OKLAHOMA STATE UNIVERSITY

A GENERAL MUTUAL EXCLUSION PRIMITIVE

BY

RAVEENDRA REDDY AVUTU

Bachelor of Engineering

University of Allahabad

Allahabad, India

1984

Submitted to the Faculty of the
Graduate College of the
Oklahoma State university
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
December 1993

OKLAHOMA STATE UNIVERSITY

A GENERAL MUTUAL EXCLUSION PRIMITIVE

Thesis Approved:

M. Samadzadeh-H.
Thesis Advisor

Donald D. Fisher

Blayne E. Mayfield

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my appreciation to and thank my graduate advisor Dr. Mansur H. Samadzadeh for his advisement, guidance, dedication, encouragement, and instruction throughout my thesis research work. I got inspiration and motivation due to his constant guidance. Without his support, motivation, and patience it would not have been possible to complete this work as it is now.

My sincere thanks to Drs. B.E. Mayfield and D.P. Benjamin for serving on my graduate committee.

Finally, I also wish to thank my wife and brother-in-law. Without their support and encouragement, it would not have been possible for me to complete my graduate studies.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 3

2.1 Control Problems in Parallel Processing 3
2.1.1 Criterion of Mutual Awareness 4
2.1.2 Criterion of Mutual Influence 5
2.1.3 Competition Among n Processes for One Resource 6
2.1.4 Competition Among n Processes for m Resources 7
2.2 Mutual Exclusion: Software Solutions 7
2.2.1 Dekker's Algorithm 8
2.2.2 Dijkstra's Algorithm 8
2.2.3 Knuth's Algorithm 9
2.2.4 De Bruijn's Algorithm 10
2.2.5 Eisenberg and MacGuire's Algorithm 11
2.2.6 Peterson's Algorithm 12
2.3 Mutual Exclusion: Hardware Solutions 12
2.4 Recent Developments in the Field of Synchronization 16
2.4.1 Data Synchronization Instructions 16
2.4.2 Distributed Synchronizers 18

III. IMPLEMENTATION ISSUES 21

3.1 Specification of the Approach 21
3.2 Implementation 25
3.3 Comparison with Other Primitives 26

IV. EVALUATION 33

4.1 The Execution Environment and Its Validation 33
4.2 A Semaphore Implementation of the New Primitive 38
4.3 Study of the Data Collected 40
4.4 Observations 41

V. SUMMARY, CONCLUSIONS, AND FUTURE WORK 43

5.1 Summary 43
5.2 Conclusions 43
5.3 Future Work 44

REFERENCES 46

APPENDICES 49

APPENDIX A- Glossary and Trademark Information 50

APPENDIX B- Program Listings 52

LIST OF FIGURES

Figure Page

1. Various Queue Sizes Resulting from a Sample Run 27

2. Cumulative Average Queue Sizes 28

LIST OF TABLES

Table Page

1. Comparison of n-Process Mutual Exclusion Algorithms 31

2. Results Obtained when Two Processes Updated a Shared Variable 35

3. Results Obtained when Sixty Processes Updated a Shared Variable 36

CHAPTER I

INTRODUCTION

Processes are concurrent if they exist at the same time. Concurrent processes can function completely independently of one another, or they may require occasional synchronization and cooperation.

As computer hardware continues to decrease in size and cost, there is a trend towards multiprocessing and parallelism. If certain operations can logically be performed in parallel, computers should be able to physically perform them in parallel in terms of concurrent activities.

An essential characteristic of the processes that make up a reasonably complex application is that they can be executed in parallel, i.e., simultaneously. For reasons concerned with the hardware on which the application is executed and/or the application itself, it may be necessary to control the execution of each of the processes with respect to the others. The set of rules and mechanisms that enable such control is generally called synchronization. The need for this control is due to two different considerations regarding processes: competition and coordination.

More specifically, depending upon the degree of awareness that the processes have of each other, the relationships among them can be classified as competition, cooperation by sharing, or cooperation by communication. In competition, processes are totally

unaware of the existence of other processes. If processes know that there are other processes when interacting with one another (without being explicitly aware of them), the relationship is known as cooperation by sharing. In cooperation by communication, every process has an environment that contains the names of other processes with which it may explicitly exchange data.

A new synchronization and mutual exclusion primitive is proposed, which is general enough to be applied to both parallel as well as distributed systems. This new primitive is implemented on the Sequent Symmetry S/81 machine. This primitive is compared with other available mutual exclusion primitives and a pilot validation for the proposed primitive is performed.

Chapter II of this thesis gives a literature survey with a brief discussion about control problems in parallel processing and the different relationships that may exist among processes. It also deals with some of the solutions to the mutual exclusion problem that have been implemented either in software or in hardware. Chapter II also discusses some of the recent developments in the area of synchronization. The detailed specifications together with the implementation issues of the new synchronization and mutual exclusion primitive are discussed at length in Chapter III. Evaluation of the new primitive is included in Chapter IV. Chapter V contains the summary and some possible areas of future work.

CHAPTER II

LITERATURE REVIEW

This chapter briefly describes some of the control problems that exist in parallel processing and the proposed software and hardware solutions to the mutual exclusion problem. It also discusses the recent developments in the area of synchronization. Definitions of the frequently used terms are given in Glossary A. Some of the more frequently used terms are:

Accesses: A heterogenous set of processes which are trying to access shared data and which require synchronization.

Synchronizing Agent: A virtual processor that enforces mutual exclusion and synchronization among accesses.

Synchronization System: A system consisting of queues and synchronizing agents.

2.1 Control Problems in Parallel Processing

Interactions among concurrent processes is in general studied based on two criteria [Raynal86]: the degree of awareness that the processes have of each other, and the influence of the behavior of one process on the behavior of others when they interact competitively or cooperatively. According to Raynal, "these criteria [awareness and influence] are independent of the number of processes n (n > 2) and of the level at which

these processes are viewed (applications, systems)" [Raynal86]. The two criteria are described in the following subsections. The last two subsections of this section discuss competition among processes for access to resources.

2.1.1 Criterion of Mutual Awareness

This criterion is concerned with the extent to which a process is aware of the environment (made up of the other processes) with which it interacts. Processes can be classified into two classes based on the degree of awareness that they have of each other: the processes that are unaware of each other, and the processes that are aware of each other [Raynal86]. The second class is broken down into further subclasses: processes indirectly aware of each other (e.g., because they use a shared object), and processes explicitly aware of each other (e.g., because they use communication primitives). These different degrees of awareness among processes have led to the different relationships of competition, cooperation by sharing, and cooperation by communication, which are further explained below.

Competition: In this relationship, processes are completely unaware of the existence of other processes. They come into conflict with other processes for the use of objects, which they should leave in the same state as they found them. This is because, as each of these processes is unaware of the existence of the others, the object (for which the processes come into conflict) must be the same for all of them. The objects involved in such conflicts make up generally what are known as system resources. Raynal states that [Raynal86], "synchronization rules aimed at resolving the problems associated with

physical constraints must be defined so that competition among processes can take place without leading to difficulties".

Cooperation by Sharing: In this relationship, processes know that there are other processes when interacting with each other, without being explicitly aware of them. This is the case with the shared variables among different processes in a system or database. Processes use and update a database (shared data) without referencing other processes, but they are aware that other processes might be using or updating the same data. This relationship deals with the shared data that the processes may transform, rather the resources that have to be allocated. The processes must cooperate in ensuring that the database they share is properly managed [Raynal86].

Cooperation by Communication: In this relationship, every process has an environment that contains the names of other processes with which it may explicitly exchange data. Each process no longer has its own particular aim, rather it participates in a common goal which links the entire set of processes. Here we are dealing with message systems, or systems of communicating processes, which "are characterized by the presence of message transmission and reception primitives" [Raynal86].

2.1.2 Criterion of Mutual Influence

This criterion deals with the influence of the behavior of one process on the behavior of the other processes when they interact competitively or cooperatively. In the case of competition, no exchange of information takes place among processes. Each process has its own code, and consequently the results of one process cannot be affected

by the actions of other processes.

On the other hand, the behavior of one process may be affected by the other processes. If there is competition for a single resource between two processes, one will have to wait for the other to finish before using that resource. Thus, one process has been slowed by another process. It is possible for a process to be denied access to a resource indefinitely, in which case it would never terminate and would never give a result. With cooperative interactions, a process may directly influence the result of another process by means of an exchange of information [Raynal86].

2.1.3 Competition Among n Processes for One Resource

Consider the case of n processes in conflict for accessing a single non-sharable resource. Since the resource can only be used by a single process at a time, it is called a critical resource (which will be used only in a critical section of each process). The scheme is made up of two parts (for acquiring and releasing the resource), which ensure that the resource is used only by one process at a time.

Dijkstra proposed the minimum number of properties that make the algorithms implementing a mutual exclusion scheme operational [Dijkstra65a]. Any new mutual exclusion scheme must allow for these properties. Dijkstra lists these properties as:

a) At any time, no more than one process can be in its critical section.

b) The critical section must be reachable, i.e., when several processes are waiting to enter the critical section and there is no process in the critical section, one of the processes must enter it within a finite amount of time.

c) The behavior of a process outside its critical section must have no influence on the mutual exclusion scheme.

d) There are no privileged processes; the problem is solved in the same way for all of them.

2.1.4 Competition Among n Processes for m Resources

One of the problems that is encountered in a model, where n processes are in competition for the use of m resources, is that of deadlock. The conditions for deadlock can be summarized as follows [Maekawa87].

a) Each process has exclusive use of each resource once the resource has been allocated to the process (this is called mutual exclusion).

b) A process may hold one resource while it requests another one.

c) A situation can arise in which process p_1 requests and acquires resource R_1 and then it requests resource R_2, while process p_2 requests and acquires resource R_2 and then requests resource R_1 (circular wait).

d) Resources can only be released by the explicit action of the processes, that is, resources cannot be preempted.

2.2 Mutual Exclusion: Software Solutions

In the subsections that follow, different software solutions for the mutual exclusion problem are presented. The following six subsections broadly describes the evolution of software solutions to the problem of mutual exclusion using only two processes in most

cases.

2.2.1 Dekker's Algorithm

In 1965, the famous Dutch mathematician T. Dekker proposed the first correct software solution for the mutual exclusion problem. The solution given by him follows [Dijkstra65a].

The two processes P_0 and P_1 share the following variables:

```
var flag: array[0 .. 1] of boolean;
    turn: 0 .. 1;
```

The variable `flag` is initialized to false and `turn` has the value of 0 or 1. Each process P_i has an integer variable j. The scheme for P_i, i = 0 and 1, is:

```
flag[i] ← true;
while flag[i] do if turn = j then
     begin
       flag[i] ← false;
       while turn = j do nothing enddo;
       flag[i] ← true;
     end;
   endif;
enddo;
<critical section>
turn ← j;
flag[i] ← false;
```

Dekker's algorithm resolves the conflicts within a finite time, both from the point of view of the critical section and from the point of view of the processes.

2.2.2 Dijkstra's Algorithm

Dijkstra generalized Dekker's solution to the case of n processes [Dijkstra65b]. The variables shared among the n processes P_0, P_1, ..., P_{n-1} are:

```
var flag: array[0 .. n-1] of (passive,requesting, in_cs);
    turn: 0 .. n-1;
```

The elements of `flag` are initialized to passive and `turn` takes some arbitrary value between 0 and n-1. Each process P_i has an integer variable j.

```
repeat
     flag[i] ← requesting;
          while turn != i do if flag[turn] = passive
                                then turn ← i
                             endif;
          enddo;
     flag[i] ← in_cs;
     j ← 0;
     while (j < n) ∧ (j = i ∨ flag[j] != in_cs)
          do j ← j + 1
     enddo;
until j ≥ n;
<critical section>
flag[i] ← passive;
```

This solution guarantees mutual exclusion and avoids deadlock. However, it does not avoid the risk of starvation. If a number of processes are constantly trying to access the critical section, there is nothing to stop one of these processes from always being the last to modify `turn` when competing for the modification of this variable.

2.2.3 Knuth's Algorithm

Dijkstra's solution guarantees mutual exclusion and the reachability of the critical section. However, it does not avoid the risk of starvation and does not guarantee fairness. Knuth proposed the first fair solution [Knuth66]. In his solution, the processes share the following variables.

```
var flag: array [0 .. n-1] of(passive,requesting,in_cs);
    turn: 0 .. n-1;
```

The variable `flag` is initialized to passive and `turn` is initialized to 0. Each process

has a local variable j in the range 0, 1, ..., n-1. The scheme for process P_i, $0 \le i \le n-1$, is given below.

```
repeat
     flag[i] ← requesting;
     j ← turn;
     while j != i do
               if flag[j] != passive then j ← turn
                              else j ← (j-1) mod n
               endif;
     enddo;
     flag[i] ← in_cs;
until testd(i) ;
turn ← i;
<critical section>;
turn ← (i-1) mod n;
flag[i] ← passive;
```

The function `testd(i)` is implemented iteratively and it is defined as follows.

```
testd(i) = true (∀j != i such that flag[j] != in_cs)
                           ∧ flag[i] = in_cs;
```

This solution guarantees mutual exclusion and is also fair.

2.2.4 De Bruijn's Algorithm

De Bruijn proposed an improvement to Knuth's algorithm, in which the delay function is polynomial rather than exponential [De Bruijn67]. Raynal states that, "in De Bruijn's suggestion, the variable `turn` is modified only once as P_i leaves its critical section, provided that its value is the number of the process leaving the critical section, or that of the process whose turn is not affected by the mutual exclusion" [Raynal86]. The scheme for process P_i, $0 \le i \le n-1$, follows.

```
repeat
     flag[i] ← requesting;
     j ← turn;
     while j != i do
               if flag[j] != passive then j ← turn
```

```
                                      else j ← (j-1) mod n
                endif;
        enddo;
        flag[i] ← in_cs;
until testd(i);
<critical section>;
if flag[turn] = passive ∧ turn = i
   then turn ← (turn-1) mod n
endif;
flag[i] ← passive;
```

2.2.5 Eisenberg and MacGuire's Algorithm

Eisenberg and MacGuire proposed a new solution in which the delay is linear, instead of exponential (as in the case of Knuth's solution) or quadratic (as in the case of De Bruijn's solution) [Eisenberg72]. This solution differs essentially in the postlude, which assigns a value to `turn` that allows a process's delay to be reduced. However, the linear waiting is not FIFO. A process may overtake another one no more than once from the moment its request to enter the critical section has been expressed. The scheme for process P_i, $0 \leq i \leq n-1$, follows.

```
repeat
     flag[i] ← requesting;
     j ← turn;
     while j != i do
                if flag[j] != passive then j ← turn
                                      else j ← (j+1) mod n
                endif;
     enddo;
     flag[i] ← in_cs;
     j ← 0;
     while( j < n) ∧ (j = i or flag[j] != in_cs)
                     do j ← (j+1)
     enddo;
until (j ≤ n) ∧ (turn = i ∨ flag[turn] = passive);
turn ← i;
<critical section>
j ← (turn+1) mod n;
while (j != turn) ∧ (flag[j] = passive) do j←(j+1)mod n
enddo;
turn ← j;
```

```
flag[i] ← passive;
```

2.2.6 Peterson's Algorithm

Peterson gave an elegant and simple solution to the mutual exclusion problem [Peterson81]. Variables shared among the processes are the following.

```
var flag: array[0 .. 1] of boolean;
    turn: 0 .. 1;
```

The variable `flag[i]`, which is initialized to false, indicates the position of P_i with respect to the mutual exclusion, and `turn` resolves simultaneity conflicts. The scheme for process P_i is as follows (i = 0, 1 and j = (i+1) mod 2)

```
flag[i] ← true;
turn ← i;
wait flag[j] = false ∨ turn = j;
<critical section>;
flag[i] ← false;
```

In this solution, **wait** stalls the flow of control (the execution of the next instruction) until its arguments become true.

