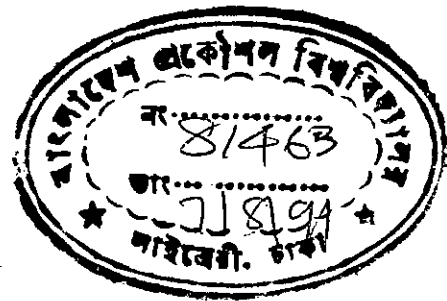
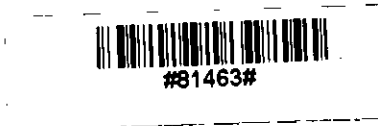


**PERFORMANCE ANALYSIS OF MULTIPROCESSORS SHARING MEMORY  
MODULES CONNECTED BY MULTIBUS NETWORK**

by

**MAHBOOB HASAN CHOWDHURY**



**A Thesis**

Submitted to the Department of Computer Science and Engineering , Bangladesh University of Engineering and Technology, Dhaka, in partial fulfilment of the requirements for the degree

of

**MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING**

**February , 1991**

**Bangladesh University of Engineering and Technology , Dhaka.**

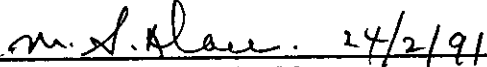
PERFORMANCE ANALYSIS OF MULTIPROCESSORS SHARING  
MEMORY MODULES CONNECTED BY MULTIBUS NETWORK

A Thesis

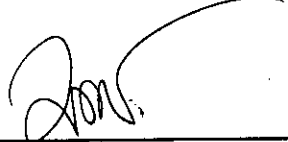
by

MAHBOOB HASAN CHOWDHURY


Approved as to style and contents by:

  
\_\_\_\_\_  
Dr. Md. Shamsul Alam  
Associate Professor  
Department of Computer Science and  
Engineering, BUET, Dhaka, Bangladesh

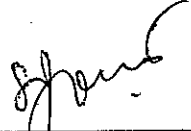
Chairman  
and  
Supervisor

  
\_\_\_\_\_  
Dr. Syed Mahbubur Rahman  
Associate Professor and Head  
Department of Computer Science and  
Engineering, BUET, Dhaka, Bangladesh

Member

  
\_\_\_\_\_  
Dr. Md. Kaykobad  
Assistant Professor  
Department of Computer Science and  
Engineering, BUET, Dhaka, Bangladesh

Member

  
\_\_\_\_\_  
Dr. Md. Mujibur Rahman  
Professor  
Department of Electrical and electronic  
Engineering, BUET, Dhaka, Bangladesh

Member  
(External)

## ABSTRACT

With the availability of high hardware parallelism through fast physical devices, the interconnection between processor and memory modules is required to be efficient enough for high performance of a multiprocessor system. For higher performance multiple bus interconnection can be used.

In this thesis work multiple bus interconnection is used for processor memory interconnection in multiprocessor system. Multiple bus connection is fault tolerant and during a bus fault only system performance decreases by a little amount and there remain paths to every memory module from each of the processors. Equal priority, unequal priority and a combination of unequal priority and random delay protocols are used for resolving bus and memory conflicts. Both synchronous and asynchronous timing and packet switched and circuit switched systems are simulated for performance analysis. For performance analysis parameters used are average queue length, processor utilization, memory bandwidth and bus utilization. Hardware design of synchronous and asynchronous arbiters are presented. In synchronous design equal priority protocol is assumed. In asynchronous system a two level unequal priority protocol is examined. Simulation result is validated by analytical solutions.

## ACKNOWLEDGEMENT

Profound knowledge and keen interest of Dr. Md. Shamsul Alam in the field of Multiprocessor System has influenced the author to carry out a research work in this field. This research work was done under his supervision. His constant guidance, supervision, suggestion at all the stages of this research have made it possible to complete this thesis. The author expresses his sincere gratitude to Dr. Alam.

The author takes the opportunity to express his heartfelt gratitude and thanks to Dr. Md. Kaykobad and Mr. Mahmood Hasan Chowdhury of Computer Science and Engineering Department, BUET, for their sincere cooperation and providing valuable materials for this research.

The author is indebted to Dr. Syed Mahbubur Rahman, Head CSE Department, Mr. Md. Musa, Director, Institute of Computer Science(ICS), BAEC and Mr. Alamgir Sarker, Senior Scientific officer, ICS, BAEC for their constant inspiration during the work.

The all-out support and services rendered by the faculty members and the staff of the department of Computer Science and Engineering, BUET are also acknowledged with sincere thanks.

DECLARATION

I do hereby declare that neither this thesis nor anypart thereof has been submitted or is being currently submitted in candidature for any degree at any other university.

*M. D. D. D.*

-----  
Candidate

## TABLE OF CONTENTS

	Page
Title Page	i
Abstract	ii
Acknowledgment	iii
Declaration	iv
Table of Contents	v
List of Figures	viii
List of Symbols	xlv
CHAPTER 1 INTRODUCTION	
1.1 General Description	1
1.2 Processor Characteristics for Multiprocessing	6
1.3 Interconnection Networks	11
1.4 Parallel Memory Organizations	15
1.5 Operating System Requirement for Multiprocessors	18
1.6 Some Examples of Multiprocessor System	19
CHAPTER 2 ARBITRATION	
2.1 General Description	25
2.2 Circuit Switched Asynchronous System	28
2.3 Circuit Switched Synchronous System	27

2.4	Packet Switched Asynchronous System	28
2.5	Packet Switched Synchronous System	29
2.6	Arbiter Design	31
CHAPTER 3	ANALYTICAL METHODS	
3.1	General Description	42
3.2	Semi-Markov Method	44
3.3	Probabilistic Method	49
CHAPTER 4	SIMULATION	
4.1	General Description	50
4.2	Development of Simulation Software	53
4.3	Circuit Switched System	55
4.4	Packet Switched System	57
CHAPTER 5	RESULTS AND DISCUSSION	63
CHAPTER 6	CONCLUSIONS AND SUGGESTIONS	
6.1	Conclusion	109
6.2	Suggestions for Further Research	111
APPENDICES		
A1:	Simulation Program of Asynchronous Circuit Switched System for Random Delay Protocol	113

A2: Simulation Program of Synchronous Circuit Switched System for Equal Priority Protocol	123
A3: Simulation Program of Synchronous Packet Switched System for Equal priority Protocol	138
A4: Simulation Program of Asynchronous Packet Switched System for Equal priority Protocol	154

REFERENCES

168





## LIST OF FIGURES

Figure 1.1	A loosely coupled multiprocessor system	2
Figure 1.2	A tightly coupled multiprocessor system	4
Figure 1.3	Multiprocessor system with a single shared bus	10
Figure 1.4	Multiple bus multiprocessor system	10
Figure 1.5	Multiprocessor system with crossbar interconnection network	10
Figure 1.6	A 2*2 switch	14
Figure 1.7	Omega network	14
Figure 2.1	Logic diagram for memory arbitration module	32
Figure 2.2	Logic diagram for bus arbitration module	33
Figure 2.3	Logic diagram for 2 to 1 arbiter	36
Figure 2.4	Logic diagram for T1 module of 8 to 4 arbiter	36
Figure 2.5	Block diagram of 8 to 4 arbiter	38
Figure 2.6	Logic diagram for T2 module of 8 to 4 arbiter	39
Figure 2.7	Logic diagram for T2 module of 4 to 2 arbiter	39
Figure 3.1	Queueing diagram of circuit switched system	43
Figure 3.2	Queueing diagram of packet switched multi processor system	43

Figure 3.3	Semi-Markov model of circuit switched synchronous system	45
Figure 3.4	Semi-Markov model of circuit switched synchronous system without residual waiting time	45
Figure 4.1	Components and queue used in simulation.	52
Figure 4.2	Flowchart of simulation of circuit switched asynchronous system.	55
Figure 4.3	Flowchart of simulation of packet switched asynchronous system.	60
Figure 5.1.1	Average Queue Length Vs. Number of Buses for asynchronous circuit switched System(unequal priority protocol).	64
Figure 5.1.2	Processor Utilization Vs. Number of Buses for asynchronous circuit switched system (unequal priority protocol).	65
Figure 5.1.3	Memory Bandwidth Vs. Number of Buses for asynchronous circuit switched system(unequal priority protocol)	66
Figure 5.1.4	Bus Utilization Vs. Number of Buses for asynchronous circuit switched system(unequal priority protocol).	67

Figure 5.2.1	Average Queue Length Vs. Number of Buses for asynchronous circuit switched system ( random delay protocol).	68
Figure 5.2.2	Processor Utilization Vs. Number of Buses for asynchronous circuit switched system ( random delay protocol).	69
Figure 5.2.3	Memory Bandwidth Vs. Number of Buses for asynchronous circuit switched system ( random delay protocol)	70
Figure 5.2.4	Bus Utilization Vs. Number of Buses for asynchronous circuit switched system ( random delay protocol).	71
Figure 5.3.1	Average Queue Length Vs. Number of Buses for synchronous circuit switched system(equal poriority protocol).	72
Figure 5.3.2	Processor Utilization Vs. Number of Buses for synchronous circuit switched system (equal priority protocol).	73
Figure 5.3.3	Memory Bandwidth Vs. Number of Buses for synchronous circuit switched system (equal priority protocol)	74

Figure 5.3.4	Bus Utilization Vs. Number of Buses for synchronous circuit switched system(equal priority protocol).	75
Figure 5.4.1	Average Queue Length Vs. Number of Buses for synchronous circuit switched system(unequal priority protocol).	76
Figure 5.4.2	Processor Utilization Vs. Number of Buses for asynchronous circuit switched system (unequal priority protocol).	77
Figure 5.4.3	Memory Bandwidth Vs. Number of Buses for synchronous circuit switched system (unequal priority protocol)	78
Figure 5.4.4	Bus Utilization Vs. Number of Buses for synchronous circuit switched system(unequal priority protocol).	79
Figure 5.5.1	Average Queue Length vs Number of Buses for synchronous packet switched system(equal priority protocol).	80
Figure 5.5.2	Processor Utilization Vs. Number of Buses for synchronous packet switched system (equal priority protocol).	81

Figure 5.5.3	Memory Bandwidth Vs. Number of Buses for synchronous packet switched system(equal priority protocol)	82
Figure 5.5.4	Bus Utilization Vs. Number of Buses for synchronous packet switched system(equal priority protocol).	83
Figure 5.6.1	Average Queue Length Vs. Number of Buses for asynchronous packet switched system( equal priority protocol).	84
Figure 5.6.2	Processor Utilization Vs. Number of Buses for asynchronous packet switched system (equal priority protocol).	85
Figure 5.6.3	Memory Bandwidth Vs. Number of Buses for asynchronous packet switched system(equal priority protocol)	86
Figure 5.6.4	Bus Utilization Vs. Number of Buses for asynchronous packet switched system(equal priority protocol).	87
Figure 5.7.1	Average Queue Length Vs. Number of Buses for synchronous circuit switched system with variable number of processors.	88

Figure 5.7.2	Processor Utilization Vs. Number of Buses for synchronous circuit switched system with variable number of processors.	89
Figure 5.7.3	Memory Bandwidth Vs. Number of Buses for synchronous circuit switched system with variable number of processors.	90
Figure 5.7.4	Bus Utilization Vs. Number of Buses for synchronous circuit switched system with variable number of processors.	91
Figure 5.8	Memory Bandwidth vs Number of Buses for synchronous circuit switched system (equal priority protocol).	92
Figure 5.9	Memory Bandwidth Vs. Number of Buses for synchronous circuit switched system(by analysis).	93
Figure 5.10.1	Average Queue Length Vs. Number of Buses for synchronous circuit switched system	94
Figure 5.10.2	Processor Utilization Vs. Number Buses for asynchronous circuit switched system	95
Figure 5.10.3	Memory Bandwidth Vs. Number of Buses for asynchronous circuit switched system	96

Figure 5.10.4	Bus Utilization Vs. Number of Buses for asynchronous circuit switched system	97
Figure 5.11.1	Average Queue Length Vs. Number of Buses for synchronous circuit switched system	98
Figure 5.11.2	Processor Utilization Vs. Number of Buses for asynchronous circuit switched system	99
Figure 5.11.3	Memory Bandwidth Vs. Number of Buses for asynchronous circuit switched system	100
Figure 5.11.4	Bus Utilization Vs. Number of Buses for asynchronous circuit switched system	101

## List of Symbols

Symbol	Meaning
$N$	Number of processors
$M$	Number of memory modules
$B$	Number of buses
$R_i$	Request from processor $i$
$M_{Si}$	Status of memory module $i$
$B_{Sk}$	Status of bus $k$
$G_i$	Grant signal for processor $i$
$G_P$	Primary grant signal
$G_S$	Secondary grant signal
$t_{tot}$	Total simulation time
$Q_a$	Average queue length
$B_w$	Memory bandwidth
$P_u$	Processor utilization
$B_u$	Bus utilization
$d$	Nominal gate delay
$r$	request rate



CHAPTER 1

INTRODUCTION



1.1 General Description:

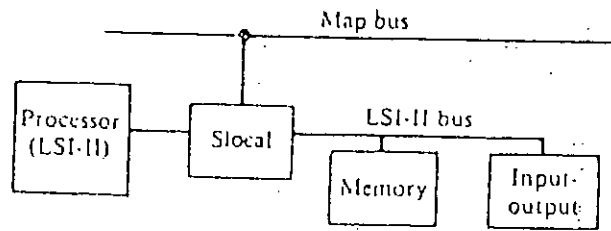
Numerous applications require ever increasing computing power which is not possible to gain from sequential computers [1]. Introduction of parallelism in computer can increase this ability to a great extent. It is possible to introduce parallelism by pipelining and by using multiprocessor machine. Then efficient algorithm for a large class of problem can be developed by exploiting parallel hardware feature of multiprocessor system. So improved performance of multiprocessor system is necessary. Multiprocessors can be grossly characterized by two attributes:

(a) A multiprocessor is a single computer including multiple processors.

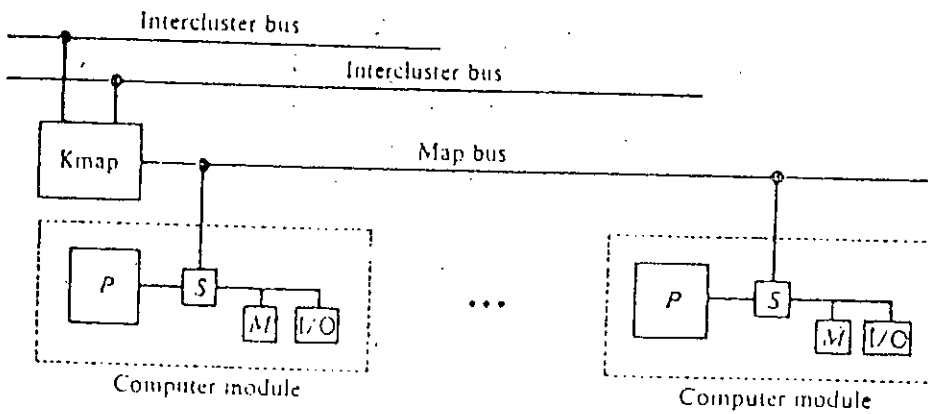
(b) Processors may communicate and co-operate at different levels in solving a given problem. The communication may occur by sending messages from one processor to the other or by sharing a common memory.

Multiprocessor system can be divided into two architectural models, such as, tightly coupled multiprocessors and loosely coupled multiprocessors [2].

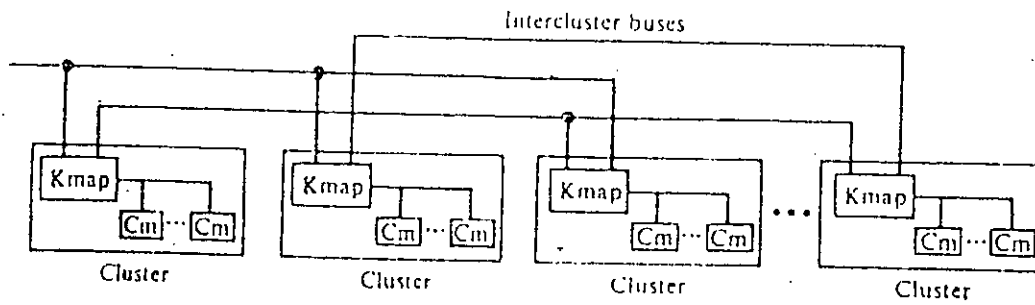
Tightly coupled multiprocessors communicate through a shared main memory. So the rate at which data can communicate from one processor to other depends on bandwidth of the memory. A high speed local memory or cache memory may exist with each



(a) A computer module



(b) A cluster of computer modules



(c) A network of clusters

Figure 1.1

: A loosely Coupled multiprocessor System. [ 2 ]

processor. A complete connectivity can exist between the processors and main memory. This connectivity can be accomplished by interconnection network- crossbar network, time shared bus, multiple bus network or by a multiported memory. One of the limiting factors to the expansion of a tightly coupled system is the performance degradation due to memory contentions which occur when two or more processors wish to access particular memory module simultaneously. Another limiting factor is processor memory interconnection network.

