The notation representing this scheme is a bracket pair '{ }' placed around a regular expression. These brackets are not allowed to be nested. An example of this type is

```
path  {READ}, WRITE  end
```

Here READ execution may overlap other READ executions but WRITE executions may not overlap with other READ or WRITE executions. Thus, once READ execution starts, it will continue for as long as there are processes requesting READ and at least one process executing READ.

All four basic schemes can be combined to form more complex path expressions. This can be demonstrated by the following construct [Campbell and Habermann, 1974] that duplicates the READER/WRITER problem described in section 2.3.1 for readers having priority over writers:

Let readers have priority over writers: from the moment a request to read a file is received no writer can access that file, and reading request is granted immediately after finishing any writing which is already in process.

```
path   WRITEATTEMPT   end
path   {REQUESTREAD} , REQUESTWRITE   end
path   {READ} , (OPENWRITE ; WRITE)  end
```

```
Where REQUESTWRITE = begin  OPENWRITE  end
      WRITEATTEMPT = begin  REQUESTWRITE  end
      REQUESTREAD  = begin READ  end
      READ = begin REQUESTREAD  end
      WRITE = begin WRITEATTEMPT; write END
```

The first path expression assures that only one writer can request writing. The braces in second path serve the purpose of

allowing all subsequent readers to go through so that reading will not be interrupted by writers. The braces in the third path allow an overlap in reading.

Although path expression schemes provide a way to express synchronization constraints described operationally, they are poorly suited for specifying condition synchronization, because execution of an operation may depend on the state of a resource that is not directly related to the history of operations already performed. For example, certain variants of the READER/WRITER problem (such as writers preference or fair access for readers and writers) may require access to the number of waiting readers and waiting writers in order to implement the desired synchronization. In order to use path expressions to specify solutions to such problems, definition of additional operations on the resource may be sufficient. However, path expressions have been useful for specifying the semantics of concurrent computations.

2.4 Concurrency Control in Database

The main purpose of any database system is to provide a measure of data independence, so that programs that access the database are immune to changes in physical or logical data structuring. Thus, one of the central issues in DBMS has been to maintain the integrity of a database in the face of concurrent updates. To accomplish this, a DBMS provides concurrency-control mechanisms that prevent interference among users who concurrently attempt to update database.

Concurrency control in a database is concerned with deciding what actions should be taken in response to requests by the individual processes to read and write into the database. Concurrency control

provides ways to cope with conflicts of separate transactions being processed at the same time. To cope with this situation, the main focus is in two areas, safe locking policies and serializability, where efficient solutions are being investigated.

In order to understand the concurrency concepts in databases, some definitions used in the database literatures need to be introduced here. A database consists of a set of inter-related named objects called 'entities'. The relationships among these entities must be maintained at all times. These relationships are prescribed by the consistency requirements or integrity constraints of the database. When an user accesses or updates a database , he may have to temporarily violate the consistency requirements [Bray, 1982]. For example, in a banking system, there may be no way to transfer funds from one account to another account in a single atomic step without temporarily violating an integrity constraint that the sum of all balances equals the total liability of the bank. Since consistency constraints cannot necessarily be enforced at each primitive action on entities such as read and write, sequences of actions are grouped to form what are called 'transactions' and each transaction is an unit of consistency. Each transaction must transform a database from a consistent state to a new consistent state.

Although transactions are guaranteed to preserve consistency when run in isolation from other transactions, concurrent execution of transactions and various failures occurring during transaction processing could cause such anomalies as lost updates and inconsistent retrievals. These anomalies can be illustrated by the following examples of on-line electronic funds transfers from remote terminals:

Example 1: Lost updates.

Consider two customers simultaneously trying to deposit into the same account. Initially the account has {$500} (the value in { } shows database condition).

Customer 1		Customer 2	
T1.1	READ Balance {$500}	T2.1	READ Balance {$500}
T1.2	ADD Deposit $100	T2.2	ADD Deposit $200
T1.3	WRITE result to Database {$600}	T2.3	Write result to Database {$700}

When each customer executes transactions in the absence of concurrency control, the terminals could read the account balance at approximately same time, compute new balances in parallel, and store the new balances back into the database depending which transaction occurs first. For example, if T1.3 is done before T2.3 then database will reflect {$700} and if T2.3 is done before T1.3 then account will have {$600}.

Example 2: Inconsistent retrievals.

Consider two customers simultaneously execute the following tansactions.

Customer 1 transfers $1000 from saving account {S} to a checking account {C}.

Customer 2 prints out the balance totaled from {S} and {C}.

Customer 1	Database Condition	customer 2
	{S} has $5000 {C} has $100	
T1.1 READ {S} = $5000 WITHDRAW $1000 WRITE result to Database	{C} has $100	
T1.2 READ {C} = $100		T2.1 READ {S} = $4000
T1.3 ADD $1000		T2.2 READ {C} =$100
	{S} has $4000	

T1.4 WRITE result {C} has $1100

T2.3 PRINT {S}+{C}
$4100

If concurrently executing transactions have the ordering of T1.1, T1.2, T2.1, T2.2, T1.3, T2.3, T1.4 then, unlike Example 1, the final value placed in into the database by this execution is correct, yet the execution is incorrect since the balance printed for customer 2 is incorrect.

These two examples illustrates ways in which users can interfere with each other and are typical of the concurrency control problems that arise in DBMSs [Bernstein and Goodman, 1981].

To prevent the previously described anomalies from occurring, it is usually required that for a given concurrent schedule of transactions there exists some serial schedule that is equivalent to it.

An update operation in a database is usually a two-step procedure. First, the data is retrieved from the database and second, the data is modified and a new value is written back to the database. Thus, a synchronization problem can occur between these two steps if some other user can modify the data before the step two is completed. To avoid this problem, the simple solution is to lock the data item until the new value has been stored. This forces all updates to be done serially rather than concurrently and also the updating process is indivisible so that another user cannot begin to process data in the middle of the update. In a complex transaction environment many update operations can take place where entities may be stored in several records and this involves locking/unlocking all the records when they are updated.