2.3 Mutual Exclusion: Hardware Solutions

This section briefly describes the solutions to the mutual exclusion problem that have been implemented in hardware. These solutions are specialized instructions that are implemented on many machines. According to Raynal [Raynal86], "the common factor linking all these instructions, and distinguishing them from all others, is the fact that they carry out two actions atomically; e.g., reading and writing, or reading and testing, of a single memory location within one instruction fetch cycle".

Test-and-Set Instruction: The testset(m) instruction executes a series of atomic actions.

It tests the value of variable m. If it is zero, it replaces it by one and returns the result as true; otherwise, it makes no change to the value of m and returns false as a result [Raynal86]. The mutual exclusion scheme implemented using this instruction requires a shared variable *bolt* initialized to 0. The scheme for process P_i, $0 \leq i \leq n-1$, follows.

```
var bolt;
repeat nothing
     until testset(bolt);
<critical section>
bolt ← 0;
```

The only process that can enter its critical section is the one that finds `bolt` set to 0. Once a process P_i is in its critical section, all other processes trying to enter the critical section are delayed in their corresponding entry code. Once P_i executes its exit code, only one of the waiting processes will be able to enter its critical section.

Lock Instruction: The atomic lock and unlock instructions are described below [Shaw74].

```
lock(m)  = begin
           repeat nothing while m = 1;
           m ← 1;
           end;

unlock(m)  = m ← 0;
```

The wait loop in this instruction is an integral part of the instruction, whereas the loop is external in the testset(m) instruction.

Replace-Add Instruction: Instruction repadd(m,v) atomically adds the contents of m to the value v, stores the sum in m, and returns its result [Gottlieb83]. A mutual exclusion scheme can be implemented using variable `bolt` that is initialized to 1 and shared by all processes. Variable ok_i is local to process P_i, $0 \leq i \leq n-1$.

```
ok_i ← false;
repeat if bolt-1 >= 0 then
       if(repadd(bolt, -1) >= 0)
```

```
            then ok_i ← true
            else repadd(bolt, 1);
         endif;
       endif;
until ok_i;
<critical section>
repadd(bolt,1);
```

Semaphore Instruction: Solutions discussed so far have certain disadvantages. First, there is busy or active wait; a process that is doing nothing nonetheless occupies the processor, thus limiting the efficiency of the system kernel, and therefore leads to loss of performance. Secondly, there is the difficulty of generalizing these solutions to more complex problems. Dijkstra originally defined the semaphore concept [Dijkstra65a]. A semaphore s is a non-negative integer variable that can be handled only by the following two primitive operations (in addition to initialization).

```
P(s) :  if s > 0 then s ← s - 1;
        else wait on s;

V(s) :  if one or more processes are waiting on s
        then let one of these processes proceed;
        else s ← s + 1;
```

The mutual exclusion scheme can be coded using a semaphore mutex (initialized to 1) as follows.

```
P(mutex);
        <critical section>
V(mutex);
```

There are many extensions to the above basic definition and implementation of the semaphores, to suit various mutual exclusion and synchronization requirements. A couple of the more complex extensions of the basic semaphore definition are given below

[Presser75]. In the pseudocodes appearing below, E represents an event variable and $|L_E|$ represents the number of entries in its waiting list.

A solution based on arrays of event variables and parametrized test values: Let $1 \leq i \leq k$ and $m_i \geq 0$ then:

$P(E_1, m_1; ... ; E_i, m_i; ... ; E_k, m_k)$
  if for all i we have $E_i \geq m_i$
  then for all i we do $E_i \leftarrow E_i - m_i$
    and the process issuing the P continues its progress
  else an entry of the form (name of process issuing P, address of P) is placed in the waiting list of the first E_i found such that $E_i < m_i$, and the process issuing the P enters the blocked state

$V(E_1, m_1; ... ; E_i, m_i; ... ; E_k, m_k)$
  For all i we do $E_i \leftarrow E_i + m_i$
  if there exists a value of i such that $|L_{Ei}| \geq 1$
  then for each i such that $|L_{Ei}| \geq 1$, remove all the entries from the associated waiting list and change the status of the corresponding processes to the ready state; the process issuing the V is placed in the ready state
  else the process issuing V continues its progress

Solution for mutual exclusion between processes prioritized by levels: Let $1 \leq i \leq k$, $t_i \geq 0$, and $\delta_i \geq 0$ then:

$P(E_1, t_1, \delta_1; ... ; E_i, t_i, \delta_i; ... ; E_k, t_k, \delta_k)$
  if for all i we have $E_i \geq t_i$
  then for all i we do $E_i \leftarrow E_i - \delta_i$
    and the process issuing the P continues its progress.
  else an entry of the form (name of process issuing P, address of P) is placed in the waiting list of the first E_i found such that $E_i < t_i$, and the process issuing the P enters the blocked state

$V(E_1, \delta_1; ... ; E_i, \delta_i; ... ; E_k, \delta_k)$
  For all i we do $E_i \leftarrow E_i + \delta_i$
  if there exists a value of i such that $|L_{Ei}| \geq 1$
  then for each i such that $|L_{Ei}| \geq 1$, remove all the entries from the associated waiting list and change the status of the corresponding processes to the ready state; the process issuing V is placed in the ready state

else the process issuing V continues its progress

The solutions presented above can lead to the indefinite postponement of a low priority process by two higher priority processes. However, the definition of the problem (solution based on prioritized levels) states that the higher priority process should be given preference over low priority process, without concern for the waiting time of any lower priority process. Hence, no assumption about the queueing discipline in the definitions of the P/V primitives was made.

2.4 Recent Developments in the Field of Synchronization

The following two subsections discuss briefly the recent developments in the area of synchronization of processes.

2.4.1 Data Synchronization Instructions

A program can be decomposed into as many concurrent tasks as possible in order to attain speed-up in a multiprocessor system. As stated by Tang et. al, "while tasks are executed concurrently on the multiple processors, the order of access to each particular data item in the original program must be preserved to guarantee the correctness of the execution" [Tang90]. This access order is called "data dependence". Three types of data dependences have been identified: 1) flow dependence, which indicates a write-after-read access order, 2) anti-dependence, which indicates a read-after-write access order, and 3) output dependence, which indicates a write-after-write access order. Data dependence can be enforced either by using implicit programming structures or by using explicit

synchronization instructions [Tang90].

When a program is decomposed into a set of concurrent tasks, there is a precedence relation associated with these tasks. The precedence relation can be expressed by parallel program constructs such as a **doall** loop or a **barrier synchronization** [Tang90]. **Doall** loops are parallel loops which do not have cross-iteration data dependencies. A barrier synchronization represents the point in a program where a set of tasks must all be completed before the next set of tasks can be started. Since the type of enforcement on data dependencies is implied in the precedence relation of the tasks, this synchronization is called implicit data synchronization.

Data dependences can be enforced by synchronizing memory accesses directly. All tasks can be concurrently executed, as long as the order of memory accesses are preserved. Stated differently, the precedence relation can be relaxed among the tasks if direct synchronization is utilized to enforce data dependence [Tang90]. This scheme allows for potentially more tasks to be executed in parallel and thus can achieve a potentially higher speed-up for programs. This type of enforcement is called explicit data synchronization.

Explicit synchronization instructions can be divided into two classes. One category of such instructions is "used to synchronize groups of memory accesses" [Tang90]. These instructions typically synchronize at the statement level and are referred to as statement-level synchronization instructions. The second category of explicit synchronization instructions "synchronize accesses at each individual data element" [Tang90]. These instructions are typically incorporated in the reading or writing of each individual data

element, and hence are referred to as data-level synchronization instructions.

The implementation of data-level synchronization instructions can be outlined as follows [Tang90]. An integer key is associated with each data element that needs data synchronization. This key has the access order (ordering information) of the data element and is initialized to 0. Each data element has to be read or written according to the order imposed in the original program. Before access to a data element by a processor can be granted, the processor "has to check its ordering number against the value of the key to see if it is its turn to access the data element" [Tang90]. The processor increments the value of the key associated with the data element after each access, to update the ordering information, so that the next processor can access that data element.

2.4.2 Distributed Synchronizers

Doddaballapur proposed a new scheme, called a distributed synchronizer, which is based on the notion of partially shared variables [Doddaballapur88]. This primitive suits the synchronization requirements of parallel algorithms executing on large, shared-memory multiprocessors. All synchronization variables are arranged in the form of a n-ary tree, where n is the number of processing elements, which is called a synchronization tree. According to Doddaballapur, "an efficient implementation of the distributed synchronizer scheme requires: a) the embedding of the synchronization tree in the processor-memory multistage interconnection network, and b) simple hardware enhancements at the switching elements of the network" [Doddaballapur88].

Rather than forcing all the n processing elements (PEs) to access a single variable,

a fixed number m of the PEs, 2 <= m << n, could be allowed to access each variable (assuming n to be a power of m). By using such a scheme, the number of synchronization variables required increases, but it might reduce memory contention. By this scheme, synchronization variables could be arranged in the form of a synchronization tree. All the variables in the tree are initialized to m. A PE proceeds to the next higher level of the tree after decrementing the variable, only if it finds the value of the variable to be zero (after decrementing), otherwise the PE waits for the variable to assume a special value of -1. This scheme is called a **walk-in** scheme. The last arriving PE decrements the root to zero and sets it to -1. At this time, walk-in ends. In this process, when a PE is waiting on a variable whose value is -1, the PE passes this information to the next lower level of the tree. This procedure is repeated recursively and is called a **walk-out** scheme. The completion time would be the time at which the last PE at the lowest level finds its variable to be -1. At this time the walk-out ends.

The advantage of the walk-in/walk-out scheme is that it requires only a fixed number m of PEs to access a node of the synchronization tree. Furthermore, a node at level i needs to be shared only by m^i PEs (some m PEs out of these m^i PEs access the node). According to Doddaballapur, "this observation suggests that the nodes of the synchronization tree may be placed in a memory hierarchy according to the degree to which the nodes are shared" [Doddaballapur88]. A memory hierarchy being a set of partially-shared memories such that a variable at level i is shared by more PEs than a variable at level j, j < i. The memory hierarchy can be used to store the variables at appropriate levels. Doing so will eliminate "expensive global memory trips to access

shared variables that need to be only partially shared" [Doddaballapur88]. The distributed synchronizer scheme utilizes the walk-in/walk-out scheme.

CHAPTER III

IMPLEMENTATION ISSUES

The main focus of this thesis is developing a new synchronization and mutual exclusion primitive which is general enough to be applied to parallel and distributed systems, and implementing this primitive on the Sequent Symmetry S/81 machine with twenty four 80386-20MHz processors each with 64K Cache memory. In its current configuration, the implementation platform has 104 Mega Bytes of RAM and 5 Gega Bytes of total hard disk storage. It has 16 serial ports, 1 ethernet port, and a 9-track 1600/6250 bpi tape drive.

3.1 Specification of the Approach

The new synchronization primitive services the arriving accesses according to their priority. The accesses are classified based on their priority. The synchronization agents are also divided into high and low priority agents which service high and low priority accesses, respectively. Out of a maximum of n synchronization agents, m agents, $m \leq n$ are designated as active synchronization agents initially. The remaining agents are initially inactive. Out of the m active agents, h agents, $1 \leq h \leq m-1$, are devoted to servicing the high priority accesses and the remaining k agents, $1 \leq k \leq n-h$, are allocated to the low priority accesses. A finite sized queue is associated with each agent. Also, two overflow

queues of finite size are provided, one for high priority accesses and the other for low priority accesses.

When a new access enters the synchronization system, depending upon its priority, the agents are checked for availability. If any free agent is available in its category, the access is allocated to that free agent. If none is available, and if the access is of high priority, then the availability of free low priority agents is checked and one allocated if available. Otherwise, the access is allocated to an agent whose waiting queue size is minimum among the agents of its priority class. In case there is no space in the associated queues and the access is of low priority, an inactive agent, if one is available, is made active and the access is allocated to this newly activated agent. Otherwise, the access is made to wait in the overflow queue. The access is allowed to enter into the synchronization system (i.e., to compete for the mutually excluded resources) if and only if there is space in the overflow queue of its priority class. Otherwise, the access is made to wait outside the synchronization system till space becomes available in the appropriate overflow queue.

When an access releases the synchronizing agent and there are other accesses waiting in the queue associated with that particular agent, the access at the head of the queue associated with that particular synchronizing agent is removed and allocated to that synchronizing agent. If there are other accesses in the corresponding overflow queue, the access at the head of the overflow queue is removed and inserted into the queue associated with that synchronizing agent. Once an access is removed from the overflow queue, if any access is waiting outside the synchronization system to enter, that access

is allowed to enter. The number of accesses allowed into the synchronization system at this time depends on space availability. All these actions, i.e., updating the queues associated with agents, updating the overflow queues, and allowing the waiting access(es) into the synchronization system, are done atomically.