Loosely coupled multiprocessor systems do not generally encounter the same degree of memory conflicts experienced by tightly coupled system. In loosely coupled multiprocessor system each processor has a set of input output devices and a large local memory where it accesses for most of the instructions and data. Here processor, its local memory and I/O interfaces are known as computer module. Processor which execute on different computer modules communicate by exchange messages through a message transfer system(MTS) [3]. The degree of coupling of such a system is very loose. Hence, this type of system is known as distributed system. Figure 1.1 shows the loosely coupled multiprocessor system(LCS)  $C_m^*$  [2]. Each computer module of the  $C_m^*$  includes a local switch Slocal. Slocal routes the processor's requests to the memory and I/O devices outside the computer module via map bus. It also accepts references from other computer modules to its local memory and I/O devices. The Kmap is a processor that is responsible for mapping addresses and routing data between Slocals. The computer modules are connected

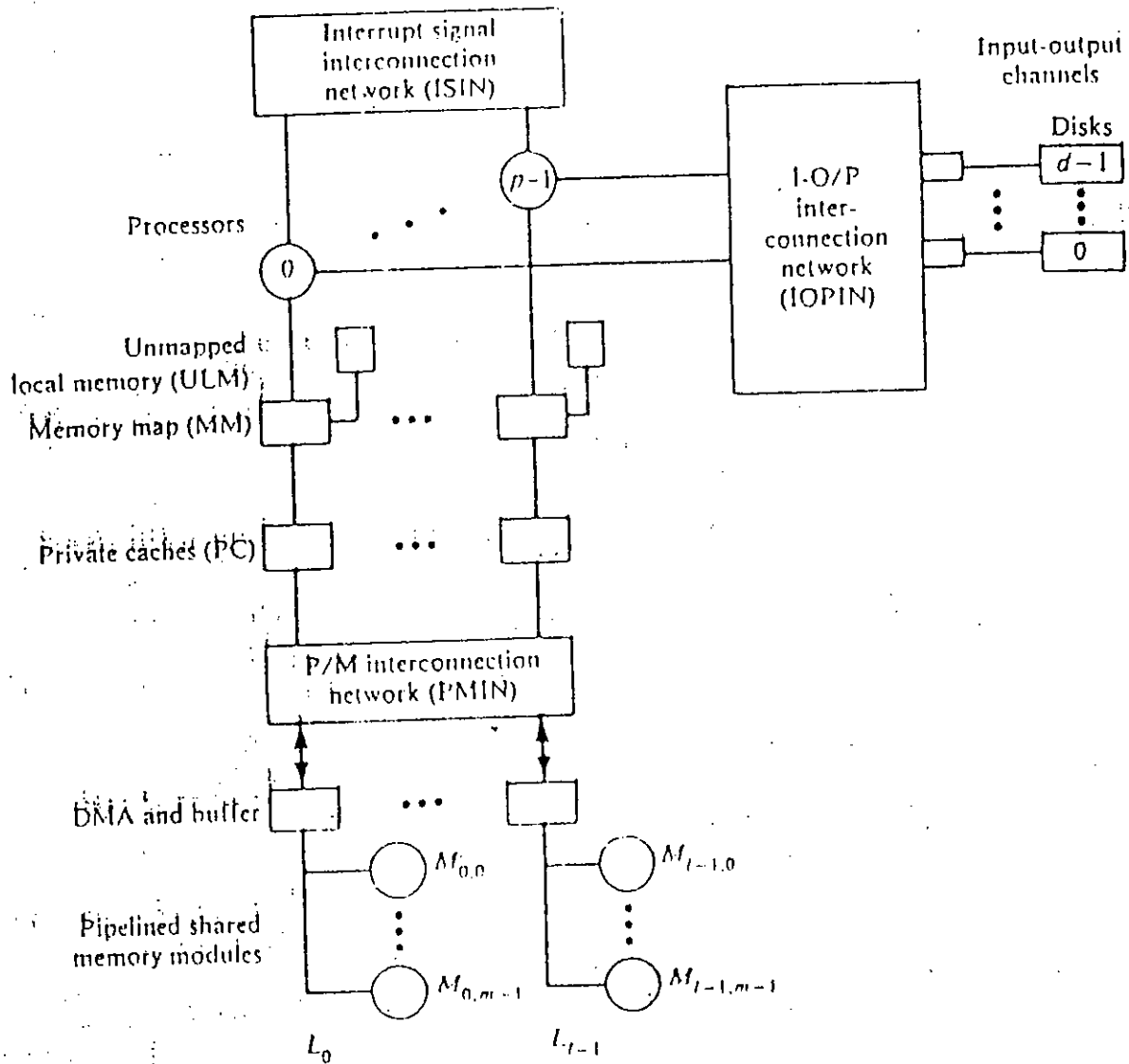


Figure 1.2 : A tightly Coupled multiprocessor system. [2]

in hierarchical clusters by two level buses. A cluster consists of computer module, Kmap and map bus. Clustering can enhance the cooperative ability among the processors of a cluster. But intercluster communication becomes time consuming. The map bus may create a bottleneck because only one transaction can take place at a time. Clusters communicate via intercluster buses.

Because of large variability of interference times, the throughput of loosely coupled multiprocessor may be too low for applications which require high response times. If high speed processing is desired, tightly coupled system (TCS) may be used. Figure 1.2 shows a tightly coupled system.

In this figure processor memory inter-connection network (PMIN) can be multiple bus or cross bar switch etc. and its efficiency is required for good system performance [2]. Here each processor references main memory and these memory references contributes to the memory conflicts at the memory modules [4]. Since each memory reference goes through the PMIN, it encounter delay in the processor memory switch and hence the instruction cycle time increases. This delay can be reduced by associating a cache with each processor to capture most of the references made by a processor. Another consequence of the cache is that the traffic through the connecting network can be reduced. With cache there is a problem called cache coherence [5]. More than one inconsistent copy of data can exist in the system. When there is a cache miss the required block can be directly found in any one of shared memory modules, or it can be currently in other processor's cache memory. Then copy back

operation is required.

## 1.2 Processor Characteristics for Multiprocessing [2]:

Most multiprocessors have been built using processors not originally designed for multiprocessor architecture. Examples of these are the C.mmp system which used DEC'S PDP-11 processors and Cm\* which used LSI-11 microprocessors. So a number of desirable architectural features are necessary for these processors and these are described as follows :

(a) **Process recoverability:** The architecture of a processor used in multiprocessor system reflect fact that the process and the processor are two different entities. If the processor fails, it should routinely be possible for another processor to retrieve the interrupted process state so that execution of process continues. Without this feature, the potential for reliability is substantially reduced. Most processors contain the process state of the current running process in internal register which are not accessible outside the processor and are not written to memory in the event of fault. With current technology, it should be possible to separate the general purpose registers from processor itself without much loss of speed. It is desired to have register file shared by all the processors.

(b) **Efficient context switching:** Another reason for a shared general purpose register is that a large register file can be used in a multiprogrammed processor. For effective utilization, it is necessary for the processor to support more

than one addressing domain and hence to provide domain change or context switching operation. Such switching requires extensive queuing and stack operations. The context switch operation saves the state of the current process and then switches to a selected ready-to-run process by restoring the state of the new process. The state of the new process is indicated by the contents of the process registers. An example of a processor with multiple domain is the IBM 370/168 . Two domains, the supervisor and user modes of operation are available. A user process can communicate with the operating system by using a mechanism provided through a supervisor call (SVC) instruction. A special instruction can be created to accomplish the context switch efficiently. An example of such an instruction is the central exchange jump (CEJ) in the Cyber-170 processor, which contains a single set of registers. The execution of the CEJ results in the saving of the context or state of the current process and the register set replaced by the state of another process taken from an area of central memory. This area is called the exchange package.

(c) Large virtual and physical address space: A processor intended to be used in the construction of a general-purpose medium to large multiprocessor must support a large physical address space. Even when an algorithm is decomposed so that it can be implemented using very small amount of code, processes sometimes need to access large amount of data object. The 16 bit address space used in C.mmp hampered effective programming of the system. In addition to the need for a large physical address space, a large virtual address space is also

desirable. If possible virtual address space should be segmented to promote modular sharing and checking of address bounds for memory protection and software reliability. For example, each processor used in the S-1 multiprocessor system has 2 gigabytes of virtual memory and 1 gigabyte of physical memory where each word is 36 bits wide .

(d) **Effective synchronization primitives:** The processor design must provide some implementation of indivisible actions which serve as the basis of synchronization primitives. These synchronization primitives require efficient mechanisms for establishing mutual exclusion. Mutual exclusion is required when two or more processors are in execution concurrently and must cooperate to exchange data during the computation. Mechanisms for establishing mutual exclusion involve some kind of read-modify-write memory cycle and queueing. One such mechanism is the semaphore. Each semaphore has a queue associated with it and the entries in the queue refer to processes which were suspended because of the semaphore value of the variable. A semaphore operation requires an indivisible operation, which can be accomplished by read-modify-write memory cycle to test and update a semaphore. The queue manipulations should also be done indivisibly. Some instructions which are used to accomplish mutual exclusion, such as, the test-and-set and compare-and-swap.

(e) **Interprocessor communication mechanism:** The set of processors used in a multiprocessor must have an efficient means of interprocessor communication. This mechanism should be implemented in hardware. A hardware mechanism is very useful for



drawing the attention of the target processor. The need for such a mechanism is even more apparent, when, in a asymmetric multiprocessor system, there are frequent requests for services exchanged between different processors. The hardware interprocessor mechanism can also facilitate synchronization between processors. This mechanism could, for example be used in the event of a processor failure to initiate a hardware signal to all functioning processors, which would then become aware of the faulty processor and start an error recovery or diagnostic procedure. Since the processors in a tightly coupled system share memory, it is possible to have software interprocessor communication without an explicit hardware mechanism. This method is inefficient as each processor will have to periodically poll its "mailbox" to see if there is a message for it. Such polling will result in intolerable response times for a large number of processors. Examples of systems with hardware inter-processor communication mechanisms are IBM 370/168 MP, Cray X-MP, and the C.mmp. It is possible that two or more processors may simultaneously attempt to access a common path in the interprocessor mechanism. Each processor must be capable of participating in the arbitration of the requests to use the path. Since arbitration implies that on simultaneous requests one or more processors must wait, the processors must have a wait state or some mechanism to suspend the processor in a queue.

(f) **Instruction set:** The instruction set of the processor should have adequate facilities for implementing high level languages that permit effective concurrency at the

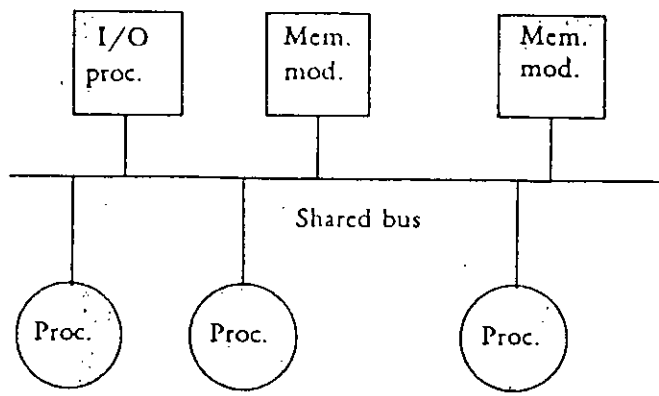


Figure 1.3 : Multiprocessor system with a single shared bus. [2]

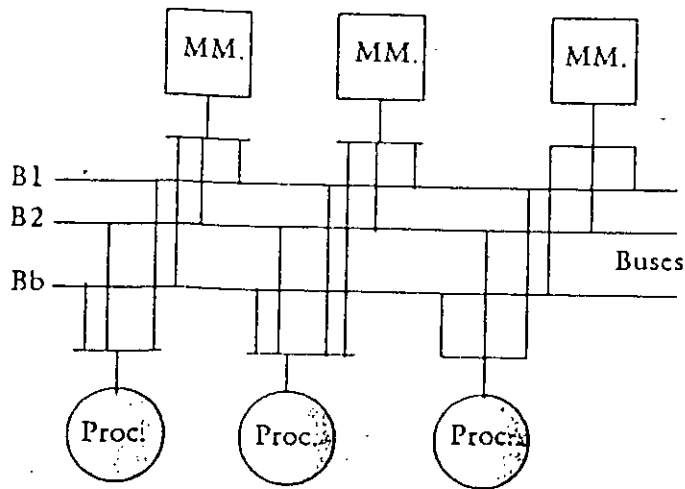


Figure 1.4 : Multiple bus multiprocessor system. [2]

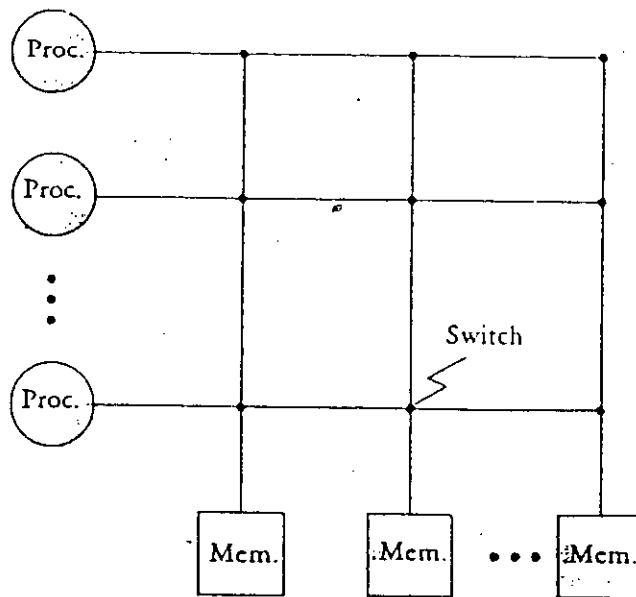


Figure 1.5 : Multiprocessor system with crossbar interconnection network. [2]

procedure level and for efficiently manipulating data structures. Instruction should be provided for procedure linkage, looping constructs, parameter manipulation, Multidimensional index computation, and range checking of addresses. Furthermore, the instruction set should also include instructions for creating and terminating parallel execution paths within a program. Thus a full set of addressing modes are desirable. Hardware counters and real-time clocks should be provided to generate a unique name of process identification and time-out signals required for process management. These times can also be used in a multiprocessing system to detect many errors by associating a "watchdog" timer with important system resources, as done in the C.mmp. A multiprocessor system provides a natural environment where each component can monitor each other relatively easily.

### 1.3 Interconnection Networks:

The principal characteristics of a multiprocessor system [2,5-8] is the ability of each processor to share a set of main memory modules. This sharing capability is provided through a interconnection network. Different types of interconnection networks are described below:

(a) Time shared or common buses: The simple interconnection system for multiple processors is a common communication path connecting all of the functional units. An examples of a multiprocessor system using the common communication path is shown in figure 1.3 The common path is often called a time shared or common bus. This organization is the least complex and the easiest to reconfigure. Such an

interconnection network is often a totally passive unit having no active component such as switches. Transfer operations are controlled completely by the bus interfaces of the sending and receiving units. An arbiter determines which processor will get control of the bus in case of more than one processor requesting concurrently. If there are a large number of processors in a system then single bus reduces the system performance because of long delay experienced by a processor waiting in queue in time of need of a bus. So to achieve greater system performance multiple bus system can be used which is also the simplest form of interconnection network. Then for a certain system with some processors and a number of memory modules, the number of buses required to achieve best system performance should be determined so that optimum bus utilization is achieved. The multiple bus multiprocessor system is shown in figure 1.4 .

(b) Cross bar switch and multiport memories: In a crossbars system separate path is available from each memory to the processors as shown in figure 1.5 . The crossbar switch possesses complete connectivity with respect to the memory modules because there is a separate bus associated with each memory module. So in crossbar system there is no bus conflict. Only conflict is memory contention when two or more processors request the same memory. The important characteristics of a system utilizing a crossbar interconnection matrix are the extreme simplicity of the switch-to-functional unit interfaces and the ability to support simultaneous requests for all memory modules. To provide these features requires major hardware

capabilities in the switch. Not only must each cross point be capable of switching parallel transmissions, but it must also be capable of resolving multiple requests for access to the same memory module occurring during a single memory cycle. These conflicting requests are usually handled on a predetermined priority basis. The result of inclusion of such a capability is that the hardware required to implement the switch can become quite large and complex. Although very large scale integration (VLSI) can reduce the size of the switch, it will have little effect on its complexity.

(c) Multistage interconnection networks(MIN): The construction of a simple crossbar switch is shown in figure 1.6. Consider the 2\*2 crossbar switch in this figure. This 2\*2 switch has the capability of connecting the input A to either the output labeled 0 or the output labeled 1, depending on the level of some control bit  $c_A$  of the input A. If  $c_A = 0$ , the input is connected to the upper output, and if  $c_A$  is 1, the connection is made to the lower output. Terminal B of the switch behaves similarly with a control bit  $c_B$ . The 2\*2 module also has the capability to arbitrate between conflicting requests. If both inputs A and B require the same output terminal, then only one of them will be connected and the other will be blocked or rejected. By introducing buffers within switches its performance can be increased [2-3]. With this basic module it is possible to build a MIN. In figure 1.7. a 8 by 8 omega network is shown build from these basic modules.