There are different methods for solving the update synchronization problems. Some of them provide general solutions, whereas others are applicable only if certain constraints are placed on the system. Techniques applicable for update synchronization in distributed DBMSs provide a subset of solutions for centralized DBMSs, therefore the following discussion will address update synchronization in distributed DBMSs.

The task of concurrency control is the same whether the database is centralized or distributed. Concurrency and synchronization between several processes require the exchange of data. In a centralized system such exchanges are controlled by means of mutual exclusion using shared variables in common memory. In a distributed system, these exchanges are made through communication channels. However, the problem is a little more complex in distributed database because (1) users may access data stored in many different locations and (2) a concurrency control mechanism at one computer cannot instantaneously know of interactions at some other location.

In a distributed system each site has its own memory and there is no common memory among the sites. The communication between these sites takes place based on following three assumptions:

1) Messages sent through a communication channel are not lost nor altered.

2) For any two sites X and Y, the order in which X sends messages to Y is identical to the order in which Y receives from X.

3) The failure or down condition of a site is detected and signaled to all sites which attempt to communicate with that site.

If a distributed DBMS allows a partitioned database, that is a

single copy of the data, then centralized DBMS procedures for synchronization are adequate for the system. However, with compound requests or multiple copies of data, a 'global locking' approach is a simple extension of a locking approached used in a centralized system. In a global locking approach all copies of the data are locked, update is made to every copy, and then all locks are released. This approach provides a strong degree of consistency and can be used for any situation, however, its performance is poor, communication gets expensive, and is time consuming. To implement this approach, five actions must be satisfied.

1. Request each node for lock.
2. Acknowledgement from each node about granted locks.
3. Issue updates to nodes.
4. Acknowledgement from each nodes of update.
5. Release all locks at all nodes.

These five messages are the minimum required for a successful update. The number of messages involved for a given update can depend on the number of successful completions for the first attempt to get locks. For example, if there were ten copies of data on ten different nodes, a request might succeed in locking nine copies; however, if another request had acquired the tenth copy then a potential deadlock condition would have to be resolved by a rollback and a restart. Communication for the purpose of requesting locks can be achieved two ways. Assume that there are four nodes (P, Q, R, and S) with requesting node P. In a serial communication approach, node P would a send request to node Q. If data were already locked at node Q then the

request would be rejected. If node Q could lock the data, then it would do so and pass the request to Node R. Node R would lock the data and pass the request to node S. Finally, Node S would lock the data and return an acknowledgment to node P that all copies were locked. The update request and unlock request would use the same serial approach for the communication. The other alternative for communication is parallel communication where a request is broadcast to all nodes simultaneously. Since a link or node failure can prohibit operation of update synchronization, the important thing in global locking is the reliability of a system communication facility.

Another approach in update synchronization is called the dominant copy method, which is more appropriate for certain patterns of data usage, such as high locality of reference of update, low frequency of updates, and updates which are not time critical. In the dominant approach, one of the copies of data is considered a dominant copy of the data to which all updates are sent first. The node at which this copy of data resides is called a dominant node. Eventually, all copies of the data are updated, but the dominant copy of the data is updated first, regardless of where the update request originated. It is obvious that for a large volume of transactions, a dominant node could become a bottleneck; however, unlike the global locking approach much of the communication overhead is reduced because only the dominant copy of the data needs to be locked. Once the dominant node updates a dominant copy of the data, it then controls the updating of all of the other copies of the data. This might present a problem of getting the latest data, but a policy of requesting retrieval from the dominant copy can assure a user of getting the latest version of the data.

As mentioned previously, if each transaction in a set of update transactions is executed to completion before the next update begins then that serial execution is defined to be correct. To provide such serialization in a multiple update environment a synchronization techniques based on timestamp ordering is used (T/O). Each transaction is assigned an unique timestamp by a transaction manager (TM). The basic of T/O implementation requires building a T/O scheduler that processes read and write operation based on some criteria. An update application rule determines the action a node needs to take to implement an update. A node compares the timestamp on its copy of data with the corresponding timestamp in the request. If the local timestamp is earlier, then the update is allowed. If timestamp on the local copy is later than the timestamp in the request then the update is not performed. A problem arises in generating a timestamp that is unique in time in a multiple node environment. Since in a distributed system each node has its own clock, it is necessary to synchronize them precisely to ensure that there is effectively only one time throughout the network. One way to resolve this problem is to supply a node number with each timestamp.

Synchronization and concurrency problems have been adequately solved for centralized systems by mechanisms similar to one used in operating systems. That is providing locks/unlocks and mutual exclusions on data items. However, in distributed systems many approaches are still being developed and tested. No single synchronization protocol is adequate for all distributed systems.

CHAPTER III

3.0 Implementation Review In Commercial Systems

This chapter provides information on how some of the commercial products have attempted to implement the concurrency and synchronization ideas that were discussed in the previous chapter.

From the business point of view, implementing concurrency in commercial products is looked upon as keeping a balance between configuring products that provide a maximum sharing of resources and producing products at a minimum cost with an acceptable reliability. On the other hand, it is the customer who balances the need versus performance (that is, how much concurrency and sharing are needed) to force developers to implement concurrent concepts in the products.

There are three types of products that are reviewed in this report. The first is a file server, the second is operating systems, and the third is a distributed database. The file server is marketed by NCR Corporation to be used by micro-computers and host systems on some type of networks. There are two general purpose operating systems that are used on small computers. The UNIX multi-tasking operating system developed by Bell Laboratories is used on micro and mini computers. The Concurrent CP/M-86 operating system is marketed by Digital Research for IBM personal computers. The distributed database system SDD-1 is developed by Computer Corporation of America and is implemented on the DEC-10 and the DEC-20.