If there are no accesses waiting in the queue of an agent, the agent is either put to sleep or made inactive depending on the past history of the arrival rate of the accesses. If the arrival rate is less than an initially defined rate (the rate of arrival is fixed based on system characteristics) and there are more than a minimum number of active agents in the synchronization system, then that particular agent is made inactive. Otherwise, it is put to sleep. The detailed pseudocode of the primitive is given below.

```
DEFINE MAX_QUEUE_SIZE
DEFINE m
MAX_SYNCH_AGENTS = N;
ACTIVE_SYNCH_AGENTS = 0;    /* initialized to 0 */
OVERFLOW_QUEUE : queue
SYNCH_AGENTS_BUSY : Boolean;

procedure : accessing (t:access){
if it is high priority access{
     if there are accesses in high_priority overflow queue
          insert(t, high_priority overflow queue);
     else if high priority synch_agent is free
          allocate t to the high priority synch_agent;
     else if the flag SYNCH_AGENTS_BUSY is false{
          find free_synch_agent out of m ACTIVE SYNCH_AGENTS;
          allocate t to the free_synch_agent;
          increment ACTIVE_SYNCH_AGENTS;
          if ACTIVE_SYNCH_AGENTS == m, then
               set the SYNCH_AGENTS_BUSY to true
          call start_processing(t, free_synch_agent);
     }/* end elseif */
     else if the flag SYNCH_AGENTS_BUSY is true
          wait(t, synch_agent[high_priority]);
}/* end if */
else{
     if there are accesses in low_priority overflow queue
          insert(t, low_priority overflow queue);
     else{
```

```
          if the flag SYNCH_AGENTS_BUSY is false{
               find a free_synch_agent;
               allocate t to the free_synch_agent;
               increment ACTIVE_SYNCH_AGENTS;
               if ACTIVE_SYNCH_AGENTS == m
                    set SYNCH_AGENTS_BUSY to true;
               call start_processing(t, free_synch_agent);
          }/* end if */
          else if the flag SYNCH_AGENTS_BUSY is true{
               find a synch_agent whose wait queue size is min
               if this queue size is equal to MAX_QUEUE_SIZE{
                    if ACTIVE_SYNCH_AGENTS <> n{
                         activate one more synch_agent;
                         increment ACTIVE_SYNCH_AGENT;
                         allocate t to this synch_agent;
                         call start_processing(t,
                                             new_synch_agent);
                    }/* end if */
                    else
                         insert(t, low_priority overflow
                                                  queue);
               }/* end if */
               else
                    wait(t, synch_agent[min_queue_size]);
          }/* end elseif */
     }/* end else */
}/* end else */
}/* end accessing() */

procedure : start_processing(t, pi){
     process t;
     signal(pi);
}

atomic procedure : signal(pi){
     if there are any accesses waiting on pi{
          start_processing(access at the head of queue, pi);
          decrement queue size;
          if there are accesses in overflow queue{
               delete access from head of overflow queue;
               insert that access into queue of pi;
          }/* end if */
     }/* end if */
     else{
          if past history suggests fewer arrivals and
                                   active_synch_agents > m
               deactivate pi from the system;
          else
               put synch_agent pi to sleep;
     }/* end else */
}/* end signal() */
```

```
atomic procedure : wait(t, pi){
     insert(t, queue[pi]);
          increment queue size;
}
```

3.2 Implementation

The new primitive was implemented on a Sequent Symmetry S/81. The various variables used to simulate the primitive such as, the total number of agents MAX_SYNCH_AGENTS (n), the total number of active agents m, the number of high priority agents HPAGENTS (h), the number of low priority agents LPAGENTS (k), the size of the queue associated with each synchronizing agent MAX_QUEUE_SIZE, and the size of the overflow queue OVERFLOWQUEUESIZE are defined at the beginning of the program.

A total of six synchronizing agents were simulated. Two agents out of six were designated as high priority agents and the remaining four agents were designated as low priority agents. A total of four agents were marked as active synchronizing agents, initially (both high priority synchronizing agents were designated as active agents). Queues of size two were associated with each synchronizing agent. The size of the overflow queue was fixed at five.

A random number generator was used to generate numbers between 0 and 5. If 0 is generated, the access was considered a high priority access, otherwise it was considered a low priority access. The range of the random values generated or the criteria used for designating the accesses (i.e., as high priority access or as low priority access) may reflect on the performance of the proposed primitive, but not significantly. A random

number generator was also used for generating the interarrival times and the processing times. A system clock variable was used to simulate the actual clock, i.e., to implement the distinct arrival and departure patterns of the accesses. The system clock variable was incremented by one time unit for every loop iteration. The interarrival times were generated in such a way that they resemble the actual arrival times, i.e., more than one access can arrive at the same time (temporally concurrent accesses).

The primitive and the simulated environment was run continuously for five days, without any interruption. Various aspects of the primitive, such as allocating accesses to their corresponding synchronizing agents, updating various queues correctly, and activating/putting to sleep a synchronizing agent as necessary were tested. The various queue sizes were displayed on the terminal screen during the course of the run. A sample display of the queue sizes is shown in Figure 1. The queue size values were multiplied by 25 and then displayed as a scaling factor for the readability of the figure. A system utility "gnuplot" was also used to display the cumulative average queue sizes. The cumulative average queue sizes of all queues were multiplied by a factor 25 (in order to have a better display) and were written to a file. Then the gnuplot utility takes this datafile as its argument and plots a graph during the course of the run, as shown in Figure 2.

3.3 Comparison with Other Primitives

There are a number of primitives available for the n process mutual exclusion problem. The algorithm proposed by Dijkstra [Dijkstra65b] suffers from unbounded

waiting time for one or more processes. The Algorithm proposed by Knuth [Knuth66] has time with an upper bound of 2^n turns. De Bruijn introduced his algorithm [De Bruijn67] with (1/2)*n*(n-1) turns of waiting time in the worst case. Eisenberg and Macguire [Eisenberg72] reduced the upper bound of waiting time to (n-1) turns. None of these algorithms are easy to understand; in fact they are all rather complicated.

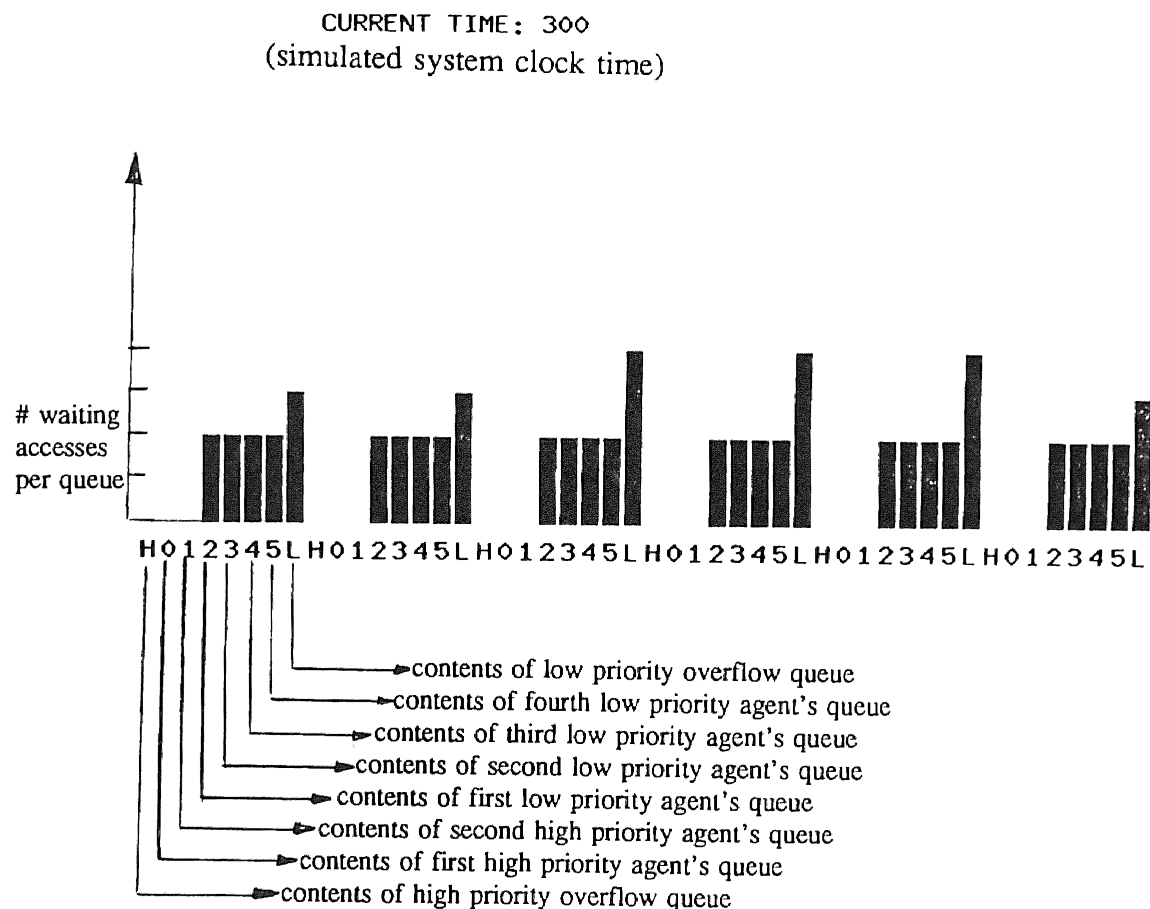

Figure 1. Various Queue Sizes Resulting from a Sample Run

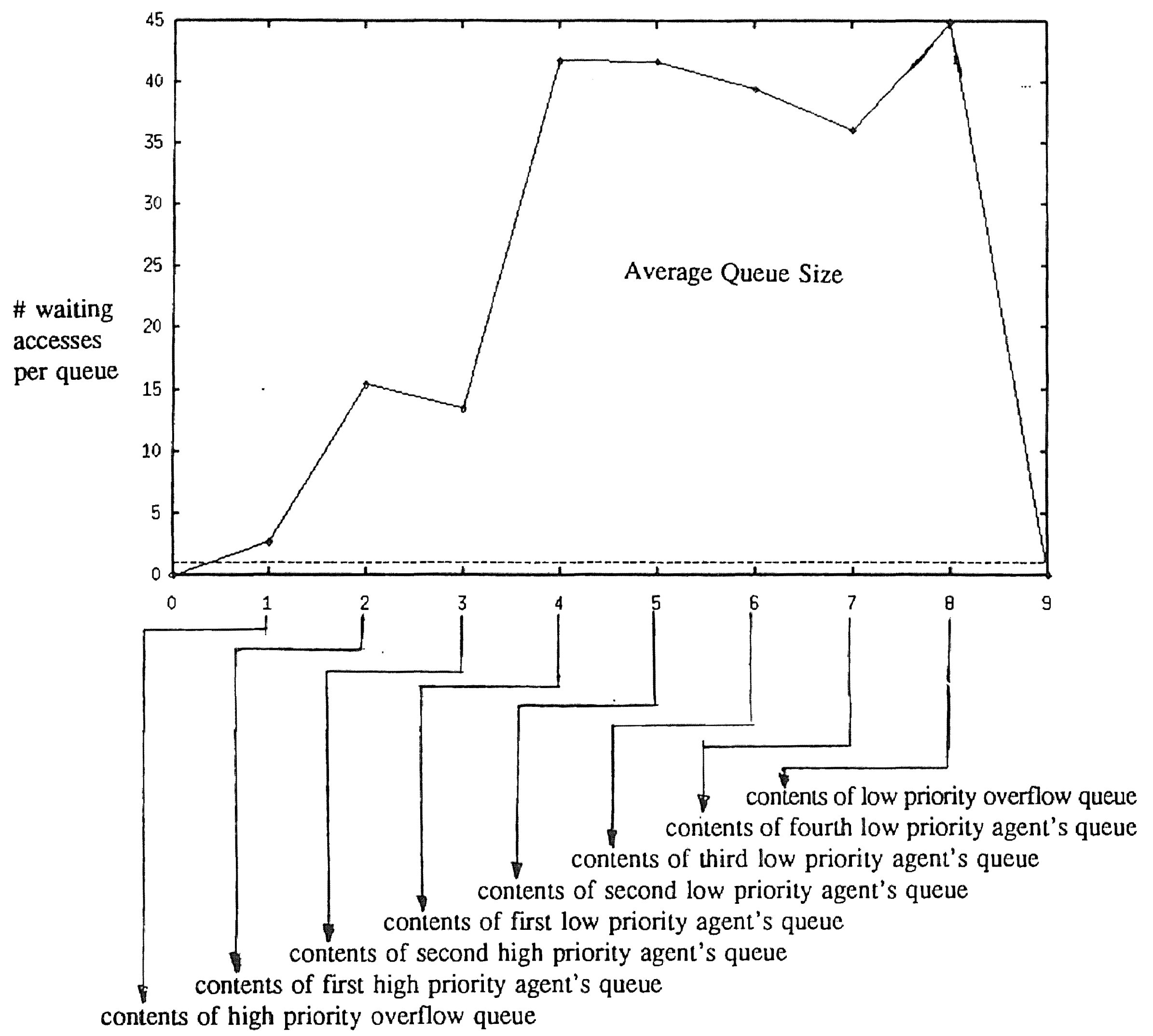

Figure 2. Cumulative Average Queue Sizes

The proposed primitive is logically simple and easy to understand as more complex constructs such as semaphores are not present. In this approach, all active synchronizing agents are checked for the completion of processing. This is done from the first agent (agent[0]) to the last agent (agent[n-1]). All the updates of the queues (the queues associated with the agents and the overflow queues) are done atomically, i.e., at the same time (at the same unit increment of the system clock) and without interrupts. Only after completing the updates of all the queues, is the next operation/execution performed. Removing the access at the head of the queue associated with an agent and allocating that access to the agent is done at the same time for accesses in the queues of all the agents.

The proposed primitive is also fair. It doesn't discriminate against any access. All the accesses in the same priority class are given the same treatment. No access is postponed indefinitely. Every access gets its turn in a finite amount of time. This primitive makes sure that the load is uniform among different active synchronizing agents by making the access to wait on the synchronizing agent whose waiting queue size is minimum.

Since the proposed primitive is intended to solve the problem of mutual exclusion among processes of different priorities, the higher priority processes are given preference. However, there are no privileged processes in the same priority class. This primitive implements the FCFS discipline among the same priority accesses. The implementation using the proposed primitive for n priority classes should be the same as the one discussed for two priority classes. In the general case, there will be n groups (one group

per class of accesses) of synchronizing agents for the n priority classes and the n overflow queues (one per class of accesses), instead of two groups (high priority and low priority) of synchronizing agents and two overflow queues (for two priority class).

It is not difficult to observe that mutual exclusion holds for the n process mutual exclusion and synchronization problem (for m priority classes). An access cannot be allocated to an agent if that agent is already processing another access. Only when the flag of that agent is set to available, can another access be allocated. If all synchronizing agents are busy, the access is made to wait either in the queue associated with one of the agents or in the corresponding overflow queue. Since the checking of the synchronizing agents for the completion of processing is done atomically, and the checking is done from agent[0] to agent[n-1], deadlock is prevented.

Allocating an access to a free synchronizing agent depends only on the priority of the access, arrival time (FCFS basis), and the availability of the synchronizing agent. It does not depend on any other waiting access. As setting a synchronizing agent's flag to available does not depend on any waiting accesses, there is no lock step synchronization among accesses. Also, there is no blocking or indefinite postponement, as no access or set of accesses can monopolize any synchronizing agent.