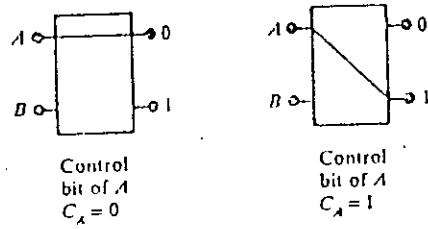


Figure 1.6 : A 2x2 switch. [2]

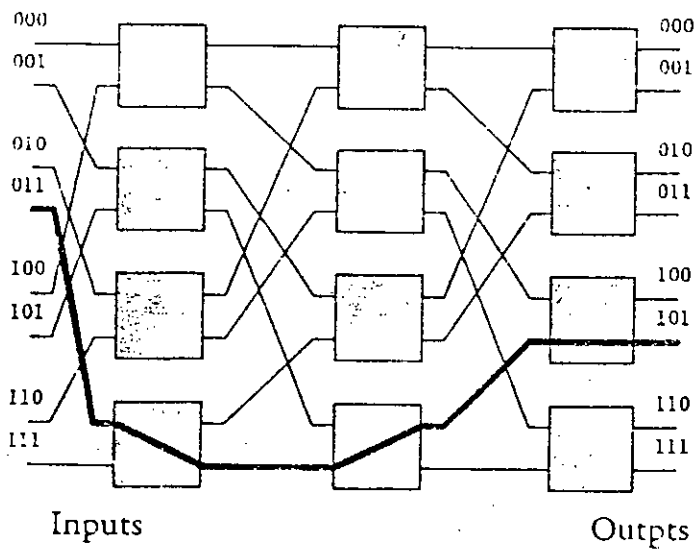


Figure 1.7 Omega network

#### 1.4 Parallel memory organizations [2]:

Low order interleaving of memory modules is advantageous in multiprocessing system when the address space of the active processes are shared intensively. If there is very little sharing, low-order interleaving may cause undesirable conflicts. Concentrating a number of pages of a single process in a given memory module of a high-order interleaved main memory is sometimes effective in reducing memory interference. In this case, a specific memory module  $M_i$  may be assigned to place most of the pages belonging to a process executing on processor  $i$ . Such a memory module is called the home memory for processor  $i$ . If the entire set of active pages of a process being executed on processor  $i$  is contained in memory  $M_i$ , and if memory  $M_i$  contains no pages belonging to processes running on other processors, then processor  $i$  encounters no memory conflicts. If every processor has the entire set of active pages of those processes that are running on it in its home memory, there will be no memory conflicts. The concept of home memory can be extended so that a set of modules  $\{M_i\}$  are assigned as the home memories of processor  $i$ . This assumes that there are more memory modules than processors, so that at all times each memory module is associated with one processor. That is  $\{M_i\} \text{ and } \{M_j\} = 0$  for  $i \neq j$ . The home-memory organization for multiprocessors has an additional architectural advantage beyond the reduction in memory interferences .

The processor-memory interconnection network (PMIN) of a multiprocessor system may be expensive, slow, and complicated. So there can be an alternative organization in which

each memory has two ports, one of which connects to the PMIN and one of which connects directly to the home processor. This topology permits enhanced access by each processor to its home memory by frequently avoiding switching time through PMIN and permitting decreased cable lengths between processors and their home memories. Since PMIN participates in only a minority of all memory accesses with this organization, its speed become less critical and substantial cost savings may also be possible. The concurrent (C) access memory configuration used for pipeline processors can also be used by multiprocessors. For tightly coupled multiprocessors, a single C access configuration can be designed to match the bandwidth requirements of the processors. In this case, the main memory and the processors are on the opposite sides of the PMIN and references to memory by the processors must traverse the PMIN. Therefore, the processor encounter memory conflicts as well as transmission delays. To reduce these effects, a private cache is usually used associated with each processor in multiprocessor so that most of the referenced data and instruction can be found in the cache. However, the data bus width may affect the cost and transfer time of a block of data.

#### 1.5 Operating system requirements for multiprocessor:

There is conceptually little difference between a operating system requirements of a multiprocessor and those of a large computer system utilizing multiprogramming. However, there is the additional complexity in the operating system when multiple processor must work simultaneously. This complexity is



also a result of the operating system being able to support multiple asynchronous tasks which execute concurrently. The functional capabilities which are often required in an operating system for a multiprogrammed computer include the resource allocation and management scheme, memory and data set protection, prevention of system deadlocks and abnormal process termination of exception handling. In addition to these capabilities, multiprocessor system also need techniques for efficient utilization resources and, hence, must provide input-output and processor load-balancing schemes. One of the main reasons for using a multiprocessor system is to provide some effective reliability and graceful degradation in the event of failure. Hence, the operating system must also be capable of providing system reconfiguration schemes to support graceful degradation. These extra capabilities and the nature of the multiprocessor execution environment places a much heavier burden on the operating system to support automatically the exploitation of parallelism in the hardware and the programs being executed[2].

An operating system which operates poorly will negate other advantages which are associated with multiprocessing. Hence, it is of utmost importance that the operating system for a multiprocessing computer be designed efficiently. The presence of more than one processing unit in the system introduces a new dimension into the design of the operating system. The influence of large number of processor on an operating system is still a research problem. The modularity of processors and the interconnection structure among them affect

the system development. Furthermore , communication schemes, synchronization mechanisms, and placement and assignment policies dominate the efficiency of operating system. There are basically three organizations that have been utilized in the design of operating system for multiprocessor, namely master slave configuration, separate supervisor for each processor, and floating supervisor control [2].

For most multiprocessors, the first operating system available assumed the master-slave mode. This mode in which the supervisor is always run on the same processor, is certainly the simplest to implement. Furthermore, it may often be designed by making relatively simple extensions to uniprocessor operating system that include full multiprogramming capabilities. Although the master slave type of system is simple it is generally inefficient in utilization of system resources. In a master slave mode, one processor called the master maintains the status of all processors in the system and distributes works to slave processors. An example of the master slave mode is in the Cyber-170, where the operating system is executed by one of the peripheral processor. All other processors are treated as slaves. When there is a separate supervisor running in each processor, the operating system characteristics are different from the master slave system. This is similar to the approach taken by the computer networks, where each processor contains a copy of a basic kernel. Resource sharing occurs at a higher level, say via a shared file structure. Each processor services its own need. However, since there is some interaction between the processors,

it is necessary for some of the supervisory code to be re-entrant or replicated to provide separate copies for each processor. The floating supervisor control scheme treats all the processors as well as other resources symmetrically or as an anonymous pool of resources. This is the most difficult mode of operation and most flexible. In this mode, the supervisor floats from one processor to another, although several of the processors may be executing supervisory service routine simultaneously. This type of system can attain better load balancing over all type of resources[2].

#### 1.6 Some Examples Of Multiprocessor Systems[1-3]:

(i) The C.mmp System: The C.mmp is composed of 16 PDP-11/40E(slightly modified) minicomputers sharing a 16 memory banks via crossbar. The average time to execute an instruction on a PDP-11/40 is approximately 2.5 us. Each processor has an 8K-byte local memory that is used primarily for operating system functions. The principle secondary memories of the C.mmp consists of four drives of 40M-byte disk controllers, three drives of 130M-byte disk controllers, and fixed head disks with zero latency controller that are used for paging space. The peripheral devices are assigned to the Unibus of specific processors. Hence there is no physical sharing of peripherals. A processor cannot initiate an I/O operation on a peripheral that is not on its Unibus. An interprocessor bus which connects the entire set of processors is used to perform the general function of interprocess communication. The bus provides a common lock as well as an interprocessor control. These two logically and functionally separate features travel separate data paths,

although they share a common control. Each processor has an interbus interface that defines the processor's bus address and makes available the bus functions to the software[2].

(ii) The S-1 Multiprocessor System: The S-1 consists of 16 uniprocessors which share 16 memory banks via a crossbar switch[2]. Each memory bank can up to  $2^{30}$  bytes of semiconductor memory and hence a total physical address space of 16 gigabytes( $2^{34}$ ). Each processor has a private cache. The S-1 multiprocessor system is developed to perform computations at an unprecedented aggregate rate on a wide variety of scientific problems. The S-1 is implemented with the S-1 uniprocessors called Mark IIAs. The uniprocessor is designed especially to facilitate pipelined parallelism in the fetching and decoding of instructions, the associated fetching of instruction operands, and the eventual execution of instructions. The preparation and execution of instructions that specify both scalar and vector operations are pipelined. Every instruction proceeds through multiple pipeline stages, including instruction preparation, operand operation, and execution.

(iii) PASM: PASM is a multifunction partitionable SIMD/MIMD system being designed at Purdue for image understanding[3]. It is to be a large scale, dynamically reconfigurable multiprocessor system, which will incorporate over 1,000 complex processing elements. Other than image processing and pattern recognition it can also be applied to speech understanding and biomedical signal understanding. PASM can also serve as a research tool for parallel processing with emphasis on

large-scale SIMD/MIMD parallelism. This system has hierarchical control with a system control unit responsible for process scheduling, resource allocation, parallelism mode and overall coordination. Microcontrollers act as the controllers for the processor memory pairs in SIMD mode and orchestrate the activities of processor memory pairs in MIMD mode. Each microcontroller consists of a microprocessor and two memory units so that memory loading and computations can be overlapped. There are four microcontrollers in the prototype system that are able to control upto four processors each. The microcontroller processors and memories are connected by a shared reconfigurable bus. Control storage contains the programs for the microcontrollers. Their loading is controlled by the system control unit. PASM's multistage network is a generalized cube with straight, exchange and broadcast capabilities. The network uses a commutative routing algorithm which is an improvement over the packet switching routing algorithm used in Texas Reconfigurable Array computer (TRAC). PASM uses different network for each function, such as, data access, instruction sharing and I/O. The PASM represents a mix of special and general purpose architecture and therefore it may be prove to be efficient for some evenly partitionable problems such as image processing, but not adequate for some real-time processing tasks because of hierarchical control and complex scheduling.

(iv) The HEP Multiprocessor System: The Heterogeneous Element Processor is a large-scale Scientific multiprocessor system which can execute a number of

sequential(SISD) or parallel(MIMD) programs simultaneously. The system consists of upto 16 processor execution modules(PEM) and upto 128 data memory modules(DMM). The PEM is designed to execute multiple independent instruction streams on multiple data stream simultaneously and it consists of its own program memory and an instruction processing unit(IPU). The program memory in each PEM has a capacity ranging from 1 to 8 megabytes. Instructions of active processors which are allocated to a PEM are buffered in the program memory. The HEP is the first commercially MIMD multiprocessor system. In the HEP system, a set of cooperating processes constitute a task. Tasks and processes are of two types: user or supervisor. The execution environment of a task is its task domain, which is defined by 64-bit task status word (TSW). The TSW provides protection and relocation for each task by a specification partition of the program, constant, register and data memories into areas. In addition to TSW There is a process status word (PSW), which contains a 20-bit program counter and other state information for a HEP process. Each PSW points to an instruction that is ready for execution. There is a process tag (PT) in the task queue for each PSW that points to an instruction that is ready for execution. When a task is first initiated, it has only one PSW; that is, one process. The software creates additional PSWs as new processes are created to initiate parallel processing within a task. There is a PSW queue which can hold a total of 128 PSWs: 64 for user processes and 64 for supervisor processes[1].

(v) The IBM 370/168MP system: IBM 370/168

Multiprocessing(MP) consists of two IBM 370/168 uniprocessor systems. The two CPUs are mutually exclusive and can not communicate each other directly. The two processors in in the 370/168 MP share from 2 to 16 million bytes of main storage. each CPU has either 8K-byte or 16K-byte cache with reduced 80-ns access time of 8 bytes. It has 22 block multiplexer channels. The block multiplexer channels permit concurrent processing of multiple channel programs for various speed peripheral devices. The multisystem control unit(MCU) provides the necessary interconnection hardware between the two CPUs and shared memories. It also contains a configuration control panel for the purpose of manual system reconfiguration. The 370/168 configuration is considered loosely coupled because two separate copies of operating systems are running in the two CPUs.

(vi) The Univac 1100/90 system: This is the most recent system by Sperry Univac. The system permit one, two, three or four central processing units(CPU) as 1100/91, 1100/92, 1100/93, and 1100/94 systems respectively. The 1100/9x is an x by x system containing x CPU and x I/O processors which can be tightly coupled. However, loosely coupled systems are also possible in which there are two independent systems sharing one mass storage subsystem. The 1100/94 system configuration, in addition to having four CPUs and four I/O processors, contains four main storage units and two system support processors. Each CPU is pipelined with an 8K word instruction and an 8K word data cache. A word is 36 bit wide. Each cache is organized into 256 sets with four blocks per set. Each block contains eight words.

The CPU uses a virtual addressing scheme with  $2^{36}$  words of address space. The initial address is divided into four portions. A segmentation scheme is used with a maximum of 262,144 segments.



## CHAPTER 2

### ARBITRATION

#### 2.1 General Description:

In a multiprocessor system processors share memories through buses. When a processor requests a memory module there may be two types of conflicts.

(i) Two or more processors can request the same memory module. So it is required to decide which processor will access the memory.

(ii) The number of processors requesting buses is greater than the number of available buses. Here also it is required a method to decide which processors will get buses.

The hardware which decides which one of the competing processors will win the resource when conflicts arise is known as arbiter and the process of making the decision is known as arbitration.

Arbiter resolves a memory or a bus conflict depending on some rules which are called arbitration protocol. Appropriate protocol should be chosen for achieving high level of system performance. Five arbitration protocols commonly used are:

- (1) Equal priority protocol.
- (2) Unequal priority protocol.
- (3) Rotating priority protocol.
- (4) Random delay protocol.
- (5) Queuing protocol.

For asynchronous circuit switched system equal

priority protocol and a combination of unequal priority and random delay protocol are used. For synchronous circuit switched system equal priority protocol and unequal priority protocol are used. For packet switched synchronous and asynchronous system equal priority protocol is used.

## 2.2 circuit switched asynchronous system:

### (a) Unequal priority protocol:

Processor can generate request after end of processing or during a cache miss. When a processor generates request it checks the state of requested memory and buses. If the requested memory and any of the buses are free then the processor goes for memory arbitration. Otherwise it waits until the requested memory and any of the buses are free. If there are more than one processor requesting one memory module then the arbiter selects the highest priority processor from them. The processors which fail in memory arbitration waits until that memory module becomes free and there is at least one free bus. After the wait, the processors resubmit requests for memory and bus arbitration. If the number of free buses is less than the number of processors which have won in memory arbitration then the arbiter selects a number of processors equal to the number of free buses from winning processors. Also in this selection the higher priority processors win. The processors which fails in bus arbitration waits until any of the buses become free. Then they resubmit and again go through memory and bus arbitration.

### (b) Random delay protocol:

Processors can generate request at any time after

end of processing or at a cache miss. When a processor generates request it scans the state of requested memory and buses. If the requested memory and any of the buses are free, it proceeds for memory arbitration. Otherwise it waits for a random amount of time. This random delay is generated by a random number generator. After waiting it resubmits request and go through memory and bus arbitration. In memory arbitration, if there are more than one processor requesting a particular memory module then the memory arbiter selects the highest priority processor. The winning processor goes for bus arbitration and the processors which have failed in memory arbitration waits a random amount of time generated by a random number generator. Then it resubmits the request for the memory module and again go through memory and bus arbitration. In bus arbitration, if the number of free buses is less than the number of processors which have won in memory arbitration then the arbiter selects a number of higher priority processors equal to the number of free buses from winning processors. But if the number of winning processors are less than the number of free buses all the processors which have won in memory arbitration win in bus arbitration. The processors which have failed in bus arbitration wait a random amount of time generated by a random number generator. And then they resubmit requests and repeat memory and bus arbitration.

### 2.3 Circuit switched synchronous system:

#### (a) Equal priority protocol:

Processors can generate requests at the beginning of system cycle. If two or more processors submit request for a

particular memory module, any one of the processors may win in the arbitration. A memory arbiter selects one processor from all requesting processors with equal probability. All the processor which win in memory arbitration go for bus arbitration. Here atmost  $B_f$  processors out of  $M_r$  processors are selected ( $B_f$  = number of free buses,  $M_r$  = number of winning processors in memory arbitration,  $M_r \leq M$  and  $M$  = total number of memory modules) by a M to B arbiter(or bus arbiter). Those processors which win in both arbitration occupy buses and perform memory operation for a number of system cycles for block replacement. Those processors which have failed in arbitration resubmit requests at the beginning of next clock. The main difference between asynchronous and synchronous system is that in synchronous system processors can generate requests only at the beginning of system cycle. **(b) Unequal priority protocol:**

Processor can generate requests at the edge of clock. Every processor has unique priority number. If more than one processor request a particular memory module the memory arbiter selects the highest priority one. In bus arbitration atmost  $B_f$  higher priority processors win out of  $M_r$  winning processors in memory arbitration. Those processors which win in both arbitration occupy buses and do memory operation for a number of system cycle( 1 to 6 slots) for a block replacement. Those processors which have failed in arbitration resubmit their requests at the beginning of next system cycle.