An attempt was made to find detailed documentation as to how the implementation of concurrency and synchronization is done in currently

available commercial products. Since most of the information is proprietary and is licensed by vendors, this effort met with little success. However, the available information is sufficient to convey many ideas on what types of concurrency and synchronization mechanisms are used, and this the basis for the following discussion.

3.1 NCR 6600 Intelligent I/O Controller

NCR 6600 I/O Controller, known as MODUS™, is a peripheral subsystem that provides disk and printer I/O services to microcomputers connected via a network. The microcomputers access MODUS to share peripherals(disk and printer) through a set of communication processors and can share files on the disk.

Figure 4 shows the basic MODUS configuration. The I/O processor (I/O PROC) communicates with peripheral devices, such as disks and printers, that are connected to MODUS. The communication processor (COMM PROC) communicates with network systems. The I/O PROC and COMM PROC communicate with each other through a bus known as MULTIBUS™ which is an industry-standard physical interconnects for microprocessor controlled boards. Each MODUS unit can have a maximum of four COMM PROCs at one time. The function of a COMM PROC in MODUS is to receive and translate network-protocol-request message packets into a format suitable for the I/O PROC and similarly to translate and send I/O PROC responses in packets suitable for a network protocol.

The MODUS is accessed through three types of communication protocols: RS-232-C, point-to-point which is a low speed asynchronous point-to-point network; NCR OMNINET™, which is a multi-drop medium speed Local Area Network; and Binary Synchronous MODUS Batch File

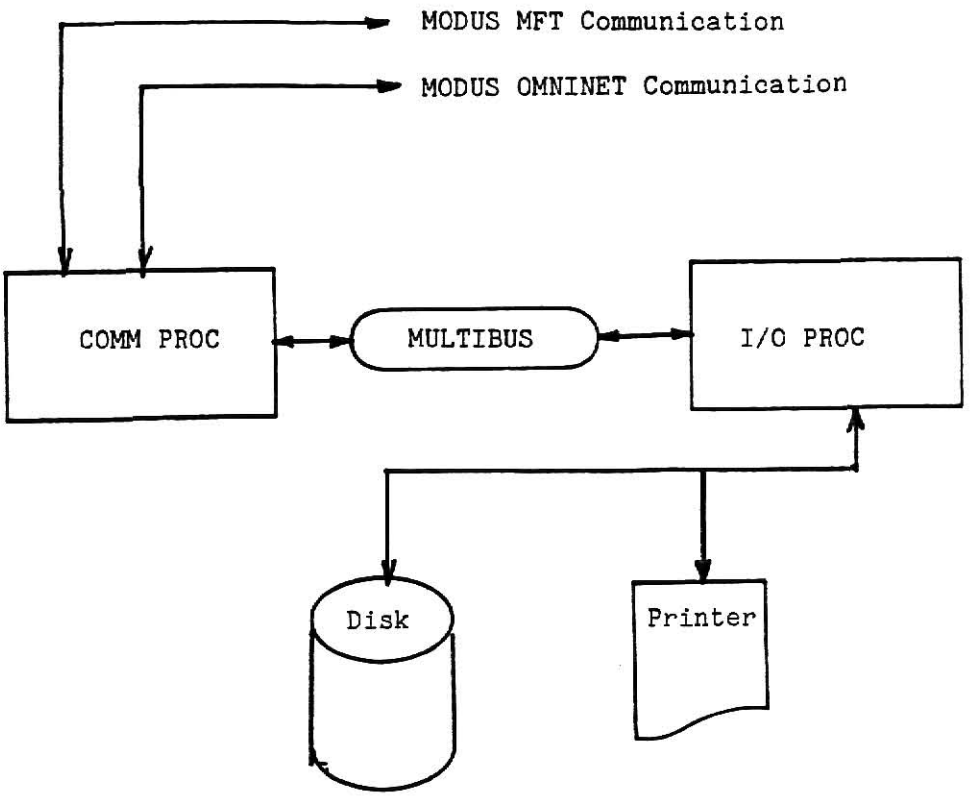


Figure 4. NCR 6600 File Subsystem

Transfer (BFT) which is a version of the industry-standard IBM 2780 or 3780 bisynchronous protocol.

Micro-computers share data files and programs using RS-232-C and OMNINET network protocols over networks. The BFT protocol is used for file transfer between MODUS and a high-order processor or a host system. There is a difference in the way in which MODUS is shared by micro-computers and host systems. The difference comes from the way network micro-computers access data on MODUS; the network systems actually process data by handling the files at a record level. On the other hand, the interaction between a host and MODUS happens at the file level only. BFT is used to send a file containing data collected on MODUS network to a host system and the host system processes the file and transfers the updated file back to MODUS.

A file transfer synchronization using RS-232-C and OMNINET protocols requires a message packet transmission to transmit requests and responses between MODUS and micro-computers. In the RS-232-C mode, a dialogue between MODUS and a micro-computer is set up in a sender-receiver mode. A transmission is initiated by a sender with a request which is responded with acknowledged (ACK) code by a receiver. This follows by the sender issuing a message passing information of destination, micro-computer identification, and data length, which is confirmed by the receiver with another ACK code. Finally, the sender starts the actual data transmission followed by an end-of-transmission (EOT) message. The receiver confirms the EOT message by another ACK. If a transmission error occurs, the receiver sends back the not-acknowledge (NAK) code, or the sender detects a time-out condition if no response is received within some time limit.

In the NCR OMNINET protocol, a micro-computer issues a request message to MODUS and receives a response message from MODUS, giving the status of the request and any requested data. The micro-computer follows by an acknowledge response. This is similar to the RS-232-C protocol but uses a different message format.

The MODUS BFT protocol transfers files in groups of records called transmission blocks. Each block contains one or more data records from a file and a set of control characters. These control characters mark the start and end of the block, which provides a means of checking for transmission errors and a means of separating records within a block. The actual format of these control characters are based on the industry-standard IBM Binary Synchronous Protocol.