Table 1 summarizes a comparison of seven n process mutual exclusion algorithms [Zand89]. The eighth algorithm is the one presented here and the remaining are chosen from among the more efficient and understandable algorithms available in the open literature. All the algorithms are compared with respect to the number of predicates used in the algorithm (as an indicator of their conceptual complexity [McCabe76]), the upper

bound for the waiting time in terms of the number of turns (only the best case is represented for the new primitive), and the number of variables used in the algorithm. In Table 1, m is the number of synchronizing agents and n is the number of processes.

TABLE 1

COMPARISON OF n-PROCESS MUTUAL EXCLUSION ALGORITHMS

Algorithm	Predicates	WaitingTime	Variables
Knuth	n+3	$2^{n-1}-1$	n+4
De Bruijn	n+6	n*(n-1)/2	n+4
Eisenberg/MacGuire	11	n-1	n+7
Peterson	3	n*(n-1)/2	2*n+3
Lamport	4	n-1	n+2
KSG	3	n-1	n+3
Habermann	5	2*(n-1)	n+6
New Primitive	5	(n-1)/m (best case)	2*m+5

The overhead of the new primitive is the queues associated with each synchronizing agent and the overflow queues, but the simplicity and the structured design of the primitive justifies this tradeoff. By providing queues (the queues associated with

each synchronizing agent and the overflow queues), we are able to achieve a load balance by having a uniform load among the synchronizing agents in the system.

CHAPTER IV

EVALUATION

This Chapter describes the evaluation of the proposed primitive. The execution environment, its validation, and a semaphore implementation of the new primitive along with some observations are discussed in detail in the following sections.

The implementation platform was the Sequent Symmetry S/81 at the Computer Science Department at Oklahoma State University running DYNIX/ptx operating system (for more details about the implementation platform, see Chapter III).

4.1 The Execution Environment and Its Validation

Since the new mutual exclusion primitive could not be tested in actual practice, it was necessary to set up a simulated execution environment with the variability of an actual heterogeneous environment. The simple problem of simultaneous, unprotected access to a shared variable was chosen as a typical mutual exclusion problem to tune and validate the simulated execution environment.

In the problem of unprotected access to a shared variable, a simple increment operation is performed by each concurrent process accessing the shared variable in a loop. The shared variable x is initialized to 0. Each process has a local variable i. The operation is described below.

```
for(i = 0; i < 10; i++)
    x = x + 1;
```

A number of processes were forked and were run for a number of times. The final values of the shared variable after each run of the above loop, was collected and compared. Two different scenarios were studied.

Scenario 1: In this scenario, two processes were forked and the shared variable was updated 10 times in the loop by each process for a total of 100 runs. Two different cases were generated to capture the variability of arbitrary processes accessing a shared variable. In the first case, process 1 sleeps between the *read* and *write* operations performed on the shared variable, whereas process 2 sleeps before the *read* operation (performed on the shared variable) only. In the second case, the ordering is reversed, i.e., process 1 sleeps before *read* only and process 2 sleeps between *read* and *write*. Ideally, the results (the final value of the shared variable) should vary between 2 and 20. Since it is difficult to generate truly random numbers (to put the process to sleep for a truly random amount of time), the results differ from the theoretical values. Table 2 summarizes the results obtained. It shows the number of times a particular value was obtained over the total number of trails. In this scenario, the waiting of the processes was passive, in the sense that the sleep instruction causes the process executing it to be swapped out (Appendix B contains the code).

TABLE 2

RESULTS OBTAINED WHEN TWO PROCESSES UPDATED A SHARED VARIABLE (100 RUNS)

Value obtained	Frequency	
	Case 1	Case 2
10	46%	17%
11	19%	25%
12	9%	24%
13	10%	12%
14	4%	9%
15	3%	7%
16	4%	1%
17	1%	3%
18	4%	1%
19	0%	0%
20	0%	1%

Scenario 2: In this scenario, 60 processes were forked and the shared variable was updated 10 times in the loop by each process for a total of 1000 runs. Here also two different cases were generated to capture the variability of arbitrary processes accessing a shared variable. In the first case, every process waits between the *read* and *write* operations performed on the shared variable. In the second case, all even numbered processes wait between the *read* and *write* operations and all odd numbered processes wait before the *read* operation performed on the shared variable. The waiting for the processes, in this scenario, is active wait, in the sense that busy loops are used to capture

the variability exhibited in an actual heterogeneous setting (Appendix B contains the code). Ideally, the final values should vary between 2 and 600. The actual values obtained and their frequencies are shown in Table 3.

TABLE 3

RESULTS OBTAINED WHEN SIXTY PROCESSES UPDATED A SHARED VARIABLE (1000 RUNS)

Value obtained	Frequency	
	Case 1	Case 2
26	0.1%	0.0%
29	0.2%	0.1%
30	0.4%	0.1%
31	1.3%	0.4%
32	2.3%	0.8%
33	4.3%	0.6%
34	6.3%	1.7%
35	5.7%	2.3%
36	7.9%	2.8%
37	8.1%	4.8%
38	7.4%	3.5%
39	8.7%	4.3%
40	8.2%	5.6%
41	5.9%	5.3%
42	6.2%	6.3%
43	4.1%	5.5%
44	3.6%	6.3%
45	3.1%	5.6%
46	3.7%	4.8%
47	3.2%	5.5%
48	2.2%	3.6%
49	1.2%	4.0%
50	1.4%	3.8%
51	1.3%	2.8%

TABLE 3 (Continued)

Value obtained	Frequency	
	Case 1	Case 2
52	0.5%	2.8%
53	0.8%	1.9%
54	0.6%	2.1%
55	0.4%	1.9%
56	0.3%	2.0%
57	0.1%	1.8%
58	0.1%	1.1%
59	0.1%	1.2%
60	0.1%	0.8%
61	0.1%	1.0%
62	0.1%	0.4%
63	0.0%	0.9%
64	0.0%	0.1%
65	0.0%	0.4%
66	0.0%	0.2%
67	0.0%	0.1%
69	0.0%	0.1%
70	0.0%	0.1%
71	0.0%	0.1%
72	0.0%	0.4%
76	0.0%	0.1%

The processes (that are forked) in the above runs are intended to depict the actual processes in a real system, i.e., a heterogeneous set of processes, trying to access shared variables.

4.2 A Semaphore Implementation of the New Primitive

For the purpose of comparison and validation of the proposed primitive, another simulation using semaphores was implemented. In this simulation, one semaphore per synchronizing agent was used. Before accessing a synchronizing agent, the P operation on the semaphore associated with that synchronizing agent is performed. Similarly, the V operation on the semaphore associated with the synchronizing agent is performed once the processing on the access using that particular synchronizing agent is completed. No explicit queues are provided for each synchronizing agent as the queues are implicit with semaphores. There are also no overflow queues.

Most of the variables used to simulate the proposed primitive were also used in this approach in order to have reasonable similarities between the two simulations. If there are no variables (such as agent status, lp_agents_busy, hp_agents_busy, lpagents, and hpagents) available along with semaphores, then in order to check for a free synchronizing agent or to find out the synchronizing agent whose waiting queue size is minimum, all the accesses will be made to wait on one synchronizing agent. This will lead to uneven allocation of accesses, i.e., even when a free synchronizing agent is available, access will be made to wait on a busy agent. Since there are no overflow queues, the accesses, as they enter the system, are either directly allocated to a free synchronizing agent or made to wait on a synchronizing agent with minimum waiting queue size by performing a P operation on the semaphore associated with that particular synchronizing agent. A random number generator is used to generate a number to designate the arriving access either as a high priority or as a low priority access. In this

approach, parallelism is achieved by forking a process for every access that enters the system. The access is made to wait for a random amount of time after having the synchronizing agent to simulate the processing time. To make the simulation generalizable, the variables, number of synchronizing agents (n), number of active synchronizing agents (m), lpagents, and hpagents were predefined at the beginning of the simulation.

Six synchronizing agents (two high priority and four low priority) are used in this approach as in the case with the proposed primitive. When a new access enters the system, depending upon its priority, the appropriate synchronizing agents are checked for availability. If any free agent is available, a P operation on the semaphore associated with that agent is performed. If none is available and if the access is of high priority, the availability of a free low priority agent is checked and, if one is available, it is allocated. Otherwise, the access is made to wait on the high priority agent, whose waiting queue size is minimum among high priority synchronizing agents, by performing a P operation on that semaphore. If the access is of low priority and there is no free low priority synchronizing agent available, the access is made to wait on the synchronizing agent, whose waiting queue size is minimum among low priority synchronizing agents, by performing a P operation on the semaphore associated with that agent.

When an access releases the synchronizing agent, a V operation is performed on the semaphore associated with that particular synchronizing agent. This V operation wakes up one of the blocked processes on that particular semaphore. The access which was woken is allocated to the synchronizing agent which is free now. A variable is used to

keep track of the number of accesses waiting on the agent. If there are no more accesses waiting on the agent, the agent is put to sleep indicating the availability of this synchronizing agent.

Both simulations, one using the proposed primitive and the other using semaphores, were run independently. The simulation using the proposed primitive was run for five days and the simulation using semaphores was run only for two hours at a time. The latter simulation could not be run for an unlimited number of accesses as there is a system defined limit on the number of processes that can be forked in the Sequent Symmetry S/81 system running the DYNIX/ptx operating system.

4.3 Study of the Data Collected

The two simulations, one with the proposed primitive and the other with semaphores, were run and tested extensively.

The simulation of the new primitive was tested for various possible cases, such as more than one access arriving at the same time, new accesses arriving when the agent's queues are full, new accesses arriving when the overflow queue is full, the arrival rate of the accesses being less than the predefined rate, etc. The simulation of the new primitive was initially tested for a few accesses by checking for each access as it enters the system. When the simulation was run continuously, the various queue sizes are displayed and monitored for any discrepancies (such as allocating new access to the agent whose queue size is not minimum, allowing an access to enter the synchronizing system when the overflow queue is full, not putting the synchronizing agent to sleep when it is appropriate

to do so or making the synchronizing agent inactive when necessary, etc.). The simulation was found to be functioning without any problems/discrepancies.

The simulation using semaphores was also run for 200 accesses. The number of accesses that was simulated was limited due to a system limitation of the number of processes that can be forked. However, the simulation was tested for different combinations of the variables and cases. In the simulation using semaphores too, the various queue sizes were monitored.

4.4 Observations

The following observations are made after running both simulations, one using the proposed primitive and the other using semaphores.

Advantages of the proposed primitive:

1) The proposed primitive is straightforward and easy to understand, when compared to the available primitives.
2) The algorithm is relatively simple, as more complex constructs like semaphores are not present.
3) The behavior of the simulation using the proposed primitive is more predictable when compared with the other simulated primitive that uses semaphores. The criteria for allocating accesses to agents is well defined in the case of the proposed primitive.
4) Contrary to the case of the new primitive, the queue sizes can become arbitrarily large in the case of the approach that uses semaphores, because there

are no overflow queues and there is no checking to find out whether the queues are full or not. In the proposed primitive, the queue sizes are fixed.

5) The proposed primitive is general enough to be implemented both on centralized systems as well as distributed systems.

Disadvantages of the proposed primitive:

1) The number of queues that should be provided (one for each synchronizing agent and one overflow queue for each priority class). This requires additional space and extra queue manipulation time.

2) System resources will be wasted in checking for availability of space (in the overflow queues), when an access is made to wait outside the synchronization system.

CHAPTER V

SUMMARY, CONCLUSIONS, AND FUTURE WORK

5.1 Summary

In Chapter I, the significance of concurrent processes, the need for their control, and the main objective of this thesis was briefly stated. Chapter II presented the literature review. The topics covered in this chapter consisted of control problems in parallel processing, a number of proposed software and hardware solutions to the mutual exclusion problem, and recent developments in the field of synchronization. Chapter III discussed the implementation issues of the new synchronization and mutual exclusion primitive. Section 1 of Chapter III addressed the specification of the new approach along with the pseudocode of the algorithm. Section 2 described the implementation details and Section 3 compared the new primitive with other existing primitives. Chapter IV outlined the pilot evaluation of the primitive.

5.2 Conclusions

The proposed primitive satisfies the requirements of an operational mutual exclusion scheme as outlined by Dijkstra [Dijkstra65a]. The primitive was successfully implemented on a centralized system, i.e., a Sequent Symmetry S/81 at the Computer

Science Department at Oklahoma State University. This primitive is general enough to be implemented not only on centralized systems, but also on parallel and distributed systems (the inter-process communication in distributed systems can be achieved by sending/receiving messages in the implementation of the proposed primitive).

In the worst case, i.e., when all accesses are concurrent and trying to access the same shared variable/resource, the proposed primitive can degenerate to the simple case of using a single semaphore (on the critical section) to access that shared variable/resource.

5.3 Future Work

The new primitive can be implemented using the tasks and queues of the C++ Task Library. The C++ Task Library is a C++ Language System coroutine library with implicit queues for communication among tasks [Stroustrup90]. In the simulation, the queues are defined explicitly. Queues of the Task Library can be used along with the Tasks to simulate virtual parallelism.

The proposed primitive can also be simulated on a distributed system and its performance can be studied. The proposed primitive can be incorporated in the USE system [Daily93] [Hassan92] [Hassan93a] [Hassan93b] [Hassan94] [Jhun92] for synchronization and mutual exclusion purposes.

The implementation of the proposed primitive did not include an interface. In other words, the new primitive must be defined as an abstract data type with a well-defined interface in the form of function calls, similar to the case of semaphores and monitors.

Without an interface, it is difficult for a user to use the proposed primitive easily.

A particular implementation aspect when different accesses of the same priority class (after being allocated to different synchronizing agents) are trying to access the same shared variable was not addressed. This can be implemented using a FIFO discipline.

To gain more confidence in the generality of the proposed primitive, well-known mutual exclusion problems (such as, the readers/writers problem, the producer/consumer problem, and the bounded-buffer problem) need to be solved using the new primitive.

REFERENCES

[Daily93] S.R. Daily and Mansur H. Samadzadeh, "Object-Oriented Simulation of Capability Based Architectures", *The Twenty sixth Annual Simulation Symposium*, Sponsored by SCS, IEEE-CS, and ACM, in conjunction with *The 1993 Simulation Multi-Conference*, pp. 258-266, Washington D.C., March 29 - April 1, 1993.

[De Bruijn67] J.G. De Bruijn, "Additional Comments on a Problem in a Concurrent Programming Control", *Communications of the ACM*, Vol. 10, No. 3, pp. 137-138, 1967.

[Dijkstra65a] E.W. Dijkstra, "Cooperating Sequential Processes", In *Programming Languages*, F. Genuys (Ed.); Academic Press, New York, NY, pp. 43-112, 1965.

[Dijkstra65b] E.W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, No. 5, p. 569, 1965.

[Doddaballapur88] Jayasimha N. Doddaballapur, "Distributed Synchronizers", *Proc. of 1988 International Conference on Parallel Processing*, University Park, PA, pp. 23-27, August 1988.