#### **2.4 Packet switched asynchronous system(Equal priority protocol):**

A processor submits request when there is a cache

miss or that it has finished processing (in a non cache system). At this time processor looks for free buses. If one or more buses are free processor goes for bus arbitration. From requesting processors atmost  $B_f$  processors are selected by arbiter with equal probability and send their requests for cache block for replacement through the occupied buses to the IMP (Intermediate message processor or controller) and then controller recognizes that a memory module have to be requested. Controller tracks the requested memory status. When a memory module becomes free and requested by more than one processor then the memory arbiter selects one of the requesting processors with equal probability. Then the winning processor's block read or write operation takes place. When memory operation ends, IMP checks if there is any bus free to transfer the requested block. If there is free bus and the request wins in bus arbitration the block is transferred through bus. Resubmission of request for bus occurs when any free bus is found and resubmission for memory occurs when requested memory again becomes free.

### 2.5 Packet switched synchronous system(Equal priority protocol):

A processor submits request when there is a cache miss or it finishes processing and it is a beginning of system cycle. The main difference between synchronous and asynchronous system is that in synchronous system request for any bus or a memory module can be generated at the beginning of system cycle. The sequence of operation in arbitration is same as that described in previous section. Resubmission of request for bus occurs at the beginning of system cycle and resubmission for memory also occurs at the beginning of system cycle.

## 2.6 Arbiter Design:

### (a) Asynchronous system:

For asynchronous system a two level unequal priority arbitration logic is used for achieving equal priority protocol. When a processor needs resource it raises its request signal high. Let  $R_i$  is the request line of processor  $P_i$ , where  $i=1,2, \dots N$ . When a processor submits request it finds if requested memory and any of the buses are free. Logic equation for generating request for memory module  $j$  by the processor  $i$  is,

$$C_{ij} = R_i \overline{M_{sj}} (\overline{B_{s1}} + \overline{B_{s2}} + \overline{B_{sk}}).$$

Where,  $B_{s1}, B_{s2}, \dots B_{sk}$  are bus status signal,  $M_{sj}$  is the  $j$ th memory status signal.

If  $C_{ij}$  is high then the processor sends its arbitration number to arbitration lines of the requested memory. For  $j$ th memory each processor has a module to see whether it has won in first level of arbitration or not. The logic equation [10] for this module is as follows:

$$C_n = C_{ij} (\overline{A_{nl}} + a_{nl}).$$

$$A_{nl}^+ = A_{nl} + C_{ij} a_{nl}.$$

Where,  $a_{nl}$  is  $l$ th bit of arbitration number.  $C_{ij}$  is compete signal for this module.  $A_{nl}^+$  is the next state of line  $A_{nl}$  after application of arbitration number by a processor. In figure 2.1] the logic diagram for first arbitration module is shown .

If a processor wins first level of arbitration it has  $Y_{ij}$  equal to 1 and it sends this  $Y_{ij}$  as compete signal to second arbitration module. In second level each memory module has

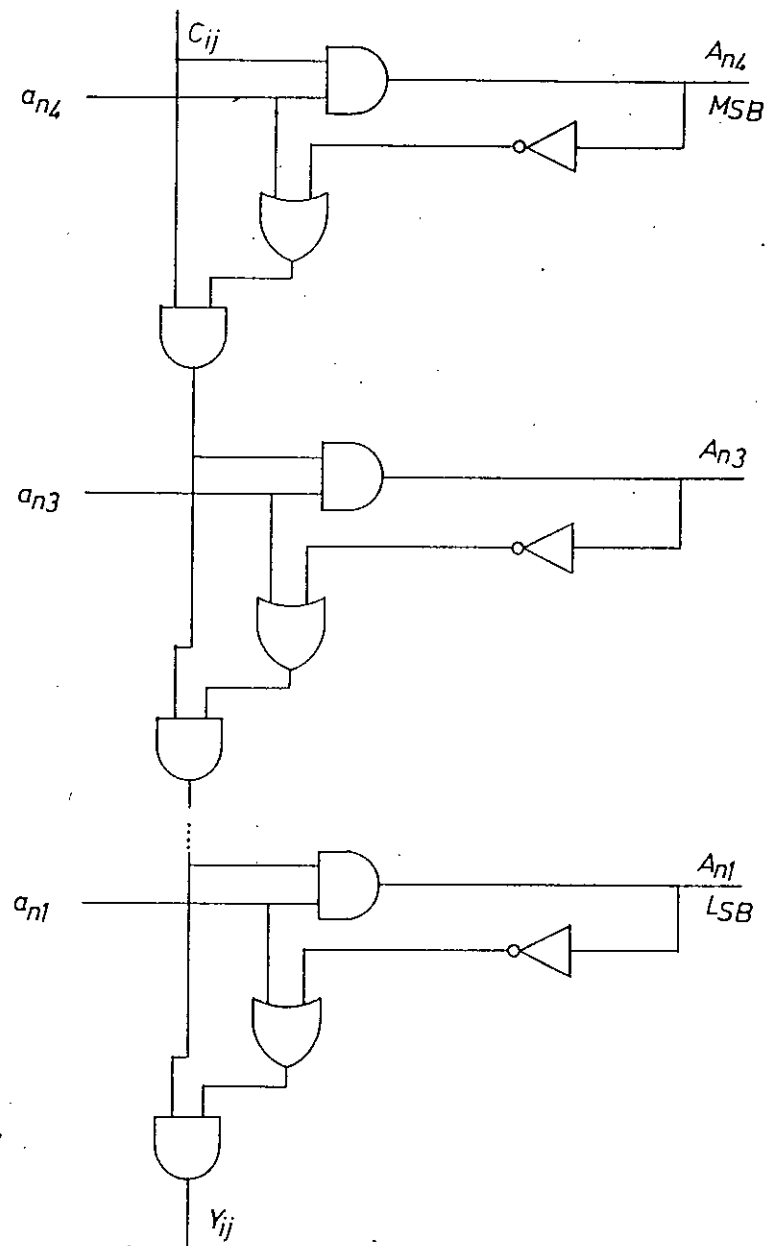


Figure 2.1 : Logic diagram for memory arbitration module [10],





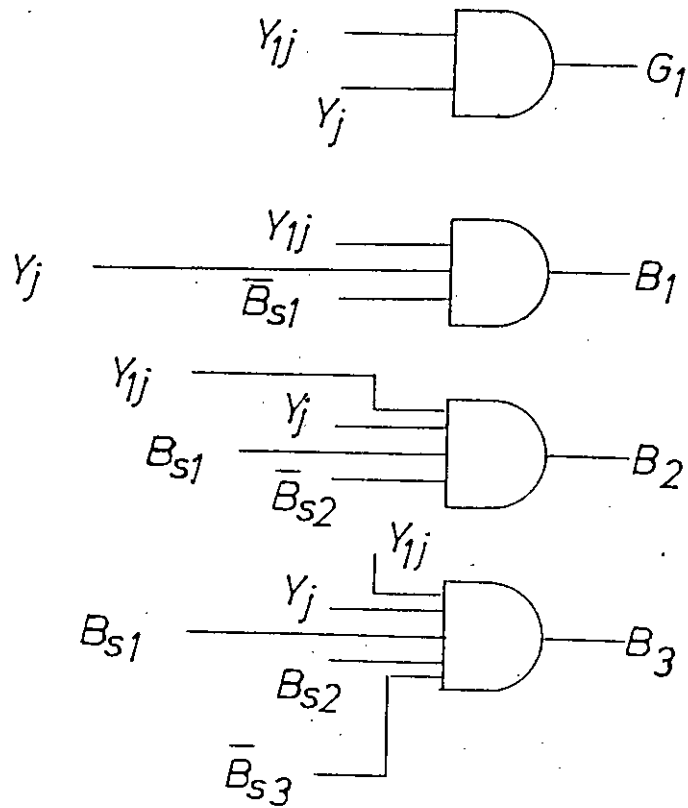


Figure 2.2 : Logic diagram for bus arbitration module.

different arbitration number. The module which has highest arbitration number will win in second level of arbitration. If a processor wins in second level of arbitration it gets its grant signal high and the bus number is send to the processor so that it can occupy the bus for data transfer. The bus number for a winning processor is determined by the following equations:

$$B_1 = y_{ij} \overline{b_{s1}}$$

$$B_2 = y_{ij} b_{s1} \overline{b_{s2}}$$

$$B_k = y_{ij} b_{s1} b_{s2} b_{s3} \dots \overline{b_{sk}}$$

The logic diagram of second arbiter module for memory module j is shown in fig 2.2 [10]. From these figures it is seen that total number of gate required for an asynchronous arbiter is:

$$M*N*[6 + 4*\log_2(N)] + M*[4*\log_2(M) + 1] + M*N*(B + 1) \\ = M*N*(B + 7) + M*(4*\log_2(M*N^N + 1))$$

And delay experienced by this arbiter is :

$$(\log_2(N)*3 + 3) + (\log_2(M)*3 + 1) + 1 \\ = 3\log_2(MN) + 5.$$

Where N = total number of processors

M = total number of memory modules

B = total number of buses

#### (b) Synchronous system:

(i) N to 1 arbiter: It is for memory arbitration. A N to 1 arbiter is made of a number of 2 to 1 arbiters. A 2 to 1 arbiter is shown in fig 2.3 [11-12]. Here  $R_0$  and  $R_1$  are two request lines. And  $G_0$  and  $G_1$  are two grant back signal to previous level and  $G_c$  is grant line from next higher

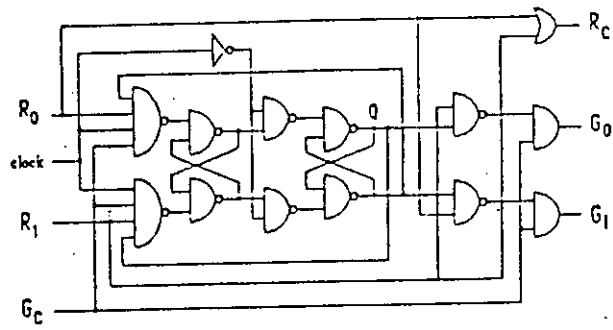


Figure 2.3 : Logic diagram for 2 to 1 arbiter [12]

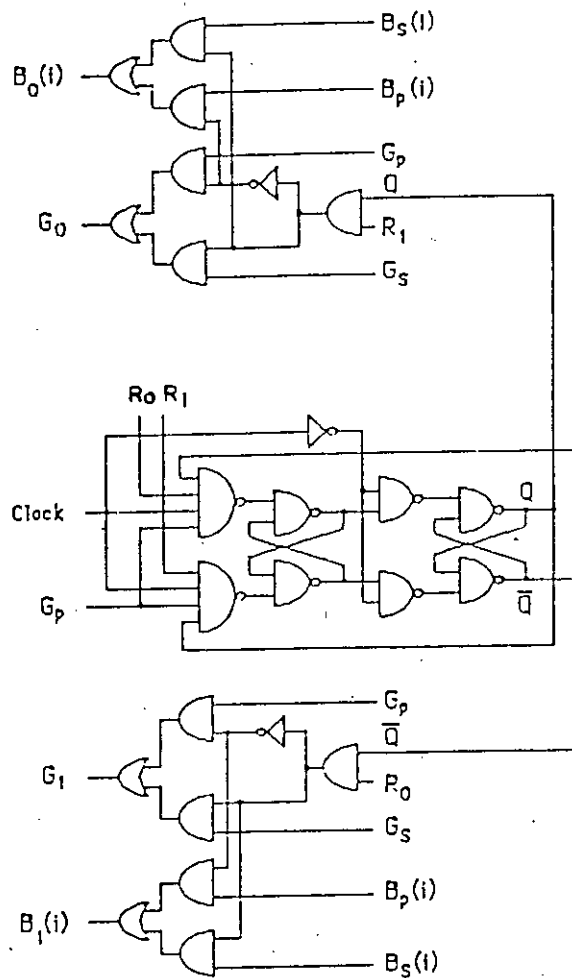


Figure 2.4 : Logic diagram of T1 module of 8 to 4 arbiter [12],

signal to previous level and  $G_c$  is grant line from next higher stage. Line  $R_c$  is request transferred to next higher level. If  $R_0$  is high and  $R_1$  is low then  $G_0$  will be high. If  $R_1$  is high and  $R_0$  is low then  $G_1$  will be high. If both  $R_0$  and  $R_1$  are low then  $G_0$  and  $G_1$  are don't care terms, because there are no request to monitor the grants. If both  $R_0$  and  $R_1$  are high then which request will get grant will depends on  $G_c$ , grant back from next higher level. If  $G_c$  is low then  $Q$  remains unchanged. But if  $G_c$  is high  $Q$  toggles. When  $R_0$  and  $R_1$  are both high then if  $Q$  is reset then  $G_0$  will be high i.e. request  $R_0$  will be serviced, and when  $Q$  is set  $G_1$  will be high, i.e., request  $R_1$  will be serviced. Number of gate required by a N to 1 arbiter is :

$$\text{gates} = 14*(N-1)$$

$$\text{delay} = 2*[\log_2(N/2) + 1]$$

Where N = total number of processors.

(ii) M-tO-B arbiter: If there are M memory modules and B buses only B of M requests can be serviced. There is needed a module T1, where there are two request lines  $R_0$  and  $R_1$ . Two grant back lines  $G_0$  and  $G_1$  to previous lower level, Two grant lines  $G_s$  and  $G_p$  from next higher level.  $B_0$  and  $B_1$  are bus numbers sent to lower level, and  $B_p$  and  $B_s$  are bus numbers from higher level.

If from  $R_0$  and  $R_1$  only one request is high then this request monitor the condition of grant back line  $G_p$ .

If both  $R_0$  and  $R_1$  are high and if state of flip-flop  $Q$  is reset, then request  $R_0$  monitors grant back  $G_p$  and request  $R_1$  monitors grant back  $G_s$ . Again when both  $R_0$  and  $R_1$  are

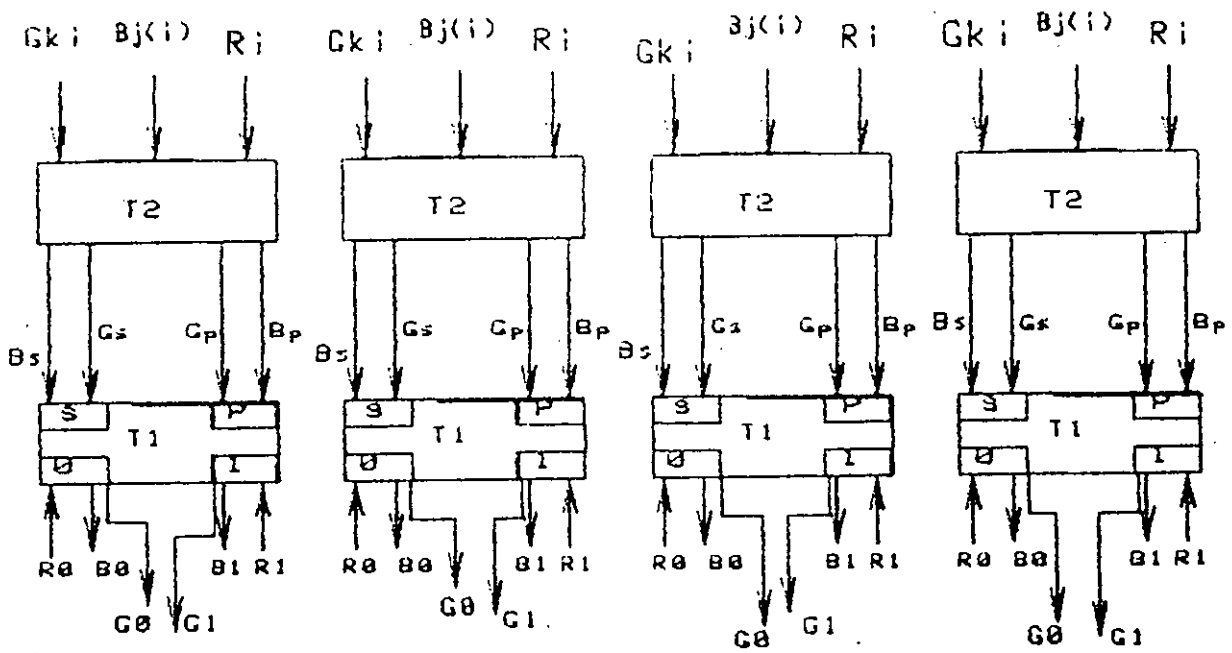


Figure 2.5 : Block diagram of 8 to 4 arbiter.