The function of COMM PROC in MODUS is to receive and translate network-protocols-message packets into formats suitable for I/O PROC and similarly to translate and send I/O PROC responses in packets suitable for network protocols. For the interaction between COMM PROC and I/O PROC, a block of resident memory for a communication channel is set up. This memory is controlled by one of the COMM PROC board. The memory is used for passing requests codes, statuses, buffer addresses, and other data. Thus, all interaction are taking through this block of resident memory.

When a valid request is made from a network computer, then COMM PROC sets an identifier code into a region of shared memory area and sets a flag to I/O PROC. This flag causes I/O PROC to issue an interrupt to initiate a task to examine common memory block which has a identifier code to determine which memory block is to be processed. After the completion of the requested function, I/O PROC signals

COMMPROC that function was performed.

The I/O PROC contains a collection of software modules, known as Controlware® that perform file service requests from microcomputers in a network. Each module performs a specific function, such as managing the memory buffers or controlling print files. The Controlware is a version of TurboDOS® software, which is a CP/M operating system alike software developed for file processing and sharing on networks.

The primary purpose of MODUS is to provide a service to microcomputer users on a network that can share hardware resources such as high capacity magnetic disks and various types of printers; and files such as programs and data. To provide such a service, some file organization has to be set up so that users can communicate with each others without any interference.

The file organization on MODUS is based on a concept of grouping files according to user libraries. There are 32 different libraries numbered from 0 to 31 of which only first 16 can be accessed through the network. All file operations are performed on the files in the library that an user has currently logged on. For sharing purposes, each file in MODUS is assigned a file attribute that characterizes the file for access privileges. A READ_ONLY file can not be written to, deleted, or renamed. A GLOBAL file in user library 0 can be accessed from any other user library provided the appropriate access mode is satisfied. A GLOBAL file accessed from a library other than 0 is limited to read-only access. A file with an F1 attribute (normally assigned to program files) either causes all data files opened by that program to be opened as shared files or changes the default open mode to permissive mode instead of exclusive mode.

The file sharing on MODUS is based on four different access modes. The modes used are:

1. In a exclusive access mode, a read-write file opened for exclusive access can not be opened by another user until the user opening the file closes the file.

2. In a read-only access mode any number of users can simultaneously open a file having the read/write attribute for read-only purposes.

3. In a shared access mode, a file can be opened by any number of users simultaneously and all users may write to it. However, in this mode record lock and unlock functions are necessary to control record locking if more than one program is to write to the file.

4. In a permissive access mode, a file can be opened by any number of users simultaneously and can be read by all users without restriction. If any user writes to the file, that user can continue to write records, but no other user can write to the file until the first user closes the file. The records written become immediately available to all users.

With the exception of the shared access mode which is a user definable, all other access modes are explicitly handled by the operating system. The record-level interlocks for files opened in the shared mode require explicit cooperative participation by all updating programs. A program must use the LOCK-RECORD function before reading a record and must use the UNLOCK-RECORD function after updating the record. The LOCK-RECORD function returns an error code to a program attempting to lock a record which is already locked by another program, and the program must try again until successful. Alternatively, the

operating system can be instructed to automatically suspend the program until the lock request can be satisfied.

To facilitate inter-process and inter-user communication, the MODUS operating system supports a special kind of file structure called a FIFO (first-in first-out) which is opened, closed, read, and written exactly like any other file. However, the difference between a FIFO and ordinary files is the way in which it is created and used. A record to a FIFO is always appended to the end and a record is always read and removed from the begining. There are two characteristics of a FIFO structure that provide interprocess communications. The first characteristic is that an attribute indicates whether a FIFO is disk-resident or memory-resident, and the maximum number of records the FIFO may contain. Memory-resident FIFO are limited in capacity (less then 127 records) but provide high speed in execution. Disk-resident FIFOs provide large capacity (65000 record) but slower speed. The second characteristic is that reading from an empty FIFO returns an end-of file condition and writing to a full FIFO returns a disk-full condition. However, if an attribute called 'suspend indicator' is set, then reading from an empty or writing to a full FIFO causes a process to suspend until the FIFO becomes non-empty or non-full.

3.2 Interprocess Communication in UNIX™

UNIX is an operating system developed at Bell Laboratories for PDP-11 series computers . It offers a number of features such as the ability to initiate asynchronous processes and compatible file, device, and inter-process I/O. UNIX is a multi-user, interactive operating system that provides file handling capability. The user interface in the

UNIX is a very powerful feature known as a Shell. The user and user programs generally interact with the Shell rather than making direct contact with the operating system. The Shell program has a language of its own right and manipulates items that are not variables and arrays but are the names of programs and files.

The system handles three types of files: ordinary files, directories, and special files. Of these three types of files, the special files are system-created to associate with I/O devices. This way file and device I/O are similar as to have name and meaning.

Interprocess communication is based on two facilities known as pipes and signals. Pipes are special files used for data transfer between processes and signals correspond to wait signal mechanisms. A process creates a pipe by system service request. Pipes may be inherited by one or more subprocesses or children of a process. The system primitive returns two descriptors, one for reading and one for writing, so that a process can pass to its descendent the capabilities for reading or writing or both. A pipe ceases to exist when all processes owning descriptors for it terminate. A pipe is supported by an implicit synchronization mechanism which prevents writers from getting ahead of readers by blocking the writers until the readers catch up.

A pipe can be used by processes having a common ancestor which sets up file description for the pipe. Writing messages to a pipe when no readers are present results in an error and reading from a pipe when no writers are presents results in an end-of-file condition. When multiple processes read or write to a pipe then the identity of producers is not preserved so that data from different writers may be

interleaved for the consumers reading the pipe.

The signal synchronization primitive is a system call that allows a parent process to suspend until a child process terminates. When a child process completes a a process, all its resources are released and a signal is passed to the parent process causing it to wake up and continue processing.