[Eisenberg72] M.A. Eisenberg and M.R. MacGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem", *Communications of the ACM*, Vol. 15, No. 11, p. 999, 1972.

[Gottlieb83] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, pp. 164-189, 1983.

[Hassan92] Khaled M. Hassan and Mansur H. Samadzadeh, "An Object-Oriented Environment for Simulation and Evaluation of Architectures", *Proceedings of the IEEE 25th Annual Simulation Symposium* in conjunction with *The 1992 SCS Simulation Multiconference*, Orlando, FL, pp. 91-97, April 1992.

[Hassan93a] Khaled M. Hassan, Ik-Jeong Jhun, and Mansur H. Samadzadeh, "Using the C++ Task Library to Solve a Classical Mutual Exclusion Problem", *Proceedings of the Ninth International Conference on Systems Engineering*, pp. 255-259, Las Vegas, NV,

July 1993.

[Hassan93b] Khaled M. Hassan and Mansur H. samadzadeh, "The USE System: An Evolving Environment for the Simulation of Operating Systems and Architectures", Submitted in August 1993 to *The International Journal of Computer Simulation* for a special issue on Computer Architecture Simulation.

[Hassan94] Khaled M. Hassan and Mansur H. Samadzadeh, "Adding Virtual Memory to the USE Object-Oriented Simulation Environment", to appear in *Proceedings of the Object-Oriented Simulation Conference*, part of the *1994 Western Multi-Conference*, Tempe, AZ, January 1994.

[Jhun92] Ik-Jeong Jhun, Khaled M. Hassan, and Mansur H. Samadzadeh, "Simulation of a Computing Environment Using Stochastic Processes and the Object-Oriented Technology", *Proceedings of the Twenty Third Annual Pittsburgh Conference on Modelling and Simulation*, Vol. 23, part 3, Published and Distributed by Instrument Society of America, Edited by William G. Vogt and Marlin H. Mickle, Pittsburgh, PA, pp. 1579-1585, April 30 - May 1, 1992.

[Knuth66] D.E. Knuth, "Additional Comments on a Problem in a Concurrent Programming Control", *Communications of the ACM*, Vol. 9, No. 5, pp. 321-322, 1966.

[McCabe76] T.J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. SE-2, pp. 308-320, December 1976.

[Maekawa87] M. Maekawa, A.E. Oldehoeft, and R.R. Oldehoeft, *Operating Systems: Advanced Concepts*, The Benjamin Cummings Publishing Co., Inc., 1987.

[Peterson81] G.L. Peterson, "Myths about the Mutual Exclusion Problem", *Information Processing Letters*, Vol. 12, No. 3, pp. 115-116, 1981.

[Presser75] Leon Presser, "Multiprogramming Coordination", *ACM Computing Surveys*, Vol. 7, No. 1, pp. 21-44, 1975.

[Raynal86] M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press, Cambridge, MA, 1986.

[Saeed90] F. Saeed, Mansur H. Samadzadeh, and K.M. George, "A Simulation Environment for n-Process Mutual Exclusion Algorithms", *Proceedings of The Twenty-First Annual Pittsburgh Conference on Modelling and Simulation*, Vol. 21, Part 1, Pittsburgh, PA, pp. 371-375, May 1990.

[Saeed92] F. Saeed, K. M. George, and Mansur H. Samadzadeh, "An Implementation of

Classical Mutual Exclusion Algorithms in Ada", *ACM Ada Letters*, Vol. 12, No. 1, pp. 73-84, January-February 1992.

[Shaw74] A.C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, New Jersey, 1974.

[Stroustrup90] B. Stroustrup and J.E.Shopiro, *A Set of C++ Classes for Coroutine Style Programming*, AT&T C++ Language System Release 2.1 Library Manual, 1990.

[Tang90] Peiyi Tang, Pen-Chang Yew, and Chuan-Q. Zhu, "Compiler Techniques for Data Synchronization in Nested Parallel Loops", *1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, pp. 177-186, June 1990.

[Zand89] M.K. Zand, Mansur H. Samadzadeh, and K.M. George, "A Control Driver for the n-Process Concurrent Programming Problem", *Proceedings of the ACM South Central Regional Conference*, Tulsa, OK, p. 51, November 1989.

APPENDICES

APPENDIX A

GLOSSARY

Accesses:	A heterogenous set of processes which are trying to access a shared variable and require synchronization.
Asynchronous Processes:	Processes that require occasional synchronization and cooperation.
Critical Section:	When a process is accessing shared data, the process is said to be in its critical section.
Deadlock:	A process in a multiprogramming system is said to be in a state of deadlock (or deadlocked) if it is waiting for a particular event that will not occur.
Data Dependence:	While tasks are executed concurrently on multiple processors, the order of access to each particular data in the original program must be preserved to guarantee the correctness of the execution. This access order is called data dependence.
Explicit Data Synchronization:	The precedence relation on tasks can be relaxed if all data dependencies can be enforced by direct synchronization, which is called explicit data synchronization.
Implicit Data Synchronization:	If the type of enforcement on data dependencies is implied in the precedence relation, it is called implicit data synchronization.
Light Weight Process:	A process that has less state (the amount of resources associated with process) and requires minimum context switching time.
Memory Hierarchy:	The arrangement of a set of partially-shared memories

to suit a particular system's memory requirements.

Mutual Exclusion:	Each process accessing the shared data excludes all others from doing so simultaneously. This is called mutual exclusion.
Process:	A program in execution.
PE:	Processing Element.
Semaphore:	A semaphore is a non-negative integer variable that can be handled only by the P and V operations.
Synchronizing Agent:	A virtual processor that enforces mutual exclusion and synchronization among accesses.
Synchronization System:	A system consisting of queues and synchronizing agents.
Tasks:	Light weight processes that share address spaces.
Walk-in:	If a processing element, after decrementing the variable at a particular level, finds the resulting value to be zero, it proceeds to the next higher level. Otherwise, it waits for the variable to assume a special value. This scheme is called walk-in.
Walk-out:	When a processing element finds that the variable, on which it is waiting, has assumed a special value, it communicates this information to the next lower level. This scheme is called walk-out.

TRADEMARK INFORMATION

DYNIX/ptx:	A registered trademark of Sequent Computer System, Inc.
Sequent S/81:	A registered trademark of Sequent Computer System, Inc.
UNIX:	A registered trademark of AT&T.
X window System:	A registered trademark of the Massachusetts Institute of Technology.