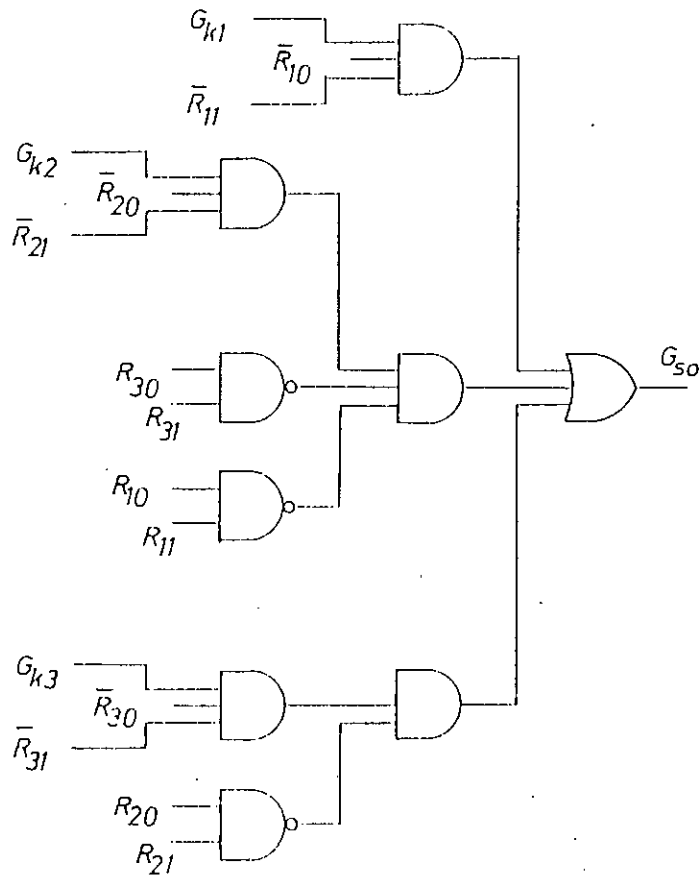


Figure 2.6 : Logic diagram of T2 module of 8 to 4 arbiter.

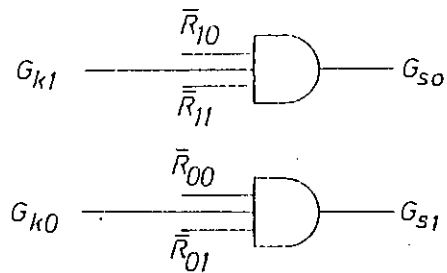


Figure 2.7 : Logic diagram of T2 module of 4 to 2 arbiter.

high and Q is set then request  $R_1$  monitors grant back  $G_p$  and request  $R_0$  monitors grant back  $G_s$ . So the logic equation [12] is as follows:

$$G_0 = \overline{Q}R_1G_p + QR_1G_s .$$

$$G_1 = \overline{Q}R_0G_p + QR_0G_s .$$

When a request is made and a grant signal is available, the number (in binary) of the available bus received from the last higher stage through the input bus port bits will be transmitted through the output bus port bits to the next lower stage. the boolean equations of a bit say bit  $i$  are similar to boolean equation for grant signals  $G_0$  and  $G_1$  with  $G_s$ ,  $G_p$  replaced by  $B_s$ ,  $B_p$ . So the logic equation for  $i$ th bit of bus number is as follows:

$$B_0 = \overline{Q}R_1B_p + QR_1B_s .$$

$$B_1 = \overline{Q}R_0B_p + QR_0B_s .$$

The logic diagram of T1 block is shown in figure 2.4 .

For 8 to 4 arbiter  $G_p$  and  $G_s$  signals can be generated in advance. If this module is called  $T_2$  [12] then the equations for four  $T_2$  modules are as follows:

$$G_{p0} = G_{k0} .$$

$$G_{s0} = G_{k1}\overline{R}_{10}\overline{R}_{11} + G_{k3}\overline{R}_{20}\overline{R}_{21}\overline{R}_{30}\overline{R}_{31} \\ + G_{k2}R_{20}R_{21}R_{30}R_{31}R_{10}R_{11} .$$

$$G_{p1} = G_{k1} .$$

$$G_{s1} = G_{k0}\overline{R}_{00}\overline{R}_{01} + G_{k2}\overline{R}_{20}\overline{R}_{21}\overline{R}_{30}\overline{R}_{31} \\ + G_{k2}\overline{R}_{30}\overline{R}_{31}\overline{R}_{20}\overline{R}_{21}\overline{R}_{00}\overline{R}_{01} .$$

$$G_{p2} = G_{k2} .$$

$$G_{s2} = G_{k3}\overline{R}_{30}\overline{R}_{31} + G_{k1}\overline{R}_{10}\overline{R}_{11}\overline{R}_{00}\overline{R}_{01} .$$

$$+ G_{k2} \overline{R_{00}} \overline{R_{01}} \overline{R_{10}} \overline{R_{11}} \overline{R_{30}} \overline{R_{31}}.$$

$$G_{p3} = G_{k3}.$$

$$G_{s3} = G_{k2} \overline{R_{20}} \overline{R_{21}} + G_{k0} \overline{R_{00}} \overline{R_{01}} \overline{R_{10}} \overline{R_{11}} \\ + G_{k1} \overline{R_{10}} \overline{R_{11}} \overline{R_{00}} \overline{R_{01}} \overline{R_{20}} \overline{R_{21}}.$$

$G_{k0}$ ,  $G_{k1}$ ,  $G_{k2}$  and  $G_{k3}$  etc. are grant signal back from higher stage. the block diagram and logic diagram of this arbiter is given in figure 2.5 and in figure 2.6. In this design gate delay is  $5d$ , where  $d$  is nominal gate delay and total number of gate required is 304.

For 4 to 2 arbiter logic equations for T2 modules are as follows:

$$G_{p0} = G_{k0}.$$

$$G_{s0} = \overline{R_{10}} \overline{R_{11}} G_{k1}.$$

$$G_{p1} = G_{k1}.$$

$$G_{s1} = \overline{R_{00}} \overline{R_{01}} G_{k0}.$$

The logic diagram of this arbiter is given in figure 2.7 .



## CHAPTER 3

### **ANALYTICAL METHODS**

#### **3.1 General Description :**

For determining performance of a system analytically it is necessary to model the system by some mathematical methods. Multiple bus multiprocessor system can be analyzed by queueing theory. The queueing model of circuit switching and packet switching multibus multiprocessor is shown in fig. 3.1 and 3.2 respectively. Depending on control strategy, switching methodology and timing philosophy there can be different kind of analysis techniques. The system where events can occur at the beginning of system cycle, i.e. synchronous system, can be represented by discrete Markov process [13] or semi-Markov process [8,14]. Synchronous system can be solved by probabilistic methods and combinatorial analysis [9]. The system where event can happen at any time, can be represented by queueing networks with infinite buffers or by queueing network with flow equivalent service center [15,16]. Equilibrium point analysis (EPA) [14] is also a good analysis tool for multiple bus multiprocessor system because it represents complicated stochastic systems with less complexity. But EPA may not be accurate enough. The rate of request is determined by local codes, cache misses and the processor speed. In asynchronous system this rate is described in terms of think time and common approximation for this time is negative exponential distribution.

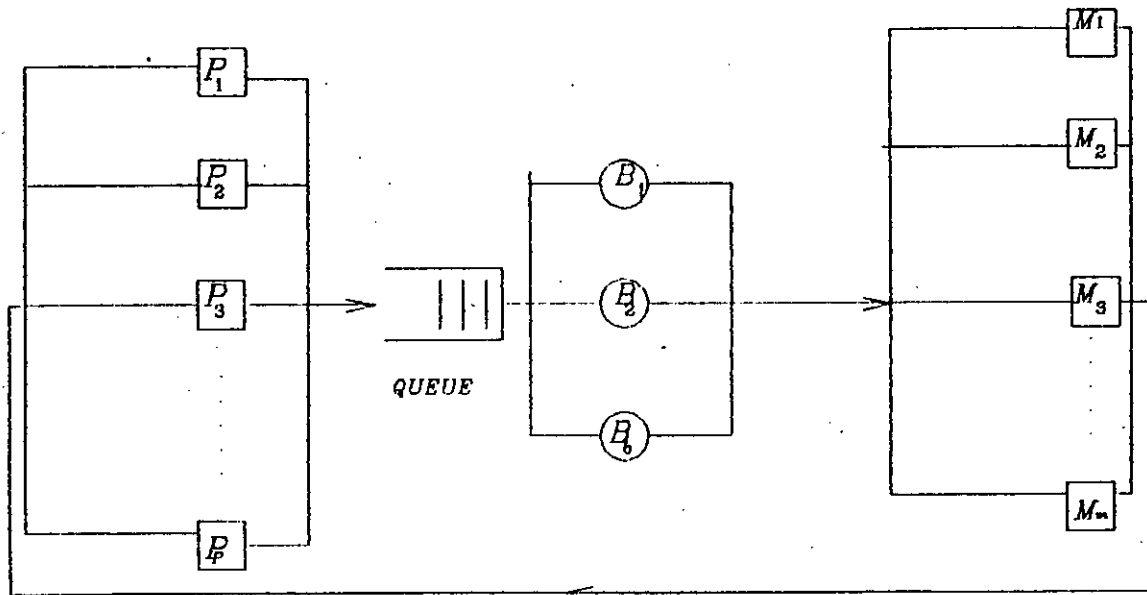


Figure 3.1 : Queueing diagram of circuit switched system.

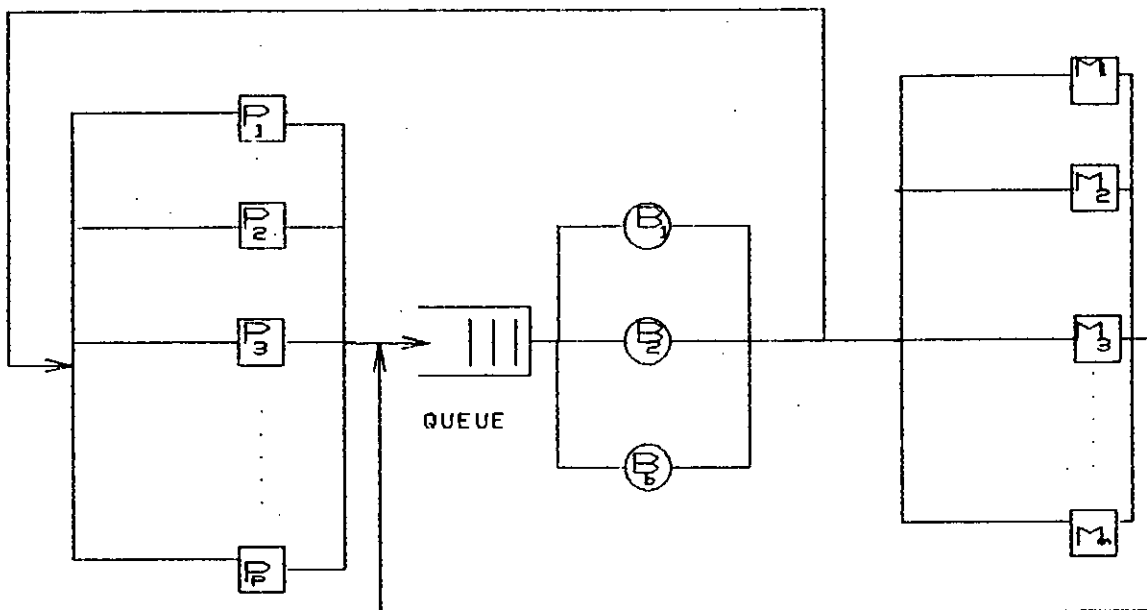


Figure 3.2: Queueing Diagram of Packet Switched Multiprocessor System

The usual approximation of the multiprocessor system is the uniform reference of the memory module. Thus a request is directed to a particular memory module with a probability  $1/M$ . This symmetric assumption makes the analysis tractable. In case of biased access to a particular memory module, the performance is expected to deteriorate because of longer queuing delay.

### 3.2 Semi-Markov process:

Multiprocessor system with uniform memory reference can be characterized by the following assumptions [8]:

(i) The behavior of the active element i.e. processor can be modeled as identical stochastic process.

(ii) The processor think for an integer number of system cycles before cache miss and this time is characterized by a discrete independent random variable  $t$  ( $\bar{t}$  is average of  $t$ ).

(iii) Each processor will submit a memory request after its thinking period; the request originating from the processor are independent of each other provided they are not resubmitting requests. The destination memory module of the non resubmitted requests originating from any processor will be determined by a discrete independent random variable which is uniformly distributed between 1 and  $M$ .

(iv) When requesting processor finds that requested memory module is busy then that processor has to wait until the connection is completed, i.e. it has to wait until the end of remaining memory connection time.

(v) When requested memory and any one of the buses are free but more than one processor submit request to get a

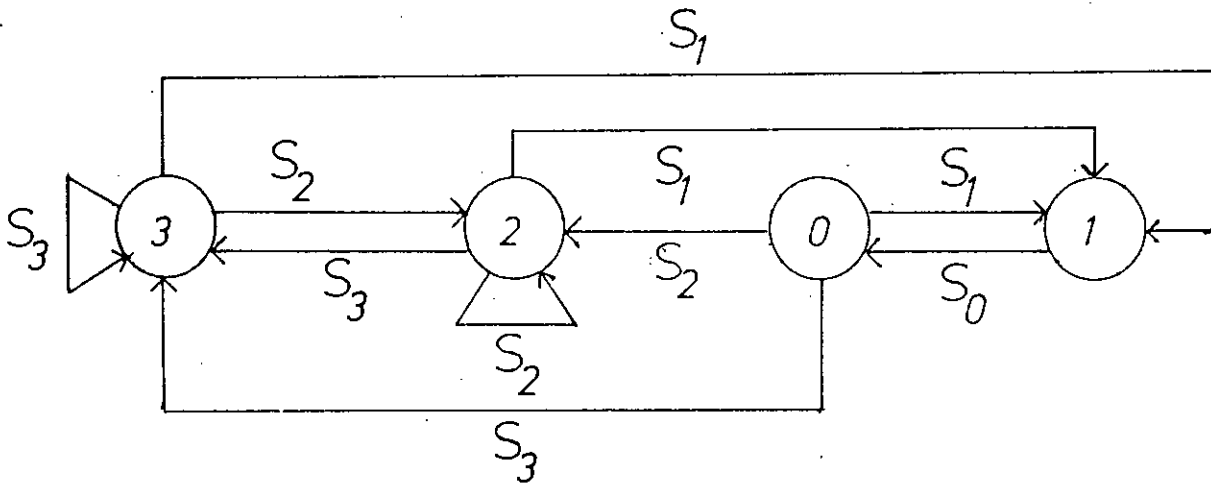


Figure 3.3 : Semi-markov model of circuiti switched synchronous system.

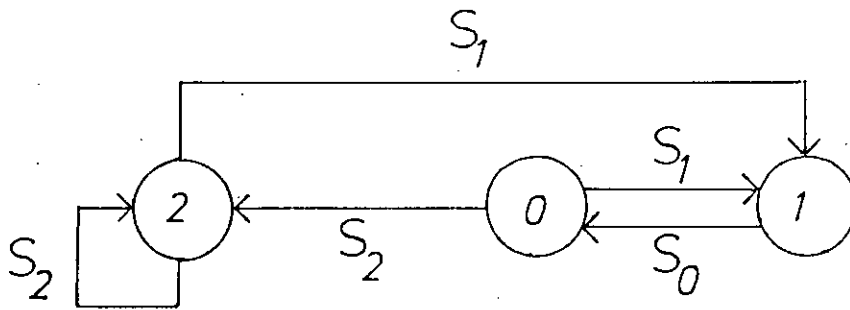


Figure 3.4: Semi-markov model of circuiti switched synchronous system without residual waiting state.

particular memory module then one processor is selected from the requested processors and other remain waited until memory connection is completed, i.e. other processor wait for a full memory connection time. So a circuit switched synchronous multiple bus system can be represented by the semi-Markov process of figure 3.3 . State 0 is processor's thinking state, state 1 is accessing or memory connection by the processor, state 2 is full wait(i.e., total memory connection wait) of the processor and state 3 is residual wait i.e. a part of memory connection wait.

Let  $S$ 's are state transition probabilities.  $S_0$  is the probability that a processor come back to thinking state after memory accessing. So  $S_0 = 1$ .  $S_1$  is the probability that a processor goes to state 1 for memory access.  $S_2$  is the transition probability a processor fails in arbitration and hence has to wait for full memory connection.  $S_3$  is the transition probability that a processor finds requested memory module busy servicing another processor. These transition probabilities can be find from the following equations[8]:

$$\begin{aligned}
 S_0 &= 1 \\
 S_1 &= (1 - \text{BUSY})\text{WIN1}*\text{WIN2} \\
 S_2 &= (1 - \text{BUSY})(1 - \text{WIN1})\text{WIN2} \quad \dots(1) \\
 S_3 &= \text{BUSY} + (1 - \text{BUSY})(1 - \text{WIN2}),
 \end{aligned}$$

Where,  $\text{BUSY} = ((N - 1)/M) (E1 - 1)L1$

$$\text{WIN1} = (1 - (1 - R)^N) / (N*r) = p / (N*r)$$

$$\text{WIN2} = \sum_{k=1}^B X(k)*Y(k)$$

$N$  = Total number of processors

$M$  = Total number of memory modules

$B$  = Total number of buses.

$E0$  = Average processing time.

$E1$  = Average memory connection time =  $C$  (say)

$E2$  = Full waiting time =  $C$

$E3$  = Residual waiting time =  $(C^2 - C)/(C - 1)$ .