An example of how basic I/O redirection and process handling mechanisms is accommodated in the Shell is given as follows:

Consider a system having 100 terminals which are numbered TT1 through TT100. Suppose it is desired to generate a file with the names of all users who are on the terminals numbers TT1 through TT50. A command sequence to do the job without pipelines can shown as:

```
WHO >tempfile
GREP  { 'TT[1-50]'  tempfile >resultfile }
RM tempfile
```

The command WHO creates a file called tempfile that has a list of currently logged-in users with format such that each line contains user name, terminal number, and sign-on time. The second command GREP is a pattern matcher that can search a string for a match to a pattern that is supplied as a part of the command. The result of this pattern match is transferred to result file. The RM command deletes file.

The pipeline facility is developed to do away with manipulation of a temporary file and allows the task to be run as one entity rather than three sequential processes. The command structure of the pipelined I/O is

```
WHO ¦ GREP { 'TT[1-50]' >resultfile }.
```

The Shell implements this command structure as follows. It defines a

temporary file of type pipeline and puts its descriptors in the place where the WHO command will look to find its output file. A child process is generated and starts up the WHO command in it. Next it places a descriptor for the same temporary file into the place where the GREP command looks for a description of its input file. A child process is generated and starts up the GREP command in it. The shell which has acted as a parent process goes to sleep awaiting a signal which it will receive when the last process in the pipeline terminates.

Pipelines are not not limited to two commands; many commands can be strung together in this way. The Shell essentially sets up all these commands to run by setting up sequentially the output from one pipe to input to another. The synchronization takes place due to I/O drivers working with pipeline files that suspend a process that tries to read an empty pipe or one that tries to write a full one.

3.3 Synchronization in Concurrent CP/M-86

Concurrent CP/M-86 is a single user, multitasking operating system that runs multiple programs simultaneously by dividing tasks between virtual consoles. This operating system is suitable for personal computers.

Concurrent CP/M-86 creates a process associated with each program. It is the process, rather than the program, that controls all access to the system resources through the operating system. It is Concurrent CP/M-86 that monitors the process, and not the program.

Processes running under Concurrent CP/M-86 have two categories. First, transient processes that run programs loaded into memory from a disk. Second, resident processes that run programs that are a part of

the operating system itself.

The following list shows a few capabilities of Concurrent CP/M-86:

- A single user can run multiple programs.
- Real-time process control allows communication and data aquisition without loss of information.
- Interprocess communication by system queues and interrupt mechanisms using flags.

Concurrent CP/M-86 has several software modules that provide support for the above mentioned capabilities. There are two modules that provide functional support for the process and file handling. They are: 1) The Real-Time Monitor (RTM) and 2) The Basic Disk Operating System (BDOS).

RTM performs process dispatching, queue management, flag control, device polling, and system timing tasks. The primary task of RTM is to transfer CPU resource from one process to another. This task of allocating the CPU is done by a part of the RTM called the Dispatcher. Each process has two data structures called Process Descriptor (PD) and the User Data Area (UDA) which are used to save or restore current state of a running process. Each process in the system resides in one of three states: ready, running, or waiting (similar to the states shown in figure 3). Concurrent CP/M-86 uses a priority scheme; therefore, RTM selects the process with the best priority for run. Processes with equal priority are scheduled in a round-robin manner and are given equal shares of resources.

The Queue management of the RTM provides the function of communicating messages between processes for synchronizing process

execution and for mutual exclusion. Each queue consists of a descriptor and a buffer which are special data structures. A process can read or write a message to a queue conditionally or unconditionally. When a queue contains no message during conditional read or when a queue is full during conditional write, then the system returns an error code to the calling process. However, if a process performs an unconditional read operation from an empty queue; the system suspends the calling process from execution until another process write a message to the queue.

A Mutual exclusion queues is a special type of queue which contains one message of zero length. Thus, a mutual exclusion queue is a binary semaphore which ensure that only one process has access to a resource at a time.

Operation of access to a resource protected by a mutual exclusion queue can be described as below:

1. A process issues an unconditional Read Queue call from the queue protecting the resource.

2. This results in the process being suspended until the message is available.

3. The process then acquires the resource.

4. The process writes the message back to the queue when the resource is released, thus enabling other processes to perform operation 1 and 2.

The BDOS module is an extension of single-tasking CP/M-86 BDOS with additional features of file locking,shared access to files,date stamps, and password protection. As a special option, users can open the same file in a shared or unlocked mode. The system supports commands that will allow record locking and unlocking for files opened

in this mode.

For the purpose of sharing files, BDOS allows three different modes for opening files. 1) In the locked mode, a process can open a file which is not opened by another process, and once opened the process owns the file until the file is closed or the process terminates. 2) In the unlocked mode, a process can open a file if the file is not currently opened, or if the file is already opened by another process in unlocked mode. In this mode several processes can open a file for the purpose of read/write operations with the exception that if the file is read-only, then a process cannot write to it or if the file requires a password to open it than the process cannot access the file. 3) In the read-only mode, a process can open a file if the file is not currently opened by another process or if the file is already opened in the read-only mode by another process. In this mode several processes can open a file, but for the read only purpose.

The BDOS module provides two different approaches to the mechanism of record update coordination: record LOCK/UNLOCK function, and TEST-AND-WRITE function. When a record is LOCKed, BDOS allocates an entry in system Lock list and keeps track of the locked record and associated calling process. While the record is locked by a process, no other process can lock or write to that record. The UNLOCK record function removes the locked entry from the system Lock list and the record is accessable to another process waiting in ready list. The number of records a process can lock is restricted, due to the length of Lock list.

The TEST-AND-WRITE function performs its verification of the user's current version of record with the version on the disk before

allowing the write operation to proceed. This verification occurs at the I/O level in a single operation.