APPENDIX B

PROGRAM LISTINGS

/**
The program sim.c is the implementation of the proposed primitive in C on Sequent Symmetry S/81. Program sim_sem.c is the implementation of the semaphores. The small X Window interface to display the various queue sizes is provided for the program using the proposed primitive. Programs env1.c, env2.c, env3.c, and env4.c are for the purpose of showing the necessity of synchronization among processes trying to access shared variables.
**/

```
                    /* Program sim.c starts here */

#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include "windows.h"
#define MAX_QUEUE_SIZE 2
#define OVERFLOWQUEUESIZE 5
#define m 4
#define MAX_SYNCH_AGENTS 6
#define HPAGENTS 2
#define LPAGENTS 4
#define BUSY 1
#define SLEEP 0
#define ACTIVE 1
#define INACTIVE 0
#define TRUE 1
#define FALSE 0

/*structure definitions */

struct _queue{
      int access_id, start_time, service_time, arrival_time;
};

typedef struct _synch_agents{
      int status;
      int status1;
      int queue_size;
        char priority[3];
      struct _queue queue[MAX_QUEUE_SIZE], active_access;
}synch;

double seed = 1.0;


float random();
```

```
void initialize_synch_agents(synch []);
void start_processing(synch [], int , struct _queue [], int *,
  struct _queue [], int *, int *, int *, int *, int *, FILE *, int []);
void adjust_queues(synch *, struct _queue [], int *);
int find_free_agent(synch [], int, char []);
int find_inactive_agent(synch []);
int find_min_queue_agent(synch [], int, char []);
void print_stat(synch [], int, int, int, int);
void copy_values(int [], int, synch [], int, int);
void write_output(synch [], int, int , int, int []);


/*******************************************************************
                     MAIN FUNCTION STARTS HERE
*******************************************************************/

main(argc,argv)
int argc;
char **argv;
{

    synch synch_agents[MAX_SYNCH_AGENTS];
    int active_synch_agents=0, synch_agents_busy=FALSE, hpsynchbusy=FALSE;
    int hpagents = 0, lpagents = 0, ACCESSALLOWED = TRUE, hist_array[20];
    struct _queue lp_overflow_queue[5], hp_overflow_queue[5];
    int lp_count = 0, hp_count = 0, jobsinthesystem = FALSE, stat_val[8];
    int no_of_accesses = 0, inter_arrival_time = 0, current_time = 0;
    int previous_time = 0, new_agent, priority, flag, i, queue_val[48];
    FILE *ofp;

    /* initializing the variables, and the window for display */

    initialize_synch_agents(synch_agents);
    initialize_window(argc, argv);

    ofp = fopen("Output", "w");

    fprintf(ofp,"Access   Arrival   Allocated   Finished\n");
    fprintf(ofp," ID       time        time         time\n\n\n");

    /* continues till there are accesses in the system */

    while((no_of_accesses < 200) || (jobsinthesystem)){

      if(no_of_accesses < 200){

        jobsinthesystem = TRUE;

      /* if more than one access arrives at the same time, they are loaded
      at the same time without incrementing the clock */

        while((current_time >= (previous_time + inter_arrival_time)) &&
                                    (no_of_accesses < 200)){

          /* access is allowed only if overflow queue is not full */

          if(ACCESSALLOWED == TRUE)
             priority = (int)(4 * random() + 0);

          /* if the access arrived into the system is of high priority */

          if(priority == 0){
```

```
/* if there are less than 5 accesses in the hp overflow queue */

if(hp_count < 5){

    /* if the over flow queue is non empty, then insert it */

    if(hp_count != 0){

      hp_overflow_queue[hp_count].access_id = no_of_accesses;
      hp_overflow_queue[hp_count].service_time =
                                  (int)(5*random()+10);
      hp_overflow_queue[hp_count].arrival_time = current_time;
      hp_count += 1;
    }

    /* if the high priority synch_agent is free, allocate */

    else if(hpsynchbusy == FALSE){

      new_agent = find_free_agent(synch_agents, m, "HP");
      synch_agents[new_agent].active_access.access_id =
                                  no_of_accesses;
      synch_agents[new_agent].active_access.service_time =
                                  (int)(5 * random() + 10);
      synch_agents[new_agent].active_access.start_time =
                                  current_time;
      synch_agents[new_agent].active_access.arrival_time =
                                  current_time;
      synch_agents[new_agent].status = BUSY;
      hpagents += 1;
      if(hpagents == HPAGENTS)
        hpsynchbusy = TRUE;
    }

    /* else check if any of the lp agents is available */

    else{

      /* if there is one of them free, then allocate to it */

      if(synch_agents_busy == FALSE){

        new_agent = find_free_agent(synch_agents, m, "LP");
        synch_agents[new_agent].active_access.access_id =
                                  no_of_accesses;
        synch_agents[new_agent].active_access.service_time =
                                  (int) (5 * random() + 10);
        synch_agents[new_agent].active_access.start_time =
                                  current_time;
        synch_agents[new_agent].active_access.arrival_time =
                                  current_time;
        synch_agents[new_agent].status = BUSY;
        lpagents += 1;
        if(lpagents >= (m - HPAGENTS))
           synch_agents_busy = TRUE;
      }

      /* if none is free, find the queue size, and if it is
      maximum, insert it into overflow, otherwise wait on
      the hp synch_agent */

      else{
```

```
        new_agent = find_min_queue_agent(synch_agents, hpagents,
                                                "HP");

        /* if all the queues are full, then insert the access
        into the hp overflow queue */

        if(synch_agents[new_agent].queue_size ==
                                MAX_QUEUE_SIZE){

          hp_overflow_queue[hp_count].access_id =
                                no_of_accesses;
          hp_overflow_queue[hp_count].arrival_time =
                                current_time;
          hp_overflow_queue[hp_count].service_time =
                                (int)(5 * random() + 10);
          hp_count += 1;
        }

        /* if there is space then insert the access into that
        queue */

        else{

          synch_agents[new_agent].queue[synch_agents[new_agent].
               queue_size].access_id = no_of_accesses;
          synch_agents[new_agent].queue[synch_agents[new_agent].
               queue_size].arrival_time = current_time;
          synch_agents[new_agent].queue[synch_agents[new_agent].
               queue_size].service_time=(int)(5*random() + 10);
          synch_agents[new_agent].queue_size += 1;
        }
      }
    }

    /* update the clock, and inter arrival time */

    previous_time = current_time;
    inter_arrival_time = (int)(5 * random() + 0);
    no_of_accesses += 1;
    ACCESSALLOWED = TRUE;

  }

  /* if the hp overflow queue is full, then the access is not
  allowed into the system */

  else
    ACCESSALLOWED = FALSE;

}

/* when a low priority access enters the system */

else{

  /* if there are accesses already in the lp overflow queue,
     then insert the new access directly into the overflow
     queue, if there is space. Otherwise the access is made to
     wait. */

 if(lp_count < 5){
```

```
/* if there are already accesses in the lp overflow queue and
   there is space, directly insert the access into the lp
   over-flow queue */

if((lp_count != 0) && (lp_count < 5)){

   lp_overflow_queue[lp_count].access_id = no_of_accesses;
   lp_overflow_queue[lp_count].arrival_time = current_time;
   lp_overflow_queue[lp_count].service_time =
                              (int)(5 * random() + 10);
   lp_count += 1;
}

/* if there are no accesses in the lp overflow queue */

else if(lp_count == 0){

   /* if there is a free synchronizing agent available, then
      allocate this new access directly to the free agent */

   if(synch_agents_busy == FALSE){

     new_agent = find_free_agent(synch_agents, m, "LP");
     synch_agents[new_agent].active_access.access_id =
                              no_of_accesses;
     synch_agents[new_agent].active_access.service_time =
                              (int)(5 * random() + 10);
     synch_agents[new_agent].active_access.arrival_time =
                              current_time;
     synch_agents[new_agent].active_access.start_time =
                              current_time;
     synch_agents[new_agent].status = BUSY;
     lpagents += 1;
     if(lpagents >= (m - HPAGENTS))
        synch_agents_busy = TRUE;
   }

   /* if all the agents are busy */

   else{

     /* find the agent whose waiting queue is minimum */

     new_agent = find_min_queue_agent(synch_agents, (lpagents
                              + HPAGENTS), "LP");

     /* if all the waiting queues are full */

     if(synch_agents[new_agent].queue_size ==
                              MAX_QUEUE_SIZE){

       /* if the queue is equal to the MAX_QUEUE_SIZE, then
          find an inactive agent available, allocate this
          access to it, otherwise insert it into lp overflow
          queue */

       if(lpagents < LPAGENTS){

         new_agent = find_inactive_agent(synch_agents);
         synch_agents[new_agent].active_access.access_id =
                              no_of_accesses;
         synch_agents[new_agent].active_access.service_time
```

```
                            = (int)(5 * random() + 10);
          synch_agents[new_agent].active_access.arrival_time
                            = current_time;
          synch_agents[new_agent].active_access.start_time =
                            current_time;
          synch_agents[new_agent].status1 = ACTIVE;
          synch_agents[new_agent].status = BUSY;
          lpagents += 1;
        }

        /* if there are no more inactive agents available,
           then insert this access into lp overflow queue */

        else{

            lp_overflow_queue[lp_count].access_id =
                            no_of_accesses;
            lp_overflow_queue[lp_count].arrival_time =
                            current_time;
            lp_overflow_queue[lp_count].service_time =
                            (int)(5 * random() + 10);
            lp_count += 1;
        }
      }

      /* if there is space in the minimum queue, then insert
         the access into the queue of that agent */

      else{

          /* if the wait queue size is less, then insert this
             access into the wait queue of the agent */

         synch_agents[new_agent].queue[synch_agents
                  [new_agent].queue_size].access_id =
                                 no_of_accesses;
         synch_agents[new_agent].queue[synch_agents
                  [new_agent].queue_size].service_time =
                                 (int)(5 * random() + 10);
         synch_agents[new_agent].queue[synch_agents
                  [new_agent].queue_size].arrival_time =
                                 current_time;
         synch_agents[new_agent].queue_size += 1;
      }
    }
  }

  /* update clock, interarrival time */

  previous_time = current_time;
  inter_arrival_time = (int)(5 * random() + 0);
  if(no_of_accesses == 7)
     inter_arrival_time = 50;
  no_of_accesses += 1;
  ACCESSALLOWED = TRUE;
}

/* if the overflow queue is full, then the access is not
   allowed to enter. */

else
```

```
              ACCESSALLOWED = FALSE;
          }

          hist_array[current_time % 20] = 1;

          if(ACCESSALLOWED == FALSE) break;
        }
      }


      start_processing(synch_agents, current_time, lp_overflow_queue,
          &lp_count, hp_overflow_queue, &hp_count, &lpagents,
          &synch_agents_busy, &hpsynchbusy, &hpagents, ofp, hist_array);

      /* print and display the queue sizes */

      if(current_time % 10 == 0){
        print_stat (synch_agents, lp_count, hp_count, current_time,
                                        no_of_accesses);
        draw_window(queue_val,current_time);
        sleep(2);
      }

      copy_values(queue_val, current_time, synch_agents, lp_count,
                                        hp_count);

      /* display average queue sizes */

      if((current_time % 15 == 0) && (current_time != 0))
         system("gnuplot test");

      current_time += 1;

      write_output(synch_agents, lp_count, hp_count, current_time,
                                        stat_val);

      hist_array[current_time % 20] = 0;

      flag = TRUE;

      for(i = 0; i < MAX_SYNCH_AGENTS; i++){
         if(synch_agents[i].status == BUSY){
            flag = FALSE; break;
         }
      }

      if(flag == TRUE) jobsinthesystem = FALSE;
    }
    fclose(ofp);
    print_stat(synch_agents, lp_count, hp_count, current_time,
                                        no_of_accesses);
    draw_window(queue_val, current_time);
    system("gnuplot test");
    sleep(4);
    close_window();
}


/************************************************************************
                    FUNCTION FOR RANDOM NUMBER GENERATOR
************************************************************************
```

```
This function is used to generate a random number between 0 and 1. This
generator  is used for generating access priority, inter-arrival time and
service time. */

float random()
{
      long a = 16807.0;
      long M = 2147483647.0;
      long q = 127773.0, r = 2836.0, lo, hi, test;

      hi = (int)(seed/q);
      lo = seed - q*hi;
      test = a*lo-r*hi;
      if (test > 0.0)
            seed = test;
      else
            seed = test + M;
      return(seed/M);
}


/************************************************************************
                        FUNCTION TO FIND FREE AGENT
*************************************************************************
This function finds and returns the free synchronizing agent depending
upon the priority. */

find_free_agent(syn_agent, count, priority)
synch syn_agent[];
int count;
char priority[];
{

      int i = 0;

      for(i = 0; i < MAX_SYNCH_AGENTS; i++){
        if((strcmp(syn_agent[i].priority, priority) == 0) &&
                               (syn_agent[i].status1 == ACTIVE))
          if(syn_agent[i].status == SLEEP) break;
      }
      return(i);
}

/************************************************************************
                        FUNCTION TO FIND INACTIVE AGENT
*************************************************************************

This function finds and returns the first inactive low priority
synchronizing agent */

find_inactive_agent(syn_agent)
synch syn_agent[];
{

      int i = 0;

      for(i = 1; i < MAX_SYNCH_AGENTS; i++)
         if(syn_agent[i].status1 == INACTIVE) break;
      return(i);
}
```

```
/**********************************************************************
                  FUNCTION TO FIND AGENT WHOSE QUEUE IS MIN.
***********************************************************************
This function finds and returns the agent number whose queue length is
minimum out of the group of synchronizing agents whose priority matches
with the argument priority  */

find_min_queue_agent(syn_agent, count, priority)
synch syn_agent[];
int count;
char priority[];
{

      int i = 0, j = HPAGENTS;

      /* for low priority agents */

      if((strcmp(priority, "LP")) == 0){
         for(j = HPAGENTS; j < MAX_SYNCH_AGENTS; j++)
            if(syn_agent[j].status == BUSY) break;
         for(i = HPAGENTS; i < MAX_SYNCH_AGENTS; i++){
            if((syn_agent[i].status == BUSY) &&
                   (syn_agent[j].queue_size > syn_agent[i].queue_size))
            j = i;
         }
      }

      /* for high priority agents */

      else{
         j = 0;
         for(i = 0; i < count; i++){
            if((syn_agent[i].status == BUSY) &&
                   (syn_agent[j].queue_size > syn_agent[i].queue_size))
              j = i;
         }
      }
      return(j);
}


/**********************************************************************
                         FUNCTION FOR INITIALIZATION
***********************************************************************
This function initializes all the synchronizing agents. First m number of
agents are designated as active synchronizing agents and the remaining
agents are initialized as inactive synchronizing agents. Also first
HPAGENTS number of agents are designated as high priority agents and the
remaining are as low priority agents */

void initialize_synch_agents(syn_agent)
synch syn_agent[];
{

      int i = 0;

      for(i = 0; i < m; i++){
        syn_agent[i].status = SLEEP;
        syn_agent[i].status1 = ACTIVE;
        syn_agent[i].queue_size = 0;
      }
      for(i = m; i < MAX_SYNCH_AGENTS; i++){
```

```
        syn_agent[i].status = SLEEP;
        syn_agent[i].status1 = INACTIVE;
        syn_agent[i].queue_size = 0;
      }
       for(i = 0; i < HPAGENTS; i++)
        strcpy(syn_agent[i].priority, "HP");
       for(i = HPAGENTS; i < MAX_SYNCH_AGENTS; i++)
        strcpy(syn_agent[i].priority, "LP");
}


/*********************************************************************
             FUNCTION TO REMOVE ACCESS FROM THE AGENT
**********************************************************************
This function is used to check whether the service time of the particular
accesshas expired or not. If so, the access is removed from the agent and
the accessfrom the head of the queue associated with that particular agent
is allocated toit. If there are no accesses waiting on that agent,
depending on the history ofthe rate of arrival of the accesses, the agent
is put to sleep or made inactive.The flags and the counts of agents are
accordingly updated. */


void start_processing(syn_agents, currenttime, lp_queue, lpcount,
      hp_queue, hpcount, active_synch_agents, synch_agents_busy,
      hpsynch_busy, HPagents, fp, histarray)
synch syn_agents[];
int currenttime, *lpcount, *hpcount, *active_synch_agents;
int *synch_agents_busy, *hpsynch_busy, *HPagents, histarray[];
struct _queue lp_queue[], hp_queue[];
FILE *fp;
{

int i = 0, sum;

sum = 0;
for(i = 0; i < 20; i++)
    sum += histarray[i];

for(i = 0; i < MAX_SYNCH_AGENTS; i++){
  if(syn_agents[i].status == BUSY){

    /* if the service time of the access is expired, it is removed from
    the agent. */

    if(currenttime >= (syn_agents[i].active_access.start_time +
                        syn_agents[i].active_access.service_time)){
       fprintf(fp,"%5d %7d %10d %10d\n",
                  syn_agents[i].active_access.access_id,
                  syn_agents[i].active_access.arrival_time,
                  syn_agents[i].active_access.start_time, currenttime);

       /* if there are accesses waiting on this agent, the access from the
          head of the queue is removed and allocated to it. */

       if(syn_agents[i].queue_size != 0){
        syn_agents[i].active_access.access_id =
                              syn_agents[i].queue[0].access_id;
        syn_agents[i].active_access.service_time =
                              syn_agents[i].queue[0].service_time;
        syn_agents[i].active_access.arrival_time =
                              syn_agents[i].queue[0].arrival_time;
```

```
        syn_agents[i].active_access.start_time = currenttime;
        if(strcmp(syn_agents[i].priority, "HP") == 0)
           adjust_queues(&syn_agents[i], hp_queue, hpcount);
        if(strcmp(syn_agents[i].priority, "LP") == 0)
           adjust_queues(&syn_agents[i], lp_queue, lpcount);
       }

       /* if no access is waiting on the agent, it is put to SLEEP.
       Depending on the arrival rate history, the agent is made inactive.
       */

       else{
        syn_agents[i].status = SLEEP;
        if(strcmp(syn_agents[i].priority, "HP") == 0){
             *hpsynch_busy = FALSE;
           *HPagents -= 1;
        }
        else{
           if((sum == 0) && (*active_synch_agents > (m - HPAGENTS)))
            syn_agents[i].status1 = INACTIVE;
           else
              *synch_agents_busy = FALSE;
           *active_synch_agents -= 1;
        }
       }
    }
  }
}
}


/*********************************************************************
                        FUNCTION TO UPDATE THE QUEUES
**********************************************************************
This function is used to update the various queues. When an access leaves
the synchronizing agent, the access from the head of its queue is
allocated to that agent. This function rearranges the accesses in the
queue associated with that particular agent and the overflow queue. */


void adjust_queues(synch_agent, Queue, count)
synch *synch_agent;
struct _queue Queue[];
int *count;
{

int i = 0;

for(i = 0; i < (synch_agent->queue_size - 1); i++){
   synch_agent->queue[i].access_id = synch_agent->queue[i + 1].access_id;
   synch_agent->queue[i].service_time =
                        synch_agent->queue[i+1].service_time;
   synch_agent->queue[i].arrival_time =
                        synch_agent->queue[i+1].arrival_time;
}
synch_agent->queue_size -= 1;
if(*count != 0){
   synch_agent->queue[synch_agent->queue_size].access_id =
                                    Queue[0].access_id;
   synch_agent->queue[synch_agent->queue_size].service_time =
                                    Queue[0].service_time;
   synch_agent->queue[synch_agent->queue_size].arrival_time =
```

```
                                  Queue[0].arrival_time;
   synch_agent->queue_size += 1;
   for(i = 0; i < ((*count) - 1); i++){
      Queue[i].access_id = Queue[i + 1].access_id;
      Queue[i].service_time = Queue[i + 1].service_time;
      Queue[i].arrival_time = Queue[i + 1].arrival_time;
   }
   *count -= 1;
}
}


/*************************************************************************
                    FUNCTION TO PRINT THE OUTPUT
**************************************************************************
This function is used to display on the screen the sizes of various queues
used at a particular instant of time. */

void print_stat(syn_agents, lpcount, hpcount, time, accesses)
synch syn_agents[];
int lpcount,  hpcount, time, accesses;
{
   int i = 0, j = 0;

   system("tput clear");
   printf("                                 TIME  IS:  %d    (ACCESSES:
                              %d)\n\n",time,accesses);
   printf("HP OVERFLOWQUEUE        : ");
   for(i = 0; i < hpcount; i++)
      printf("*");
   printf("\n");
   for(j = 0; j < MAX_SYNCH_AGENTS; j++){
        printf("SYNCHRONIZING AGENT # %d: ", j);
      for(i = 0; i < syn_agents[j].queue_size; i++)
        printf("*");
      if(syn_agents[j].status1 == INACTIVE)
        printf("      INACTIVE");
      else if(syn_agents[j].status == SLEEP)
        printf("      SLEEP");
        printf("\n");
   }
   printf("LP OVERFLOWQUEUE        : ");
   for(i = 0; i < lpcount; i++)
      printf("*");
   printf("\n");
}



/*************************************************************************
                    FUNCTION TO DISPLAY THE QUEUE SIZES
**************************************************************************
This function is used to display in the newly created window, the various
queue sizes at a particular instant of time. A filled rectangle is
displayed corresponding to the particular queue size */

draw_window(queue, currenttime)
int queue[], currenttime;
{

    int i, j, k;
```

```
    /* string to differentiate the various queues is defined */

    char name[20];
    char name1[10];
    char string1[] = "H";
    char string2[] = "0";
    char string3[] = "1";
    char string4[] = "2";
    char string5[] = "3";
    char string6[] = "4";
    char string7[] = "5";
    char string8[] = "L";

    strcpy(name, "CURRENT TIME: ");
    sprintf(name1, "%d", currenttime);
    strcat(name, name1);

    /* clears the window before displaying the rectangles corresponding to
       the size of queue */

    XClearWindow ( mydisplay, mywindow);

    XDrawImageString (mydisplay,mywindow, mygc,100,50, name,
                                          strlen(name));
    /* the string is diplayed in the window */
    j = 0;
    for(i = 0; i < 48; i++){
        if(i % 8 == 0)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string1,
                                          strlen(string1));
        if(i % 8 == 1)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string2,
                                          strlen(string2));
        if(i % 8 == 2)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string3,
                                          strlen(string3));
        if(i % 8 == 3)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string4,
                                          strlen(string4));
        if(i % 8 == 4)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string5,
                                          strlen(string5));
        if(i % 8 == 5)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string6,
                                          strlen(string6));
        if(i % 8 == 6)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string7,
                                          strlen(string7));
        if(i % 8 == 7)
         XDrawImageString (mydisplay,mywindow, mygc,j,350, string8,
                                          strlen(string8));
      /* filled rectangles are displayed according to the size of the
         queue */

      XFillRectangle(mydisplay,mywindow,mygc,j,(330 - (queue[i] * 25)),
                                          10,(queue[i] * 25));
      j += 12;
    }

    XFlush( mydisplay );
}
```

```
/*******************************************************************
                       FUNCTION TO INITIALIZE THE WINDOW
********************************************************************
This function is used to initialize the window with the name and
resources, and then realizing the window. */

initialize_window(argc, argv)
int argc;
char **argv;
{
      /* the variable name is used to display a name to the window */

      char name[] = "QUEUE SIZES DISPLAY";
      mydisplay = XOpenDisplay("");

      /* setting resource parameters */

      myscreen = DefaultScreen (mydisplay);
      mybackground = WhitePixel (mydisplay, myscreen);
      myforeground =BlackPixel (mydisplay, myscreen);
      myhints.x = 350; myhints.y = 375;
      myhints.width = 580; myhints.height = 400;
      myhints.flags = PPosition | PSize;

      /* creating the window and assigning the above declared resources to
         the window */

      mywindow = XCreateSimpleWindow (mydisplay,DefaultRootWindow
            (mydisplay), myhints.x,myhints.y,myhints.width,
            myhints.height,5,myforeground, mybackground);
      XSetStandardProperties (mydisplay, mywindow, name,name,None, argv,
            argc, &myhints);
      mygc = XCreateGC (mydisplay, mywindow, 0,0);
      XSetBackground (mydisplay, mygc, mybackground);
      XSetForeground (mydisplay, mygc, myforeground);
      XSelectInput (mydisplay, mywindow,
                        ExposureMask| ButtonPressMask|KeyPressMask );

      /* window is realized */

      XMapRaised (mydisplay, mywindow);
}


/*******************************************************************
                        FUNCTION TO DESTROY THE X WINDOW
********************************************************************
This function is used to destroy and close the window. All the resources
associated with the window are freed */

close_window()
{
      XFreeGC (mydisplay, mygc);
      XDestroyWindow (mydisplay, mywindow);
      XCloseDisplay (mydisplay);
}


/*******************************************************************
                   FUNCTION TO GENERATE VALUES FOR DISPLAY
********************************************************************
This function is used to copy the values of different queue sizes into an
```

```
array, which will be used for display. */

void copy_values(queue, count, syn_agents, lpagents, hpagents)
int queue[], count, lpagents, hpagents;
synch syn_agents[];
{

   int i, j;

   j = 0;

   switch(count){
           case 0 : queue[0] = hpagents;
                    for(i = 1; i < 7; i++)
                       queue[i] = syn_agents[j++].queue_size;
                    queue[i] = lpagents;
                    break;
           case 1 : queue[8] = hpagents;
                    for(i = 9; i < 14; i ++)
                       queue[i] = syn_agents[j++].queue_size;
                    queue[i] = lpagents;
                    break;
           case 2 : queue[16] = hpagents;
                    for(i = 17; i < 23; i ++)
                       queue[i] = syn_agents[j++].queue_size;
                    queue[i] = lpagents;
                    break;
           case 3 : queue[24] = hpagents;
                    for(i = 25; i < 31; i ++)
                       queue[i] = syn_agents[j++].queue_size;
                    queue[i] = lpagents;
                    break;
           case 4 : queue[32] = hpagents;
                    for(i = 33; i < 39; i ++)
                       queue[i] = syn_agents[j++].queue_size;
                    queue[i] = lpagents;
                    break;
           case 5 : queue[40] = hpagents;
                    for(i = 41; i < 47; i ++)
                       queue[i] = syn_agents[j++].queue_size;
                    queue[i] = lpagents;
                    break;
           default: for(i = 0; i < 40; i++)
                       queue[i] = queue[i + 8];
                    queue[40] = hpagents;
                    for(i = 41; i < 47; i ++)
                       queue[i] = syn_agents[j++].queue_size;
                    queue[i] = lpagents;
                    break;
   }

}


/************************************************************************
                  FUNCTION TO WRITE AVERAGE QUEUE SIZES TO A FILE
************************************************************************

This function is used to write the average queue sizes to a file
"out.dat". This file is used to plot these average values by using GNUPLOT
*/
```

```
void write_output(syn_agents, lpqueue, hpqueue, time, stat_val)
synch syn_agents[];
int lpqueue, hpqueue, time, stat_val[];
{
   FILE *ofp;
   int i;

   ofp = fopen("out.dat", "w");

   stat_val[0] += hpqueue;
   for(i = 1; i < 7; i++)
      stat_val[i] += syn_agents[i-1].queue_size;
   stat_val[i] += lpqueue;
   fprintf(ofp,"0.00\n");
   fprintf(ofp, "%f\n",((float) ((float)stat_val[0] /(float)time)) *
                                                         25.0);
   for(i = 1; i < 7; i++)
      fprintf(ofp,"%f\n",((float)((float)stat_val[i] /(float)time)) *
                                                         25.0);
   fprintf(ofp, "%f\n",((float)((float)stat_val[7] /(float)time)) * 25.0);
   fprintf(ofp,"0.00\n");
   fclose(ofp);
}
```