$r$  = Probability that a processor generates request to a particular memory module at the beginning of system cycle.

$$X(k) = \sum_{i=1}^k \min(k, i) / i \binom{M-1}{i-1} p^{(i-1)} (1-p)^{(M-1)}$$

$$Y(k) = {}^B C_k q^{(B-k)} (1-q)^k$$

$$q = (n-1)(E1-1)L1/B$$

$$p = 1 - (1-r)^N$$

$$L1 = P1/E1$$

After defining the proper semi-Markov process it is necessary to find  $P_0$ ,  $P_1$ ,  $P_2$  and  $P_3$  which are the probability that a processor will be in state 0 or state 1 or state 3 respectively. And these probabilities can be deduced as follows:

$$P0 = E0.S1.M.r$$

$$P1 = E1.S1.M.r$$

$$P2 = E2.S2.M.r \quad \dots (2)$$

$$P3 = E3.S3.M.r$$

For iteration  $L1 = 0$  and  $r = 1/M$  is used.

Now, bandwidth,  $B_w = N * P_1$ , where  $n$  is the total number of processor and  $P_1$  is the probability that a processor is accessing a memory module.

Processor utilization,  $P_u = P_0 + P_1$ , i.e. processor is thinking or accessing a memory module.

Bus utilization,  $B_u = N * P_1 / B$ , the number of bus is

utilized.

Average queue length,  $Q_a = N(P_2 + P_3)$ , i.e. a processor is waiting (full or a part of memory connection) for requested memory module.

If memory word size is equal to number of data line, then memory connection and bus cycle time both equals. Then at the beginning of system cycle when processors may submit request the memory modules remain free. So, a processor which fails in arbitration has to wait for full memory connection and residual waiting state 3 of previous representation could be eliminated. With this the semi markov representation of circuit switched synchronous system becomes as shown in figure 3.4 . Where  $S_0$  is =1, the transition probability that a processor returns to thinking state after accessing .

$S_1$  = the probability of success that a processor become successful in memory or bus arbitration and access a memory module. It is the transition probability that from state 0 or state 2 to state 1.

$S_2$  = the probability that a processor fails in memory arbitration or bus arbitration. It is transition probability from state 0 or state 2 to state 2.

Then, bandwidth,  $B_w = N * P_1$ .

Processor utilization,  $P_u = P_0 + P_1$ .

Bus utilization,  $B_u = N * P_1 / B$ .

Average queue length,  $Q_a = N * P_2$ .

### 3.3 Probabilistic method:

Let the probability that a particular processor  $P_i$  submits request at the beginning of system cycle is  $r$ . Then, if there are  $M$  memory module, then the probability that a particular memory module  $M_j$  is requested by  $P_i$  is  $r/M$ , where  $M$  is the total number of memory modules. Then probability that  $M_j$  is not requested by  $P_i$  is  $(1 - r/M)$ , if there are total  $N$  number of processors. Then, the probability that no processor request  $M_j$  is  $(1 - r/M)^N$ . And then probability that at least one processor submit request to memory module  $M_j$  is  $(1 - (1 - r/M)^N)$ . It is the probability that memory module  $M_j$  is requested and one of the requesting processor definitely wins it. Let, it is denoted by  $\text{Pr}[E_j]$ . Then,

$$\text{Pr}[E_j] = 1 - (1 - r/M)^N.$$

Now, the probability that,  $i$  memory modules out of  $M$  memory modules is requested is given by,

$$f(i) = {}^M C_i \text{Pr}[E_j]^i (1 - \text{Pr}[E_j])^{(M-i)}.$$

So, memory bandwidth, which is dependent on the number of buses is given as [9]:

$$B_w = \sum_{i=1}^B i * f(i) + \sum_{i=B+1}^M B * f(i); \text{ Where } B = \text{total number of buses.}$$



## CHAPTER 4

### SIMULATION

#### 4.1 General Description:

To determine characteristics and performance of a system it can be possible to make a model that resembles the actual system. This model can be built by software programming. Also a small version of actual system can be pursued to determine the system performance. In computer, by simulation it implies to simulation by software. Simulation is pursued when actual measurements are time consuming and complex. By simulation it is possible to predict system performance accurately. Mathematical modeling is another way to evaluate system performance without going into actual design. But exact mathematical model of some practical system becomes much difficult, whereas the performance of these systems can be evaluated almost accurately by simulation.

There are three types of simulation [18-19].

- (i) Time driven simulation.
- (ii) Event driven simulation.
- (iii) Process driven simulation.

In time driven simulation all parameters are to be updated after a specific time interval. Simulation runs for a specific amount of time, say,  $t_{tot}$  and starts at some time, say  $t_{init}$ . Beginning from initial time all the processors, memory modules and bus conditions are to be updated after each unit time

of the system (1 slot).

In event driven simulation all parameters are to be updated when new event occurs. An event is said to have occurred, when an active element(i,e, processor) starts some processes or finishes them, i.e. when active elements changes state. There is a total specific amount of time. As simulation proceeds the events are updated and statistics are collected when an event occurs. Simulation ends at  $t_{tot}$ .

In the process driven simulation it is needed to prescribe the conditions (process) which cause an activity to start or end. The events which start or end the activity are not scheduled but are initiated from the conditions specified for the activity.

Simulation can be classified as discrete simulation and continuous simulation [19].

In discrete simulation the dependent variables, i.e. variables to be updated or calculated, changes discretely at specific point in simulated time referred to as event times. The time variable is either continuous or discrete depending on whether the discrete changes in the dependent variable can occur at any point in time or at specific points. In continuous simulation the dependent variables of the model may change continuously over simulated time. A continuous model may be either continuous or discrete in time, depending on whether the values of the dependent variables are available at any point in simulated time or only at specified points in simulated time.

#### **4.2 Development of Simulation Software:**

In the simulation program (A1 - A4) four components



## 4.2 Development of Simulation Software:

In the simulation program (A1 - A4) four components are simulated by structures and those are Processor(pro), bus, memory(mem) and controller(cont) as shown in figure 4.1. Event queue is formed by controller. In controller structure there are three fields. Processor number(p\_no), where identification number of processors are kept, current time field and next field to keep track of next member in the event queue. In processor structure(pro) there are five fields. The field next event(n\_e) is for making decision what next event should be done. The field, memory number (m\_no) is for deciding the memory number occupied by a processor and the field, bus number(b\_no) is for the number of buses occupied by a processor. In memory structure there are three fields. State field shows memory status i.e. it shows if the memory is free or being occupied by a processor. In bus structure there are three fields. State field shows bus status i.e. it shows if the bus is busy or idle. Current events are taken from front of event queue. Processed events are placed in proper place in event queue by a routine(insert). Different types of protocols are simulated by sorting the position of the processors in the event queue or by logic actually used in the programs (A1 - A4).

Number of busy processor, queue length, number of busy memory modules, number of busy bus are taken after end of each event. Here is an example :

Say  $n_i$  is the number of busy processor at time  $t_i$ , then the number of average busy processor,  $n_{avg}$  is:

$$n_{avg} = (n_1(t_2 - t_1) + n_2(t_3 - t_2) + \dots + n_i(t_{i+1} - t_i)) / t_{tot} \quad \dots (1)$$

The input parameters of the simulation are the processing time (geometric distribution and uniform distribution), memory connection time (constant) for circuit switched system; and processing time (uniform), bus connection time (constant) and memory connection time (constant) for packet switched system. For random delay protocol random delay time (uniform) should be supplied as an input parameter.

The output parameters are Average queue length, Processor utilization, Memory bandwidth and Bus utilization. These are evaluated as follows:

$$\text{Average queue length, } Q_a = (Nw_1(t_2 - t_1) + Nw_2(t_3 - t_2) + \dots + Nw_i(t_{i+1} - t_i)) / (t_{\text{tot}} - t_{\text{trans}}) \dots (2)$$

Where,  $Nw_i$  = is the number of processors waiting in queue in time  $t_i$  and

$t_{\text{trans}}$  = Some initial or transient time which is not taken into account in collection of statistics. This time is allowed to bring the simulated system into steady state condition. Now,

$$\text{Processor utilization, } P_u = n_{\text{avg}} / N \dots (3)$$

$$\text{Memory bandwidth, } B_w = (Mb_1(t_2 - t_1) + Mb_2(t_3 - t_2) + \dots + Mb_i(t_{i+1} - t_i)) / (t_{\text{tot}} - t_{\text{trans}}) \dots (4)$$

$$\text{and, Bus utilization, } B_u = (Bb_1(t_2 - t_1) + Bb_2(t_3 - t_2) + \dots + Bb_i(t_{i+1} - t_i)) / (t_{\text{tot}} - t_{\text{trans}}) \dots (5)$$

Where,  $Mb_i$  = Number of busy memory module at time  $t_i$ .  
 $Bb_i$  = Number of busy bus at time  $t_i$ .

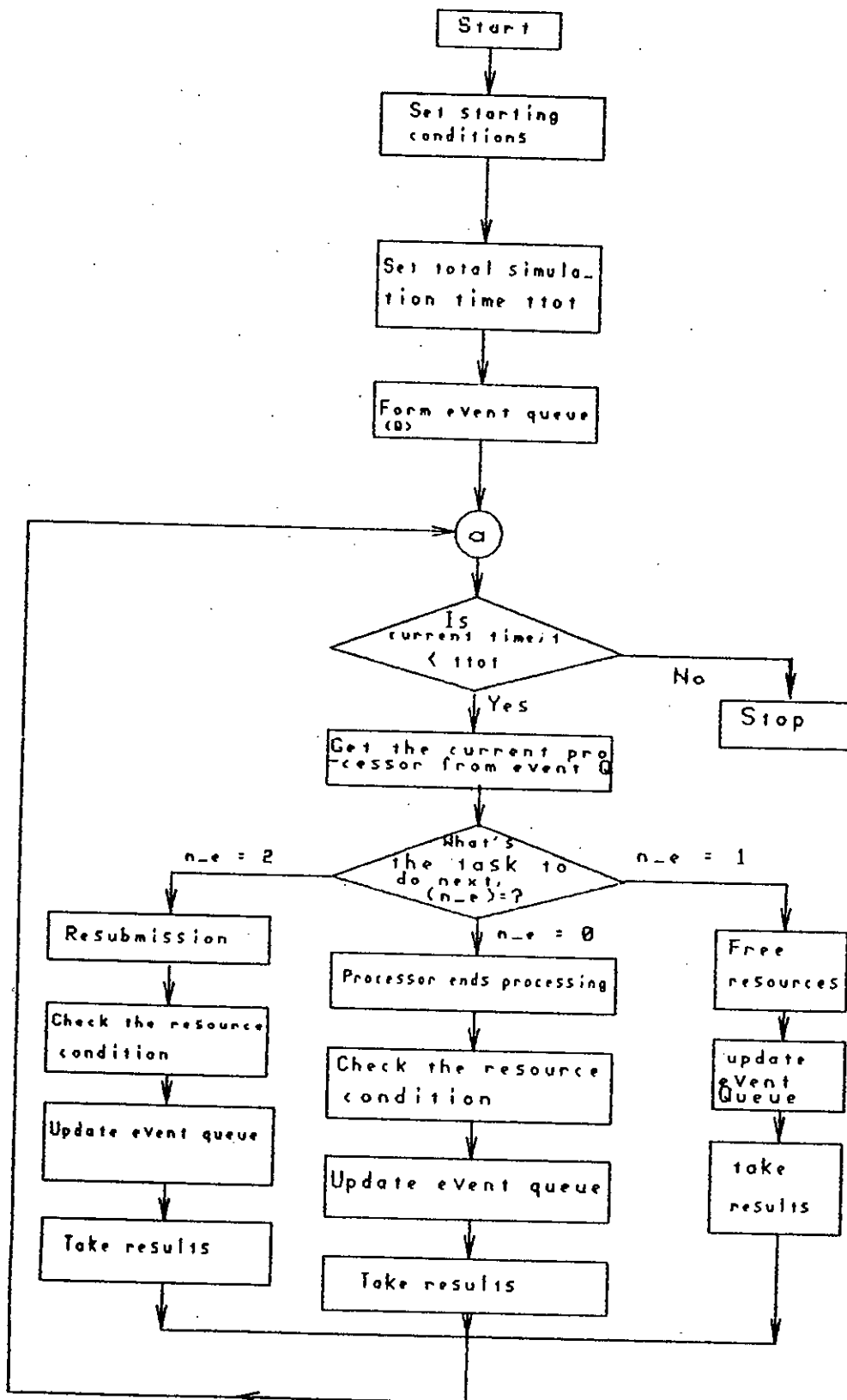


Figure 4.2 Flow-chart of simulation of circuit switched asynchronous system.

### 4.3 Circuit Switched System

#### (a) Asynchronous System:

First an event queue is formed where processor number and current time of processor are kept. At the beginning all the processors have been kept in processing state. Occurrence of a new event and the kind of new event are checked.

The system (Processors, Memory modules and buses) can experience three events by which simulation is performed. These are described in the following paragraphs.

(i) **End of processing:** when a processor ends processing it goes for arbitration. First step is memory arbitration followed by bus arbitration. The performance parameters should be updated according to success and failure in arbitration. Its time is incremented to update simulation. And it is assigned the next event,  $n_e = 1$  (fig 4.1), release of memory and bus (i.e. resource release), if it have won in arbitration and occupied resources. Otherwise, it should be the end of waiting for resubmission. All statistics are to be collected for performance evaluation.

(ii) **Processor has just ended memory connections:** When memory connection ends busy bus and memory are freed and performance parameters and number of busy bus and memory are updated. Processor's time is incremented to run the simulation and to make the processor busy. Its next event is given 1 (end of processing).

(iii) **Resubmission of request by processor:** The processor is given memory connection through bus if it wins arbitration. Then its time will be incremented by an amount equal to the connection time and next time it will cause an event after

ending memory connection i.e. its next event will,  $n_e = 2$ . But if the processor fails in arbitration its time is incremented by an amount equal to waiting time (according to protocol) and its next event will,  $n_e = 3$ .

These three states are repeated until total simulation time is completed. At each event performance measures - number of busy buses, number of busy memory modules and number of busy processors are updated. The simulation flowchart is shown in figure 4.2 .

#### (b) Synchronous System:

First an event queue is formed, taking all the processor information as queue element. At the beginning all processors are kept in processing state. Occurrence of new events and their numbers are checked to serve them appropriately.

If processor ends processing or there is a cache miss at any time other than at the beginning of system cycle then processor goes to wait state. In this case processor's time is incremented so that it can submit request at the beginning of next clock. Processor keeps on waiting. Its next event is set to be the end of waiting, i.e. state 3. Queue length increases: number of busy buses and memory modules are not changed.

As the processor ends processing at the beginning of system cycle then there can be three events as described in asynchronous system.

### **4.4 Packet Switched System:**

#### (a) Asynchronous system:

First an event queue is formed keeping all the processor information as queue element. At the beginning all the



processors are kept in processing state. Occurrence of new events and their numbers are checked to serve the events properly. There are seven events in which a processor can be present.

(i) End of processing(eop): When a processor ends processing it submits request to get a bus. If it wins in bus arbitration it keeps the bus busy for packet transfer for a fixed bus transfer time, say .05 s. Processor time is updated by bus transfer time so that it can release bus after end of packet transfer through bus and next event is set to 1, i.e. end of bus transfer in forward direction. Number of busy processors remains the same and number of busy buses is increased by one. If it fails in bus arbitration it will resubmit request when any one of the buses become free and processor time is incremented so that it can resubmit when a free bus is found. Processor's next event is set to 2. The number of busy processor is decreased by one.

(ii) End of bus transfer in forward direction:(eobf) The processor releases bus so number of busy bus is decreased by one. If it wins in memory arbitration it keeps the requested memory busy for a fixed memory connection time, and next event is set to 4, i.e. end of memory connection. The number of busy processor is decreased by one. If the processor fails in memory arbitration it has to wait until the requested memory is released by winning processor, i.e. the memory becomes free again. The processor time is updated so that it can resubmit request when requested memory become free and next event is set equal to 3, i.e. end of wait for memory. The number of busy processor is decreased by one and queue length is increased by

one.

(iii) End of wait for bus in forward direction (entbf): The processor resubmit request for bus. If it wins in arbitration it keeps bus busy for a specific amount of time, say so that processor can release bus after end of bus transfer and processor's next event is set to 1, i.e. end of bus transfer in forward direction. The number of busy processors and the number of busy buses are increased by one. If it fails in arbitration it will resubmit request when any one of the buses become free and processor time is incremented so that it can resubmit when a free bus is found. And processor's next event is set to 2, i.e. end of wait for bus.

(iv) End of wait for memory module (entm): The processor resubmit request for memory. If it wins in memory arbitration, it keeps the requested memory busy for a fixed memory connection time say 6 slots. So the processor time is incremented by the memory connection time, and next event is set to 4, i.e. end of memory connection. If the processor fails in memory arbitration it has to wait until the requested memory is released by the winning processor, i.e. the memory module becomes free again. The processor time is updated so that it can resubmit request when requested memory module becomes free and next event is set to 3, i.e. end of wait for memory module. Queue length is increased by one.

(v) End of memory connection(eom): The processor releases memory so the number of busy memory is decreased by one.