3.4 Synchronization in SDD-1: DDBMS

SDD-1 is a prototype distributed database system that has been developed by the Computer Corporation of America. Reliable Network (RelNet), which is a subsystem of the SDD-1 system, is a communication medium incorporating facilities for site status monitoring, event timestamping, multiple-buffered message delivery, and the atomic control of distributed transactions.

The RelNet consists of a set of facilities intended to ensure communication and coordination among related processes operating at sites connected by means of a communication network. In a distributed system, a number of processes executing in parallel at distinct sites of a network will find occasion to communicate and synchronize with each other. On a practical basis, such a system may fail at any point in time, so each site must be prepared to recognize and react tc the failures of its failing nodes. However, instead of embedding this responsibilty in the application logic and code of each node, the design of RelNet provides each process running in the system with a set of facilities for reliable communication and interaction with other processes. These facilities are utilized by invoking a set of procedure calls. Thus RelNet can be effectively thought of as a virtual network. The facilities used by the system are:

1) There exists within the network a single Global Clock that can be accessed from any site. The clock provides an uniform and consistent ordering of events occuring at different sites in the system. User

processes can inspect the current value of the clock.

2) Every site in the network is at any time in one of the two states, UP or DOWN. The UP state characterizes correct operation and is able to deliver timely responses to messages sent by other sites. The DOWN state characterizes a site not operating at all. Transitions between these states are called Crash/Recovery and occur instantaneously with respect to the Global Clock.

3) Two guarantees are provided for reliable communication service. First, messages sent from one site to another are received in the same order that they are sent. Second, on user request, a message can be marked for guaranteed delivery such that a message sent to a DOWN site will be received by that site upon its recovery.

4) A distributed transaction control is provided so a process running at one site can coordinate the activities of a number of "companion" processes that are running at different sites. The principal feature of this facility is its global abort/commit capability.

The RelNet is organized in a series of software layers, based on the above-mentioned four facilities. Each layer provides a subset of the facilities as a whole. The lowest level of the RelNet is known as the Global Time Layer which provides the global clock and site status features described above. For the purpose of this paper only this layer is discussed since it provides mechanisms to coordinate and synchronize actions being performed at different sites in the network.

The structure of Global Time Layer is as shown in figure 5 [Bernstein, 1980].

Layer	Functionality
Local Clock	Maintains logical clock
Local Status	Manages site status information
Global Clock	Simulates global clock by maintaining consistency among local clocks
Global Status	Provides site failures and recoveries status

Figure 5. Structure of SDD-1 Global Time Layer

The central concept in the Global Time Layer is that of a global clock which is a mechanism used to achieve an ordering of events occurring in the system. The global clock is used to order events by associating with each event the value of the clock at the time of its occurrence which is a timestamp.

The essential property of a global clock is that it be consistent with intersite event ordering. This means that if an event occurs at site X at time t1, then an event that occurs at site Y after Y learns of the A event must occur at time t2, where t2 > t1. Since each site operates according to the timestamps issued by its own local clock, there is no 'global clock'. Therefore, the Global Clock Layer simulates a global clock by means of a collection of independently running local clocks.

The Global Clock Layer uses the mechanisms of guardians and TIMESIGNAL messages to construct a true global clock out of a collection of local clocks. For each site, a set of "guardians" is maintained, each of whose clocks is guaranteed to be at most a constant value less than the value of the guarded site's clock; this is achieved by means of special messages called TIMESIGNALs sent among these sites.

Thus when the failure of site X is detected by site Y, then Y will be informed by one of X's guardians of an upper bound on X's clock at the time it failed; now, Y can bump its own clock to be greater than this value. This method attains the event ordering demanded by the system.

CHAPTER IV

4.0 Evaluation

This Chapter describes how the implementation of synchronization in commercial systems relates to the theoretical work presented in Chapter II. The chapter concludes with some discussion on new computer architecture trend which could provide concurrency.

Looking back over the last 40 of years progress, it seems that the theme for concurrency problem solving has been resource management. One of the best strategies for solving concurrency problem is not to share anything ! Sharing means that at some point in time some process has to be excluded from what is being shared. Systems, without the need for sharing, do not require synchronization. The basic hardware for interrupt mechanisms and sequencing of a set of instructions has not changed over a long period. This indicates that present hardware mechanisms are adequate for satisfying concurrency needs of users and the language constructs of semaphores and queue management have provided added value to enhance virtual concurrency in products. The commercial products studied in this report substantiate this claim.

4.1 Comparison

Concurrent CP/M-86 uses a priority scheme for processes to utilize CPU resources. With priority dispatching, control is never passed to a lower-priority process if there is a higher-priority process on the ready list. Since compute-bound processes have higher priority, they tend to monopolize the CPU resource, so it is necessary to reduce the priority of compute-bound processes to avoid degrading overall

system performance.

There are similarities between the semaphore operations that have been described earlier and the kinds of operations that the queue management facility of Concurrent CP/M-86 provides. For an example, the variation on the queue data structure in CP/M-86 called a Mutual exclusion queue which contains a message of zero length. A system read or write to this queue results into the exactly the same thing as a call to a WAIT on semaphore or SIGNAL to a process. Here the language syntax is different. Instead of a semaphore being set or reset, the difference is whether a queue message is available or not. The actual message is not important; but what is important is the condition under which an operating system suspends or resumes a process. The message queue mechanism can also be used to implement a counting semaphore by defining a queue that can hold 'n' short messages. Commercial systems have exploited this idea using various names of data structure of memory that can be manipulated.

The UNIX operating system provides file structures called pipes similar to the queue management of CP/M-86, except it is more like the mailboxes described in Chapter II. Data from any program or device can easily be connected, without programming, to any other program or device through the use of intermediate disk files (pipes) to provide concurrent operation of I/O tasks.

For the purpose of comparing synchronization mechanisms used by products reviewed in Chapter III, a comparison chart is shown in figure 6.