/* Program sim_sem.c starts here */

/* This program is the simulation using semaphores. One semaphore per synchronizing agent is used. This simulation is compared with the simulation using proposed primitive. All semaphore operations are defined in a header file named sem.h and this header file is included. */

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<parallel/parallel.h>
#include"sem.h"
#define TRUE 1
#define FALSE 0
#define BUSY 1
#define SLEEP 0
#define MAX_AGENTS 6
#define m 2
#define LPAGENTS 4
#define HPAGENTS 2
#define SEMKEY_0   6666
#define SEMKEY_1   1111
#define SEMKEY_2   2222
#define SEMKEY_3   3333
#define SEMKEY_4   4444
#define SEMKEY_5   5555

struct _agents{
      int start_time, service_time, status, status1, queue_size,access_id;
      char priority[3];
};

struct _agent{
      struct _agents AGENT[MAX_AGENTS];
};

int AG[MAX_AGENTS];
double seed = 1.0;
float random();
struct _agent *initialize();

main()
{
      struct _agent *agent;
      int *lp_agents_busy, *no_of_accesses, *current_time, prev_time = 0;
      int *hp_agents_busy, *lpagents, *hpagents;
      int *no_agents, inter_time=0, pid, jobsinthesystem = FALSE, status;
      int *new_agent, s_time, i, x, j, time;

      /* shared memory allocation for the shared variables */

      agent = (struct _agent *)shmalloc(sizeof(struct _agent));
      lp_agents_busy = (int *)shmalloc(sizeof(int));
      hp_agents_busy = (int *)shmalloc(sizeof(int));
      no_agents = (int *)shmalloc(sizeof(int));
      new_agent = (int *)shmalloc(sizeof(int));
      hpagents = (int *)shmalloc(sizeof(int));
      lpagents = (int *)shmalloc(sizeof(int));
      current_time = (int *)shmalloc(sizeof(int));
      no_of_accesses = (int *)shmalloc(sizeof(int));
```

```
        *lp_agents_busy = FALSE;
        *hp_agents_busy = FALSE;
        *no_agents = 1;
        *lpagents = 0;
        *hpagents = 0;
        *current_time = 0;
        *no_of_accesses = 0;

        /* semaphores initialization */

        AG[0] = seminit((key_t)SEMKEY_0, 1);
        if(AG[0] == -1){
           printf("Semaphore initialization failed\n");
           exit(0);
        }
        AG[1] = seminit((key_t)SEMKEY_1, 1);
        if(AG[1] == -1){
           printf("Semaphore initialization failed\n");
           exit(0);
        }
        AG[2] = seminit((key_t)SEMKEY_2, 1);
        if(AG[2] == -1){
           printf("Semaphore initialization failed\n");
           exit(0);
        }
        AG[3] = seminit((key_t)SEMKEY_3, 1);
        if(AG[3] == -1){
           printf("Semaphore initialization failed\n");
           exit(0);
        }
        AG[4] = seminit((key_t)SEMKEY_4, 1);
        if(AG[4] == -1){
           printf("Semaphore initialization failed\n");
           exit(0);
        }
        AG[5] = seminit((key_t)SEMKEY_5, 1);
        if(AG[5] == -1){
           printf("Semaphore initialization failed\n");
           exit(0);
        }


initialize(agent);

/* execution continued till 200 accesses are processed */

while((*no_of_accesses < 200) || (jobsinthesystem)){
    while((*no_of_accesses < 200)&&(*current_time >= (prev_time +
                                                        inter_time))){
         jobsinthesystem = TRUE;

        /* if the access trying to enter is high priority access */

         if(((int)(7 * random() + 0)) == 0){

           /*if the high priority agent is free, allocate the agent by
             executing the P operation corresponding to that semaphore */

           if(*hp_agents_busy == FALSE){

             *hpagents += 1;
             if(*hpagents == HPAGENTS)
```

```
    *hp_agents_busy = TRUE;

   /* fork a child process. The child process does the required
      semaphore operations , the parent process continues further
      execution */

     if((pid = fork()) == -1){
       printf("Fork failed\n");
       exit(0);
     }

   /* child process starts execution. Free agent is found and
      P operation on the corresponding semaphore is executed */

     if(pid == 0){
        for(i = 0; i < 2; i++)
           if(agent->AGENT[i].status == SLEEP) break;
        P(AG[i]);
        agent->AGENT[i].access_id = *no_of_accesses;
        agent->AGENT[i].start_time = *current_time;
        agent->AGENT[i].status = BUSY;
        agent->AGENT[i].service_time = (int)(5*random() + 2);
        sleep(agent->AGENT[i].service_time);
        V(AG[i]);
        if(agent->AGENT[i].queue_size > 0)
           agent->AGENT[i].queue_size -= 1;
        else{
           agent->AGENT[i].status = SLEEP;
           *hpagents -= 1;
           *hp_agents_busy = FALSE;
        }
        exit(1);
      }
}

/* if a free lp agent is availabel, then this HP access is
allocated to it, after executing the P oparation on the
corresponding semaphore */

else if(*lp_agents_busy == FALSE){

   *lpagents += 1;
   if(*lpagents == LPAGENTS)
    *lp_agents_busy = TRUE;

   /* fork a child process. The child process does the required
      semaphore operations , the parent process continues further
      execution */

     if((pid = fork()) == -1){
       printf("Fork failed\n");
       exit(0);
     }
     if(pid == 0){
        for(i = 2; i < 6; i++)
           if(agent->AGENT[i].status == SLEEP) break;
        P(AG[i]);
        agent->AGENT[i].access_id = *no_of_accesses;
        agent->AGENT[i].start_time = *current_time;
        agent->AGENT[i].status = BUSY;
        agent->AGENT[i].service_time = (int)(5*random() + 2);
        sleep(agent->AGENT[i].service_time);
```

```
         V(AG[i]);
         if(agent->AGENT[i].queue_size > 0)
            agent->AGENT[i].queue_size -= 1;
         else{
            agent->AGENT[i].status = SLEEP;
            *lpagents -= 1;
            *lp_agents_busy = FALSE;
         }
         exit(1);
       }
  }

  /* else, the hp access is made to wait on a hp agent whose wait
     queue is minimum */

  else{

    /* fork a child process. The child process does the required
       semaphore operations , the parent process continues further
       execution */

      if((pid = fork()) == -1){
        printf("Fork failed\n");
        exit(0);
      }
      if(pid == 0){
        j = 0;
        for(i = 0; i < 2; i++){
          if(agent->AGENT[j].queue_size > agent->AGENT[i].
                                                       queue_size)
             j = i;
        }
        agent->AGENT[j].queue_size += 1;
        P(AG[j]);
        agent->AGENT[j].access_id = *no_of_accesses;
        agent->AGENT[j].start_time = *current_time;
        agent->AGENT[j].status = BUSY;
        agent->AGENT[j].service_time = (int)(5*random() + 2);
        sleep(agent->AGENT[j].service_time);
        V(AG[j]);
        if(agent->AGENT[j].queue_size > 0)
           agent->AGENT[j].queue_size -= 1;
        else{
           agent->AGENT[j].status = SLEEP;
           *hpagents -= 1;
           *hp_agents_busy = FALSE;
        }
        exit(1);
      }
  }
}

/* if the new access is of low priority access */

else{
    if(*lp_agents_busy == FALSE){

       /* find the free low priority agent, and if it available
          allow  the access to execute the corresponding semaphore
          */

       *lpagents += 1;
```

```
        if(*lpagents == LPAGENTS)
           *lp_agents_busy = TRUE;

        /* fork a child process. The child process does the required
           semaphore operations , the parent process continues
           further execution */

        if((pid = fork()) == -1){
           printf("Fork failed\n");
           exit(0);
        }
        if(pid == 0){
           for(i = 2; i < 6; i++)
              if(agent->AGENT[i].status == SLEEP) break;
           P(AG[i]);
           agent->AGENT[i].access_id = *no_of_accesses;
           agent->AGENT[i].start_time = *current_time;
           agent->AGENT[i].status = BUSY;
           agent->AGENT[i].service_time = (int)(5*random() + 2);
           sleep(agent->AGENT[i].service_time);
           V(AG[i]);
           if(agent->AGENT[i].queue_size > 0)
             agent->AGENT[i].queue_size -= 1;
           else{
             agent->AGENT[i].status = SLEEP;
           *lpagents -= 1;
           *lp_agents_busy = FALSE;
        }
        exit(1);
     }
  }

  /* the access is put in the wait queue of an agent whose queue
  size is minimum. */

  else{

    /* fork a child process. The child process does the required
       semaphore operations , the parent process continues further
       execution */

      if((pid = fork()) == -1){
        printf("Fork failed\n");
        exit(0);
      }
      if(pid == 0){
        j = 2;
        for(i = 2; i < 6; i++){
           if(agent->AGENT[j].queue_size > agent->AGENT[i].
                                                       queue_size)
              j = i;
        }
        agent->AGENT[j].queue_size += 1;
        P(AG[j]);
        agent->AGENT[j].access_id = *no_of_accesses;
        agent->AGENT[j].start_time = *current_time;
        agent->AGENT[j].status = BUSY;
        agent->AGENT[j].service_time = (int)(5*random() + 2);
        sleep(agent->AGENT[j].service_time);
        V(AG[j]);
        if(agent->AGENT[j].queue_size > 0)
           agent->AGENT[j].queue_size -= 1;
```

```
                else{
                   agent->AGENT[j].status = SLEEP;
                   *lpagents -= 1;
                   *lp_agents_busy = FALSE;
                }
                exit(1);
             }
         }
      }

      *no_of_accesses += 1;
      prev_time = *current_time;
      inter_time = (int)(5 * random() + 0);
   }

   *current_time += 1;

   if((agent->AGENT[0].status == FALSE) && (agent->AGENT[1].status ==
     FALSE) && (agent->AGENT[2].status == FALSE) &&
     (agent->AGENT[3].status == FALSE) && (agent->AGENT[4].status ==
     FALSE) && (agent->AGENT[5].status == FALSE))

       jobsinthesystem = FALSE;

   if(*current_time % 10 == 0){
      print_stat(agent, *current_time, *no_of_accesses);
      sleep(3);
   }
}