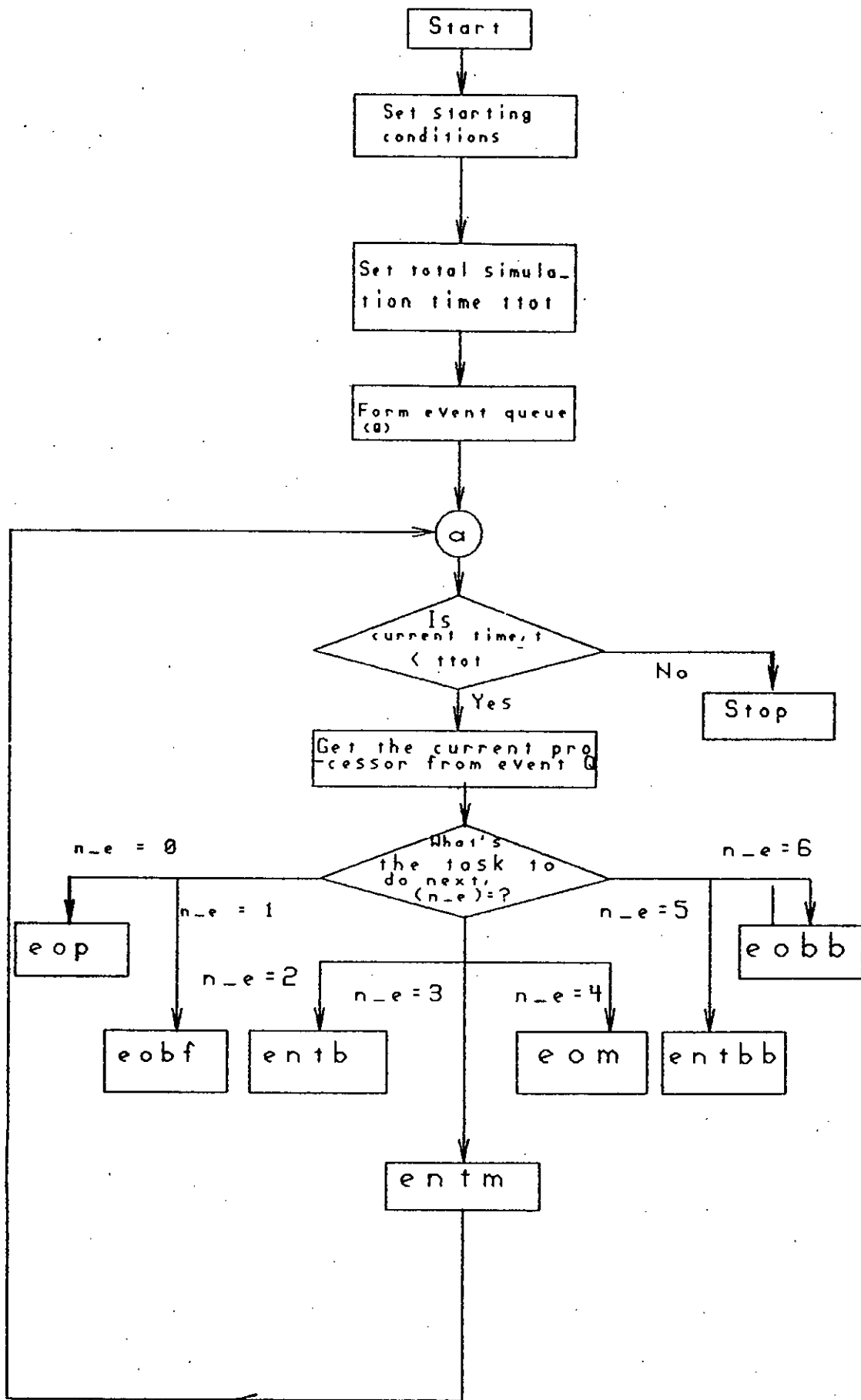


Figure 4.3 Simulation flowchart of packet switched system.

It wants bus for turning back requested block to cache. If the processor wins in bus arbitration, it keeps a bus busy for packet transfer for a fixed bus transfer time say 5 slots. Processor time is updated by bus transfer time so that it can release bus after end of packet transfer through bus and next event is set to 6, i.e. end of bus transfer in returning of requested block (in backward direction) operation. Number of busy processor and busy bus are increased by one. If it fails in bus arbitration it will resubmit request when any one of the buses become free and processor time is incremented so that it can resubmit when a free bus is found. And processor's next event is set to 5, i.e. end of wait for bus.

(vi) End of wait for bus in reverse direction (entbb): The processor resubmit request for bus. If it wins in bus arbitration it keeps the bus busy for returning requested packet to processor cache for a fixed amount of time say 5 slots. Processor's time is updated by this transfer time and its next event is set to 6, i.e. end of bus transfer in reverse direction. The number of busy processors and busy buses are increased by one. If it fails in bus arbitration it will resubmit request when any one of the bus become free. And processor time is incremented so that it can resubmit request when a free bus is found. And processor's next event is set to 5, i.e. end of wait for bus in reverse direction.

(vii) End of bus transfer in reverse direction(eobb): The processor releases bus and enters into processing state. The number of busy bus is decreased by one. The

processor time is updated so that it can submit request after it ends processing. Its next event is set to 0, i.e. processor ends processing. The simulation flowchart is shown in figure 4.3 .

(b) Synchronous System:

First an event queue is made, taking all the processor information as queue element. At the beginning all the processors are kept in processing state. Occurrence of new events and their processor number is checked to serve the processors appropriately. Initialization of clock time is made. If processor ends processing at a time other than at the beginning of system cycle , processor has to wait until beginning of next clock. Processor time is updated so that it can submit request at the beginning of next system cycle and next event is set equal to 2, i.e. end of wait for bus in forward direction. The number of busy processors is decreased by one and queue length is increased by one. If a processor ends processing at the beginning of system cycle it can go to any one of the seven states as described in asynchronous packet switched system.

## CHAPTER 5

### RESULTS AND DISCUSSION

Multiprocessor systems with 16 memory modules, number of processors upto 128 and number of buses in the range 1-16 are investigated. For synchronous system, system cycle is assumed one slot(minor cycle). Uniform and geometric distribution of processing time are considered. Memory connection time is assumed to be constant. For packet switched system bus connection time is also assumed to be constant.

Performances of multiprocessor system with multiple buses are evaluated as shown in graphs 5.1.1 through 5.11.4 . The performance measures plotted in different graphs are shown below:

No. of Figures	Description of Figures
5.1.1, 5.2.1 5.3.1, ... 5.7.1 And 5.10.1, 5.11.1	Average Queue Length vs. no of Buses
5.1.2, 5.2.2 5.2.2, ... 5.7.2 And 5.10.2, 5.11.2	Processor Utilization vs. no of Buses
5.1.3, 5.2.3 5.3.3, ... 5.7.3 And 5.8, 5.9, ... 5.11.3	Memory Bandwidth vs. no of Buses
5.1.4, 5.2.4 5.3.4, .. 5.7.4 And 5.10.4, 5.11.4	Bus Utilization vs. no of Buses

81463

In crossbar interconnection network there is no bus conflict. The only conflict is memory conflict, which happens

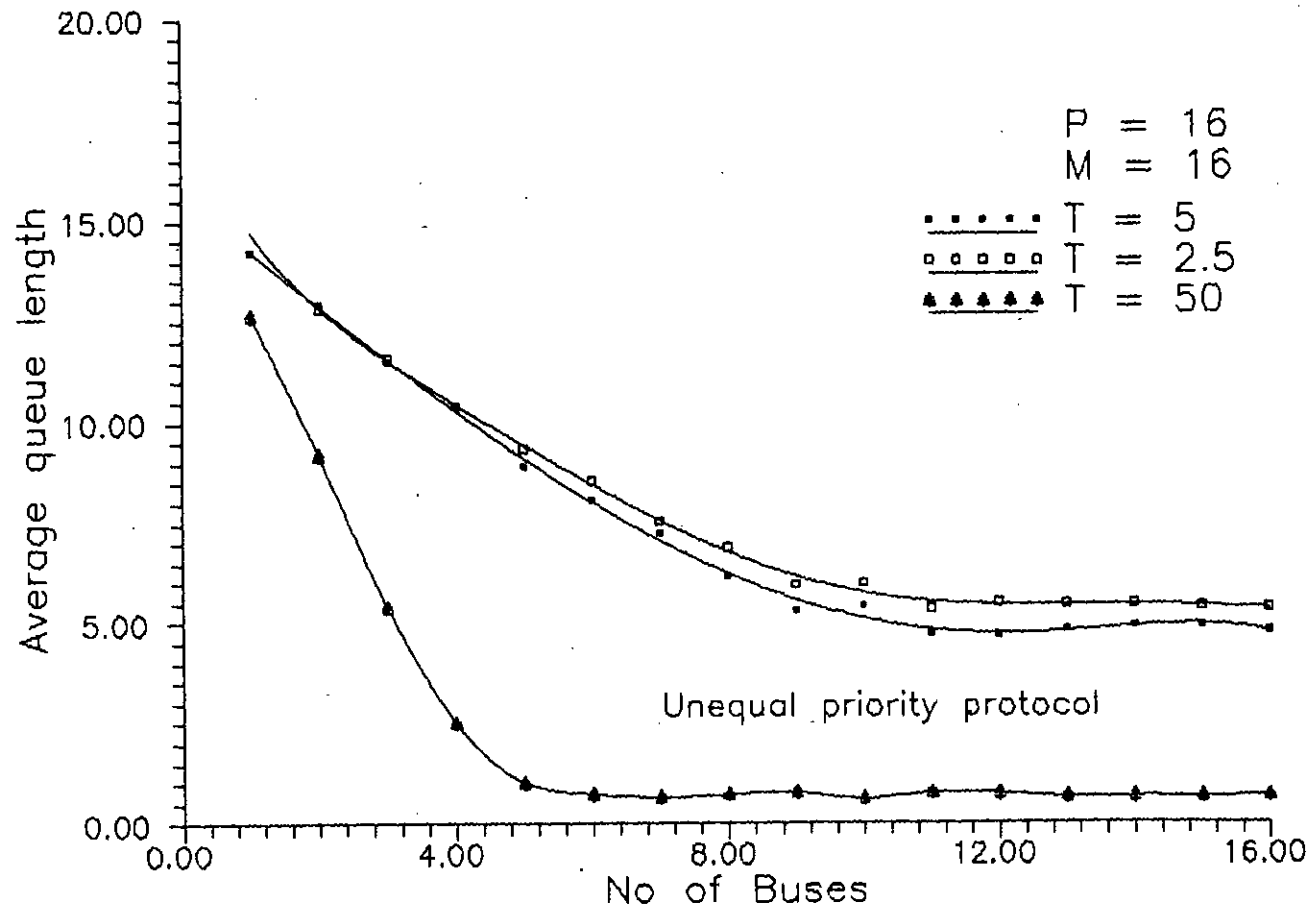


Figure 5.1.1: Average queue length vs number of buses for asynchronous circuit switched system.

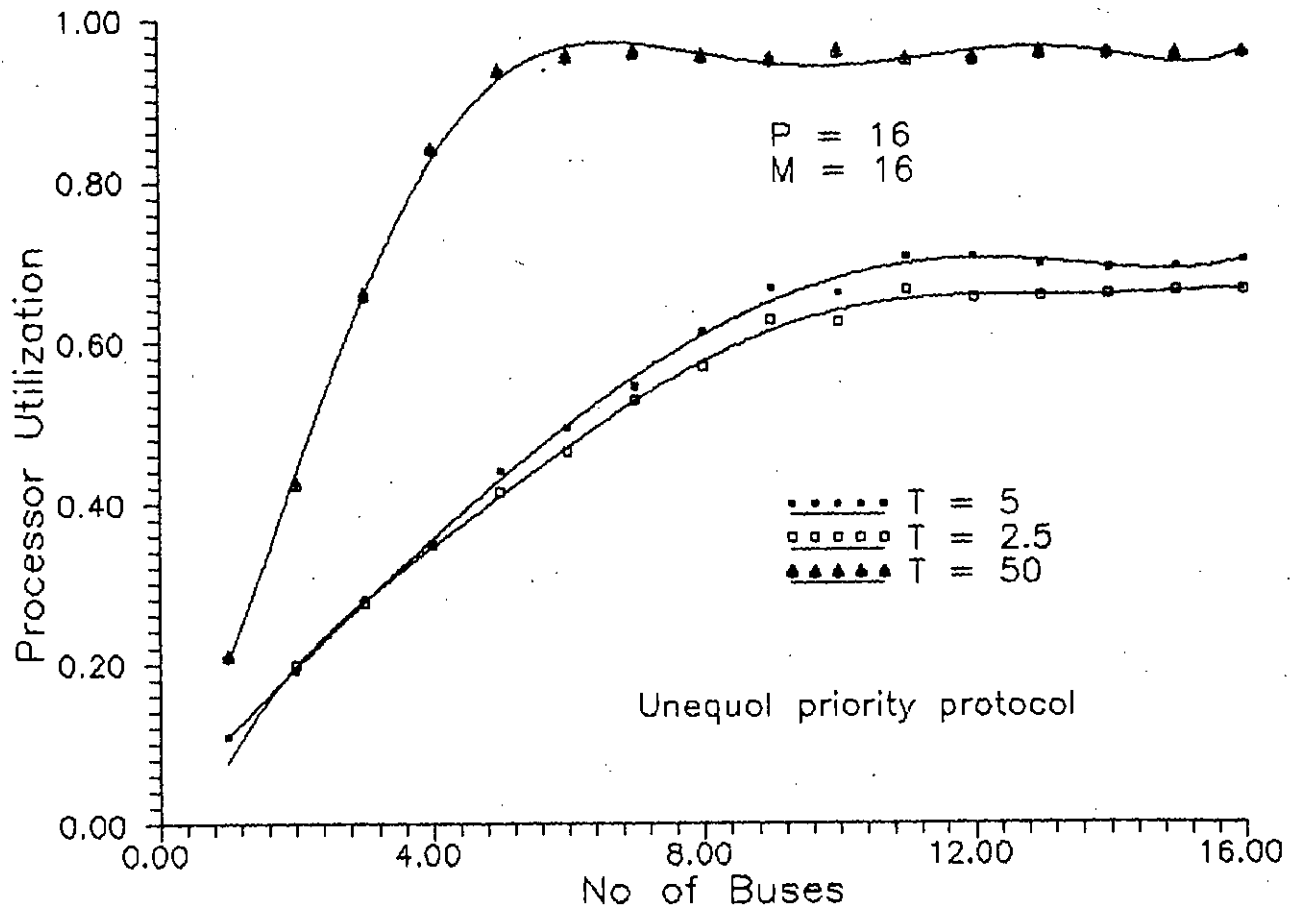


Figure 5.1.2: Processor utilization vs number of buses for asynchronous circuit switched system.



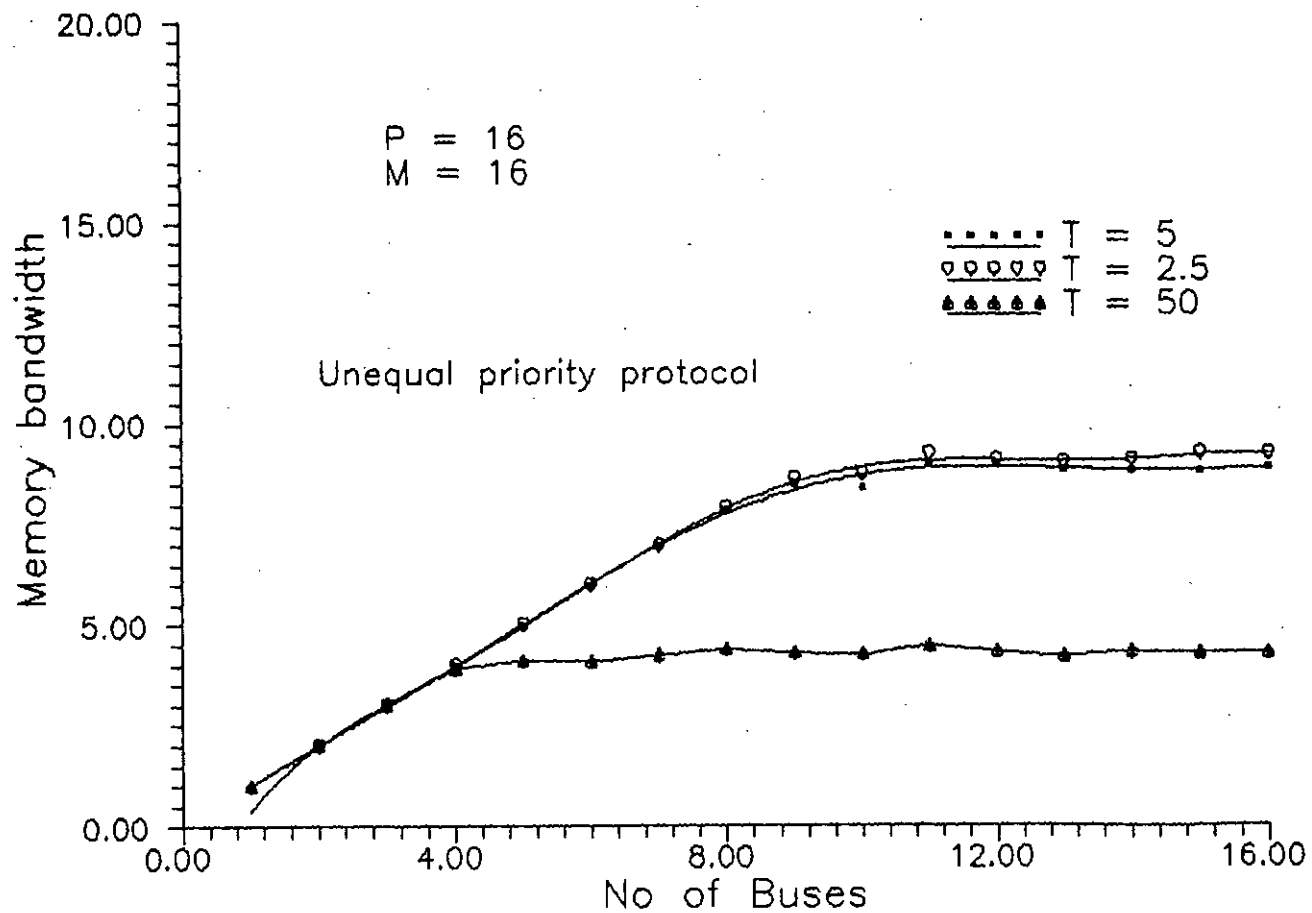


Figure 5.1.3: Memory bandwidth vs number of buses for asynchronous circuit switched system.