	Products			
Synchronization Mechanisms	NCR MODUS	UNIX	CP/M-86	SDD-1
Semaphores	x		x	
Message Passing		x	x	x
Timestamps				x

Figure 6. Major Synchronizing Mechanisms In Commercial Products.

A little explanation is needed for the chart in figure 6. The chart indicates highest level mechanisms available in each products in the implementation of synchronization. In the NCR product, although a FIFO structure is used in message passing mode, it is primarily used as a semaphore for task switching by making the FIFO's message length equal to zero. On the other hand, UNIX and CP/M-86, using PIPES and Queues, not only simulate semaphores but create concurrent processes for users to execute jobs. This difference may exist because the operating system products are general purpose systems and the MODUS product is a utility product used by various host systems and personal computers.

It is hard to compare SDD-1 mechanisms for synchronization without detailed implementation information about it, but a geographically distributed system must provide some means of message passing that includes a time history to synchronize processes at the local level by means of other simple mechanisms.

4.2 Limitation

It is beyond the scope of this paper to study in a comparative way all the synchronization primitives introduced to solve particular problems of synchronization. There are two aspects of concurrency and synchronization in terms of implementation have not been discussed which could provide a basis for further research. First is the area of deadlocks and second is the area of the overhead increased when managing information, i.e., who is doing what and when.

A deadlock is a consequence of two or more processes each of which is holding onto resources required by the other. Neither can continue, since each is blocked indefinitely from a request to acquire resources held by the other process and neither is willing to relinquish control until the request for additional resources is granted. The area of deadlocks is a large subject and numerous papers are available [Bernstein et al.,1980],and [Isloor and Marsland, 1980]. There are three categories of algorithms which have been proposed for deadlock management schemes. 1) Detection, in which deadlocks are discovered and remedied by some pre-emption policy. 2) Avoidance, in which processes declare in advance their anticipated resource requirements and the system only grants possible requests. 3) Prevention, in which deadlocks are prevented by pre-analyzing possible deadlocks.

In present commercial products, deadlock situations are resolved by a pre-emption policy, by which conditions and states of processes prior to pre-emption are saved and restored. This is very convenient and easy to do by using available low level mechanisms such as registers and memory.

The problem in a DBMS related to deadlocks is not only avoiding them but also protecting the database from inconsistencies. What this means is that after detecting a deadlock between two transactions, proper care must be taken during rollback of transactions so that data inconsistency does not result. No single policy to handle deadlock problems is suitable for all database systems and much work remains to be done.

The second problem area related to synchronization is managing information about processes and using them without affecting the performance of systems. What this means is that processes that handle synchronization need to be protected from other processes, and the synchronizing processes have to be short in order to prevent them from using too much CPU time. Maybe this is why most of the products examined have not implemented synchronization mechanisms dealing with elaborate programming techniques such as path expressions mentioned in Chapter II.

The overhead problem in database systems is even bigger since keeping track of processes and the states of various nodes requires well-run communication system.

Future research could be directed towards deadlock conditions on synchronization mechanisms.

4.3 Improvements

There are two areas of the computing environment that can have an impact on improving concurrency and synchronization problems. The first is the area of hardware technology, and the second is the area of programming techniques.

The architecture of conventional computers have two major drawbacks that inhibit concurrent operations. First, a processor unit is separated from its data by long communication paths. These paths are long enough to substantially slow down the operations. Second, the Von Neumann architecture, which is used extensively, provides inherently sequential execution of operations.

The VLSI process in semiconductor technology offers an opportunity to overcome the drawbacks of the conventional architecture by providing processing structure and memory structure in close proximity [Mead and Conway, 1980]. The VLSI technology eliminates bus structures consisting of wires and printed-circuit boards and therefore the speed of execution can be improved by several magnitudes. What silicon devices can offer is a process-oriented architecture that can keep all needed information in a processor's private resource; thereby, process-switching time can be improved and processes can be protected from interfering with each other. Also the VLSI technology can provide multi-processors in a more compact form to design low cost systems with dedicated scheduling-processors.

The second improvement area is programming. Presently, a variety of concurrent programming languages are available such as Concurrent Pascal and Ada. The use of programming languages which provide abstract data structures is a way to represent concurrent operations and to provide machine independent programs. Abstraction can hide irrelevent differences of different computer hardware.

The other possibility is to combine various constructs of languages, such as monitors, to provide a more powerful representation of concurrency.

The major problem in concurrent programming is a lack of formal techniques to aid programmers in constructing reliable concurrent programs and much concurrent programming language development work remains.

CHAPTER V

5.0 Summary And Conclusion

One goal of an organization normally is to use its computer systems in a cost-effective way. The notion of sharing resources partially fulfills such a goal. Systems of the early 1960's provided multi-tasking operation by sharing CPU cycles on jobs that were submitted in a batch mode. However,this approach did not solve the problem of I/O bottle-necking. To prevent bottle-necking at the I/O device level, the concept of I/O processors operating in an interrupt mode was developed. Since 1970, users of computing systems have been looking for ways to share resources to gain economy of scale in information systems. This required distributing the work load of the system operations by sharing computers and storages.

None of these were possible without the notion of concurrent programming which provided mechanisms such as semaphores and monitors for synchronizing operations, resources, and data.

However, looking at the present commercial systems that provide resource sharing, it seems that hardware capabilties to support synchronization and concurrency are still very similar to those introduced in the 1960's. For example, the interrupts and sequential instruction execution design ideas in a personal computer such as the IBM PC are similar to those found in the large operating system such as the IBM 360. What has been witnessed is progress. For instance, a job completion that previously was received at some "window" of a computer room is now readily available at the users' desk. We have also witnessed progress in programming languages to provide abstract ideas

such as monitors and classes for mutual exclusion. Many of the microcomputer operating systems, such as CP/86 and UNIX make this possible by providing similar features to handle process synchronization with similar data structures for communication among processes.