/* parent process waits till all the child processes exit */
for(i = 0; i < 100; i++) wait(&status);

/* kill all the semaphores which were initialized in the begining */

semkill(AG[0]); semkill(AG[1]); semkill(AG[2]); semkill(AG[3]);
semkill(AG[4]); semkill(AG[5]);

/* free the shared memory allocated */

shfree(agent); shfree(no_agents); shfree(lp_agents_busy);
shfree(hp_agents_busy); shfree(lpagents); shfree(hpagents);
shfree(current_time);
}

/*************************************************************************
                   FUNCTION TO GENERATE RANDOM NUMBER
*************************************************************************
This function is used to generate a randum number between 0 and 1. */

float random()
{
      long a = 16807.0;
      long M = 2147483647.0;
      long q = 127773.0, r = 2836.0, lo, hi, test;

      hi = (int)(seed/q);
      lo = seed - q*hi;
      test = a*lo-r*hi;
      if (test > 0.0)
            seed = test;
      else
```

```
            seed = test + M;
      return(seed/M);
}

/***********************************************************************
                     FUNCTION TO INITIALIZE VARIABLES
************************************************************************

This function is used to initialize variables used in the program. */

struct _agent *initialize(agents)
struct _agent *agents;
{
int i =0;
for(i = 0; i < MAX_AGENTS; i++){
   agents->AGENT[i].status = SLEEP;
   agents->AGENT[i].start_time = 0;
   agents->AGENT[i].service_time = 0;
   agents->AGENT[i].queue_size = 0;
   agents->AGENT[i].access_id = NULL;
}
for(i = 0; i < HPAGENTS; i++)
   strcpy(agents->AGENT[i].priority, "HP");
for(i = HPAGENTS; i < MAX_AGENTS; i++)
   strcpy(agents->AGENT[i].priority, "LP");
return(agents);
}

/***********************************************************************
                    FUNCTION TO PRINT THE QUEUE SIZES
************************************************************************
This function is used to display on monitor screen, the sizes of the
various queues at a particular time. */

print_stat(agents, time, accesses)
struct _agent *agents;
int time, accesses;
{

      int i, j;

      system("tput clear");
      printf("                   TIME IS %d (ACCESSES %d)\n\n\n", time,
                                                              accesses);
      for(i = 0; i < MAX_AGENTS; i++){
          printf("SYNCHRONIZING AGENT #%d :", i);
        for(j = 0; j < agents->AGENT[i].queue_size; j++)
          printf("*");
        if(agents->AGENT[i].status == SLEEP)
          printf("            SLEEP");
        printf("\n");
        }
```

```
                    /* sem.h header file */

/* This header file is a collection of routines for using semaphores in C:
      1. seminit - to initialize a semaphore.
      2. P - to perform P(S) operation.
      3. V - to perform V(S) operation.
      4. semkill - to remove a semaphore.
   These routines call UNIX System V system routines:
      1. semget - to get a semaphore
      2. semctl - semaphore control operations.
      3. semop - semaphore operations.
*/

#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

struct semun{
      int val;
      struct semid_ds *buf;
      char *array;
}arg;
static void semkill();
static void P();
static void V();
static void semcall();

/* Initializing semaphore based on "key" parameter to "initval" */

static int seminit(key, initval)
key_t key;
int initval;
{
      int sid;
      struct semun semun;

     /* get the semaphore id based on the key */

      if((sid = semget(key, 1, 0600 | IPC_CREAT)) == -1)
            perror("semget");
      else{
          printf("SEMAPHORE ID (sid) = %d\n", sid);

          /* initializing semaphore to its initial value */

          semun.val = initval;
          if(semctl(sid, 0, SETVAL, semun) == -1)
            perror("semctl");
      }
      return(sid);
}

/* removing semaphore with id (sid) from the system */

static void semkill(sid)
int sid;
{
      if(semctl(sid, 0, IPC_RMID, 0) == -1)
          perror("semctl(kill)");
      printf("SEMAPHORE with value of sid = %d is killed\n", sid);
}
```

```
/* Performs the P operation on the semaphore. This should be called upon
entry to the critical section. */

static void P(sid)
int sid;
{

      semcall(sid, -1);

}

/* Performs the V operation on the semaphore. This should be called upon
exit from the critical section. */

static void V(sid)
int sid;
{

      semcall(sid, 1);

}

/* Performs the designated "op" operation on the semaphore. If "op" is -1,
then this implements the P operation; it decrements the value of the
semaphore if it is > 0, or is blocked if = 0. If "op" is 1, then V
operation is implemented; 1 is added to the current value of the
semaphore. */

static void semcall(sid, op)
int sid, op;
{

      struct sembuf sb;

      sb.sem_num = 0;
      sb.sem_op = op;
      sb.sem_flg = 0;
      if(semop(sid, &sb, 1) == -1)
          perror("semop");
```

```
/* windows.h header file */

#include <X11/Xlib.h>
#include <X11/Xutil.h>


      Display *mydisplay;
      Drawable mywindow, mys;
      Pixmap pixmap;
      GC mygc;
      XEvent myevent;
      XSizeHints myhints;
      int myscreen;
      int done = 0, time = 10;
      unsigned long myforeground, mybackground;
```

```
                /* program env1.c starts here */

/* In this program, two processes are forked, and the shared variable is
updated (incremented) 10 times in the loop. This loop is run for 100 times
and the final value of the shared variable after each run is printed.
Process one sleeps between read and write of the shared variable and
process 2 sleeps only before read operation of the shared variable. The
processes are put to sleep (passive wait) for a randum amount of time. */

#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<signal.h>

#define MAX_TIMES 10
#define MAX_PROC 2

float random();
double drand48();
double seed = 2.0;

/* main program starts here */

main()
{

      FILE *ofp;
      int i = 0, k, *x , j, pid, status, y;
      float rand_num;


      ofp = fopen("OUT1", "w");

      /* program is run for 100 times */

      for(k = 0; k < 100; k++){
            x = (int *)shmalloc(sizeof(int));
            *x = 0;

            /* MAX_PROC times processes are forked to update the shared
               variable */

            for(i = 0; i < MAX_PROC; i++){

                  /* if fork fails, it prints an error message and exits
                     the program */

                  if((pid = fork()) == -1){
                        printf("Fork failed\n");
                        exit(0);
                  }

                  /* child process does all the processing */

                  if(pid == 0){
                        switch(i){
                              case 0:
                                    /*process 0 sleeps between read and
                                      write of the shared variable */
                                    srand48(getpid());
                                    for(j = 0; j < MAX_TIMES; j++){
                                          y = *x;
```

```
                                        rand_num = drand48();
                                        sleep(rand_num);
                                        *x = y + 1;
                                }
                                break;
                        case 1:
                                /* process 1 sleeps before read */
                                srand48(getpid());
                                for(j = 0; j < MAX_TIMES; j++){
                                        rand_num = drand48();
                                        sleep(rand_num);
                                        y = *x;
                                        *x = y + 1;
                                }
                                break;
                        }
                        exit(0);
                }
        }

        /* parent process waits for the child processes to terminate
        */
        for(i = 0; i < 2; i++)
                wait(&status);
        fflush(stdout);
        fprintf(ofp,"The final value is %d in loop %d.\n\n", *x, k+1);
        shfree(x);
    }
    fclose(ofp);
}
```

```
                /* program env2.c starts here */

/* In this program, two processes are forked, and the shared variable is
incremented 10 times in the loop. This loop is run for 100 times and the
final value is printed. Process one sleeps before reading the shared
variable, whereas process2 sleeps between read and write of the shared
variable. The processes are put to sleep for a randum amount of time. */

#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<signal.h>

#define MAX_TIMES 10
#define MAX_PROC 2

float random();
double drand48();
double seed = 2.0;

/* main program starts here */

main()
{

      FILE *ofp;
      int i = 0, k, *x , j, pid, status, y;
      float rand_num;

      ofp = fopen("OUT2", "w");

      /* program is run for 100 times */

      for(k = 0; k < 100; k++){

            x = (int *)shmalloc(sizeof(int));
            *x = 0;

            /* MAX_PROC times processes are forked to update the shared
               variables */

            for(i = 0; i < MAX_PROC; i++){

                  /*if fork fails it prints an error message and exists */

                  if((pid = fork()) == -1){
                        printf("Fork failed\n");
                        exit(0);
                  }

                  /* child process does all the processing */

                  if(pid == 0){
                        switch(i){
                              case 0:
                                    /* process 0 sleeps before read
                                       operation of the shared variable */
                                    srand48(getpid());
                                    for(j = 0; j < MAX_TIMES; j++){
                                          rand_num = drand48();
                                          sleep(rand_num);
                                          y = *x;
```

```
                                        *x = y + 1;
                                }
                                break;
                        case 1:
                                /* process 1 sleeps between read and
                                   write operations */
                                srand48(getpid());
                                for(j = 0; j < MAX_TIMES; j++){
                                        y = *x;
                                        rand_num = drand48();
                                        sleep(rand_num);
                                        *x = y + 1;
                                }
                                break;
                        }
                        exit(0);
                }
        }

        /* parent process waits for child processes to terminate */

        for(i = 0; i < 2; i++)
                wait(&status);
        fflush(stdout);
        fprintf(ofp,"The final value is %d in loop %d.\n\n", *x, k+1);
        shfree(x);
    }
    fclose(ofp);
}
```

```
                /* program env3.c starts here */

/* In this program 60 processes are forked, and the shared variable is
updated (incremented) 10 times in a loop. This loop is run for 1000 times
and the final value of the shared variable is printed after every run.
Each process sleeps between read and write of the shared variable, for a
randum amount of time. This is achieved by making the process to execute
a for loop which does nothing for a randum number of times (active wait).
*/

#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<signal.h>

#define MAX_TIMES 10
#define MAX_PROC 60

double drand48();
double seed = 2.0;

/* main program starts here */

main()
{

      FILE *ofp;
      int i = 0, k, *x , j, pid, status, y,l, m, rep[8000];
      float rand_num;

      ofp = fopen("OUT1", "w");
      for(i = 0; i < 8000; i++)
            rep[i] = 0;

      /* program is run for 1000 times */

      for(k = 0; k < 1000; k++){
            x = (int *)shmalloc(sizeof(int));
            *x = 0;

            /* MAX_PROC times processes are forked */

            for(i = 0; i < MAX_PROC; i++){
                  if((pid = fork()) == -1){
                        printf("Fork failed for %d call\n", i);
                        exit(0);
                  }
                  /* child proceess does the processing */
                  if(pid == 0){
                        srand48(getpid());
                        for(j = 0; j < MAX_TIMES; j++){
                              y = *x;
                              rand_num = drand48();
                              m = (int)(rand_num * 100000);
                              /* every process does nothing for cetain
                                 randum amount of time */
                              for(l = 0; l < m; l++);
                              *x = y + 1;
                        }
                        exit(0);
                  }
            }
```

```
        /* parent process waits for all child processes to terminate
        */

        for(i = 0; i < MAX_PROC; i++)
              wait(&status);
        rep[*x] += 1;
        fflush(ofp);
        printf("The final value is %d in loop %d and %d times.\n", *x,
                                                  k+1, rep[*x]);
        shfree(x);
    }

    /* the number of times each value obtained is printed */

    printf("\n\nFinal value              #times generated\n");
    for(i = 0; i < 8000; i++){
        if(rep[i] != 0){
              printf("%-6d %-30d\n", i, rep[i]);
        }
    }
    fclose(ofp);
}
```

```
                /* program env4.c starts from here */

/* In this program 60 processes are forked, and the shared variable is
updated (incremented) 10 times in the loop. This loop is run for 1000
times and the final value of the shared variable is printed after every
run. All even numbered processes sleep between read and write of the
shared variable and all odd numbered processes sleep before reading the
shared variable, for a randum amount of time. This is achieved by making
the processes to execute a for loop which does nothing for a randum number
of times (active wait). */

#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<signal.h>
#define MAX_TIMES 10
#define MAX_PROC 60

double drand48();
double seed = 2.0;

/* main program starts here */

main()
{
	int i = 0, k, *x , j, pid, status, y,l, m, rep[6000];
	float rand_num;

	for(i = 0; i < 6000; i++)
		rep[i] = 0;

	/* program is run for 1000 times */

	for(k = 0; k < 1000; k++){
		x = (int *)shmalloc(sizeof(int));
		*x = 0;

		/* MAX_PROC times processes are forked */

		for(i = 0; i < MAX_PROC; i++){
			if((pid = fork()) == -1){
				printf("Fork failed for %d call\n", i);
				exit(0);
			}
			/* child process does the processing */
			if(pid == 0){
				switch(i % 2 ){
					case 0:
						/* even numbered processes wait
						   between read and write operations*/
						srand48(getpid());
						for(j = 0; j < MAX_TIMES; j++){
							y = *x;
							rand_num = drand48();
							m = (int)(rand_num * 100000);
							for(l = 0; l < m; l++);
							*x = y + 1;
						}
						break;
					case 1:
						/* odd numbered processes wait before
						   reading the shared variable */
```

```
                    srand48(getpid());
                    for(j = 0; j < MAX_TIMES; j++){
                          rand_num = drand48();
                          m = (int)(rand_num * 100000);
                          for(l = 0; l < m; l++);
                          y = *x;
                          *x = y + 1;
                    }
                    break;
                }
                exit(0);
            }
        }

        /* parent process waits for child processes to terminate */

        for(i = 0; i < MAX_PROC; i++)
              wait(&status);
        rep[*x] += 1;
        fflush(stdin);
        printf("The final value is %d in loop %d and %d times.\n", *x,
                                                      k+1, rep[*x]);
        shfree(x);
    }

    /* number of times each value obtained is printed */

    printf("\n\nFinal value             #times generated\n");
    for(i = 0; i < 6000; i++){
          if(rep[i] != 0){
                printf("%6d %30d\n", i, rep[i]);
          }
    }
}
```

VITA

Raveendra R. Avutu

Candidate for the Degree of

Master of Science

Thesis: A GENERAL MUTUAL EXCLUSION PRIMITIVE

Major Field: Computer Science

Biographical:

Personal Data: Born in Vallabhapuram, India, July 10, 1962, son of Mr. and Mrs. A. R. Reddy.

Education: Received Bachelor of Engineering in Mechanical Engineering from University of Allahabad, Allahabad, India, in July 1984; completed requirements for the Master of Science Degree at Oklahoma State University in December 1993.

Professional Experience: Was Graduate Research Assistant in University Computer center at Oklahoma State University, from January 1992 to August 1993. Worked as Assistant Manager for Production with Andhra Polymers Ltd., Hyderabad, India, from January 1989 to July 1991. From May 1986 to December 1988, worked as Assistant Engineer for W.I. Enterprises Ltd., Pune, India. Was Scientist 'B' in Defence Research and Development, Department of Defence, India, from September 1984 to April 1986.