Sharing of physical resources such as printers or disk drives can only be realized as a sequential availability, that is only one process or one user can get use of them at any given time. What has been achieved is the virtual availability of these resources due to their operational speed difference and selective needs in applications.

The selective needs are based on information or data which reside in systems. It is the sharing of data which has been greatest interest to users. The ability to widely share data has promulgated the design activity of database systems. Unlike physical resources, data can be duplicated and moved around very easily; this provides an incentive for networking. Much of the synchronization and concurrency mechanisms of database systems are similar to those in operating systems; that is transactions are handled the same way processes are managed in an operating system with respect to synchronization and sharing. Again, the database system has to rely on the primitive mechanisms of hardware that have had a very little growth in capabilities which facilitate handling information simultaneously. Faster-operating peripherals and semiconductor devices have provided virtual concurrency.

Problems of concurrency have been resolved by commercial systems in some justifiable economic sense. What this means is that synchronization and concurrency in systems are kept at a minimum level for the usefulness they provide. Sharing resources using products of multiple vendors is difficult without some type of standardization on

interaction of these products. Some vendors have provided across-the-board compatibility by building a user-level shell which hides the details of interaction. Examples include the UNIX operating system and the NCR MODUS which respectively provide multi-user environment and different personal computer networking capabilty.

The problems of real-time applications are much harder to solve because the concurrency-handling mechanism must respond to variety of nondeterministic requests from its environment. These problems will not be solved until we see some change in present hardware architectures and some new programming techniques.

BIBLIOGRAPHY

[1] Andrews, Gregory R. and Schneider, Fred B., Concept and Notations for Concurrent Programming. ACM Computing Surveys. 15,1(March 1983), 3-43.

[2] Bernstein, P.A.and Goodman, N., Concurrency Control in Distributed Database Systems. ACM computing Surveys. 13, 2(June 1981), 185-222.

[3] Bernstein, P.A., Rothnie, J.B., and Shipman, D.W. "Concurrency Control In A System for Distributed Databases(SDD-1)". ACM Trans. Database Systems.5, 1(March 1980),pp.18-51.

[4] Bernstein, P.A.and Goodman, N. "Approaches TO Concurrency Control In Distributed Database Systems".Proc.AFIPS 1981 NCC,Vol.50,AFIPS Press, Arlington,Va.,pp.813-820.

[5] Bray, Olin H., Distributed Database Management Systems, Lexington Books, Lexington, Mass., 1982.

[6] Brinch Hansen, P., Operating System Principles, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[7] Brinch Hansen, P., The Architecture of Concurrent Programms, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[8] Brinch Hansen, P., A Keynote Address on Concurrent Programming, IEEE Computer Software & Application Conference, Chicago, Illinois, November 1978.

[9] Bunt, R.B., Scheduling Techniques for Operating Systems, Computer, Vol. 9, No. 10, October 1976, pp. 10-17.

[10] Campbell, R. H., and Habermann, A.N. "The Specification of Process Synchronization by Path Expressions." Lecture Notes in Computer Science, vol.16. Springer-Verlag, New YORK, 1974, pp.89-102.

[11] Decitre, Paul."A Concurrency Control Algorithm In A Distributed Environment".Proc.AFIPS 1981 NCC,Vol.50,AFIPS Press, Arlington,Va.,pp.437-479.

[12] Freeman, D.E. and Perry, O.R., I/O Design: Data Management in Operating Systems, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1977.

[13] Habermann, A.N., Introduction to Operating System Design Science Research Associates, Inc., Chicago, Illinois, 1976.

[14] Hammer, Michael, Shipman, David. Reliability Mechanisms for SDD-1; A System for Distributed Database. ACM Transaction on Database System, Vol. 5, No. 4, December 1980, pp. 431-466.

[15] Holt, R.C., Graham, G.S., Lazowska, E.D., and Scott, M.A. Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley Publishing Company, Reading, Massachusetts, 1978.

[16] Isloor, Sreekaanth S. and Marsland, T.A., The Deadlock Problem: An Overview, Computer, Vol. 13, No. 9, September 1980, pp. 58-78.

[17] Kohler, Walter H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. ACM computing Surveys. 13,2(June 1981), 149-184.

[18] Mead, Carver and Conway, Lynn. Introduction to VLSI Systems, Addison-Wesley Publishing Company, Reading, Massachusetts, 1980

[19] Saito, Nobuo, Synchronization Mechanisms for Parallel Processing. Lecture Notes in Computer Science, Springer-Verlag, New York, 1980,pp. 2-22.

[20] Siewiorek, D.P., Bell, C. Gordon and Newell, Allen. Computer Structures: Principles and Examples, McGraw-Hill Book Company, New York, 1982.

[21] Theaker, Colin J., Brookes, Graham R., A Practical Course on Operating Systems, Springer-Verlag, New York, 1983, pp. 170-177.

[22] UNIX Programmer's Manual, Volume II, Bell Telephone Laboratories, Inc., CBS College Publishing, New York, N.Y., 1983.

[23] _____,MODUS Publication, NCR Publication Control, Sugar Camp, Dayton, Ohio. 1984.

CONCURRENCY AND SYNCHRONIZATION ISSUES IN SHARED INFORMATION SYSTEMS

by

MADHU C. PATEL

B.S.M.E., University of Illinois, 1965

AN ABSTRACT OF A MASTER'S REPORT

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

ABSTRACT

This study was undertaken as an interest in studying how concurrency and synchronization problems are viewed in computing systems. This paper discusses the concept of concurrency and synchronization in general and reviews the introduction of concepts throughout the growth of computing systems. Concepts of synchronization in operating systems and Data Base Management Systems (DBMS) are explored through a review of specific mechanisms that are used in implementing the concepts. A review of synchronization and sharing concepts that have been applied by some commercial products such as NCR 6600, a file server; two micro-computer operating systems, Concurrent CP/M-86 and UNIX; and SDD-1, a distributed database system has been done.