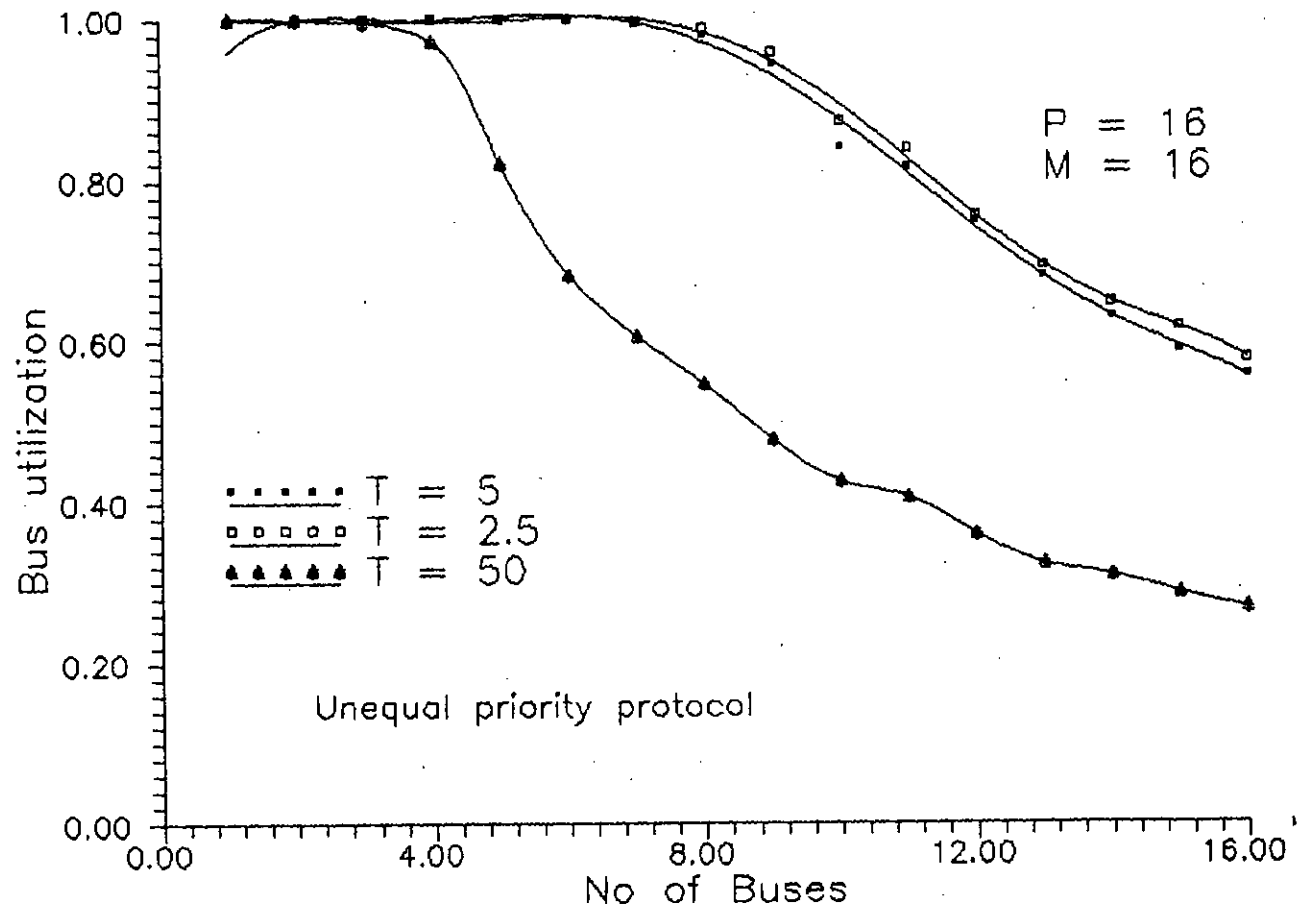


Figure 5.1.4: Bus utilization vs number of buses for asynchronous circuit switched system.

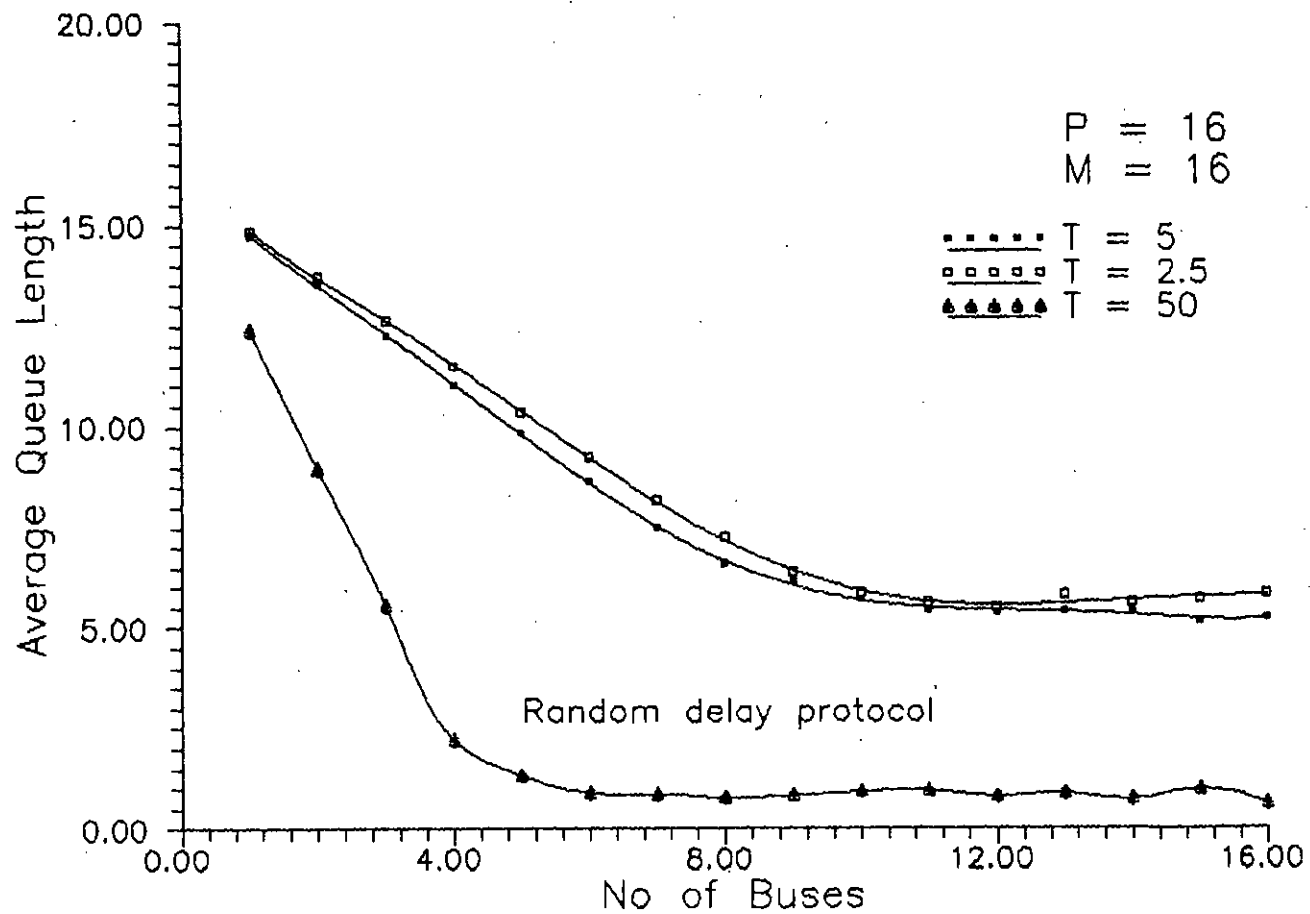


Figure 5.2.1 Average queue length vs number of buses for asynchronous circuit switched system.

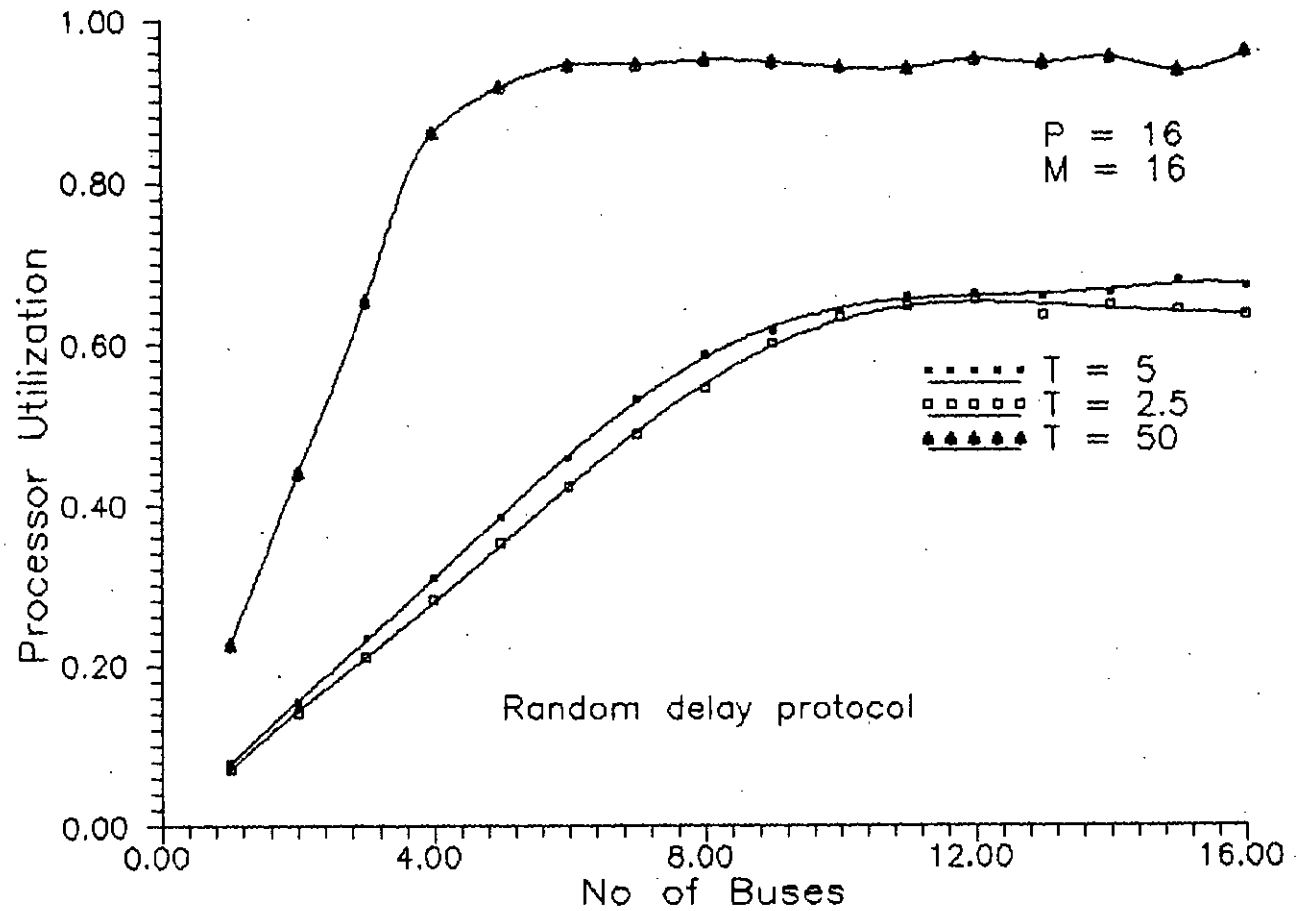


Figure 5.2.2: Processor utilization vs number of buses for asynchronous circuit switched system.

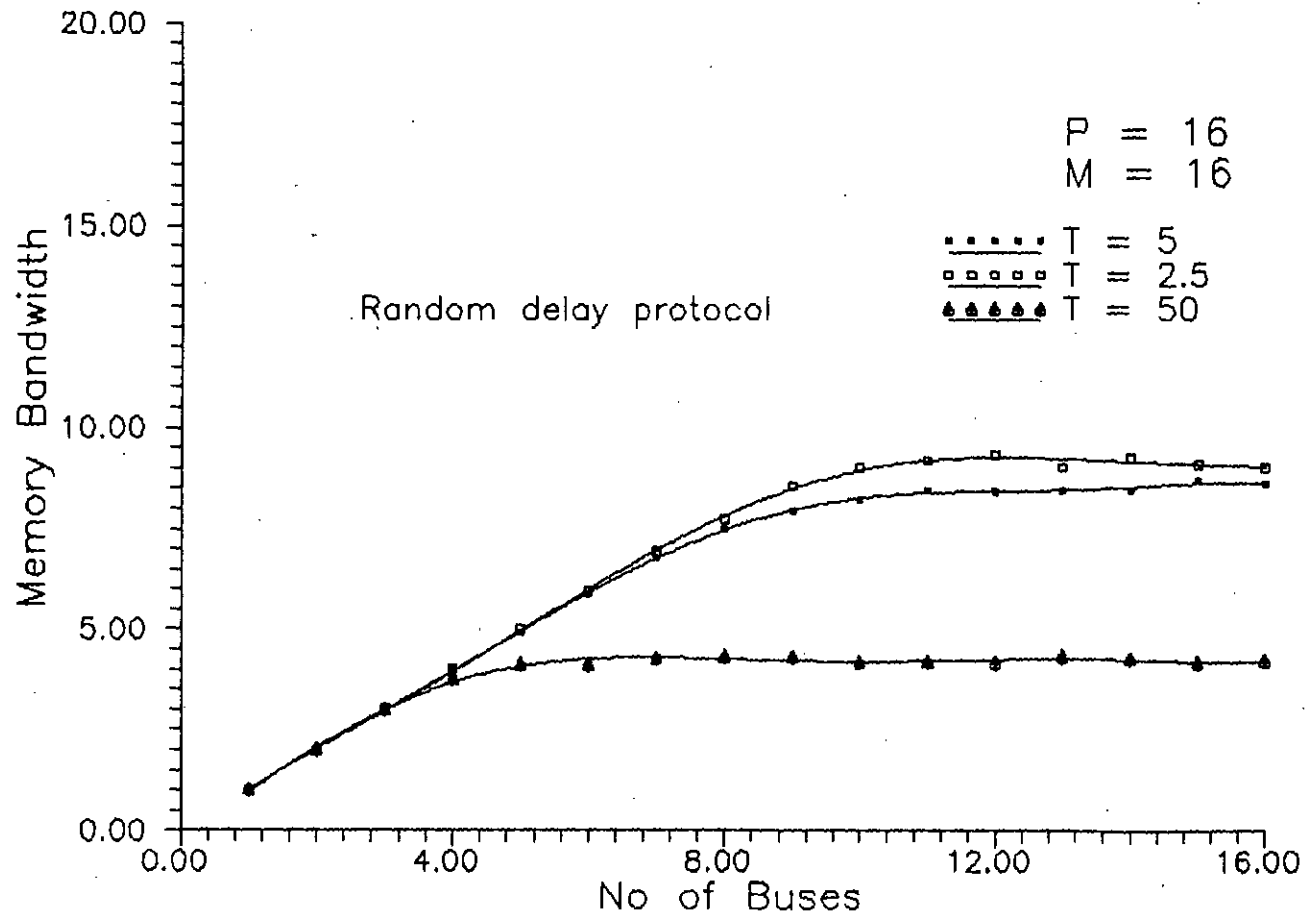


Figure 5.2.3: Memory bandwidth vs number of buses for asyn-circuit switched system.

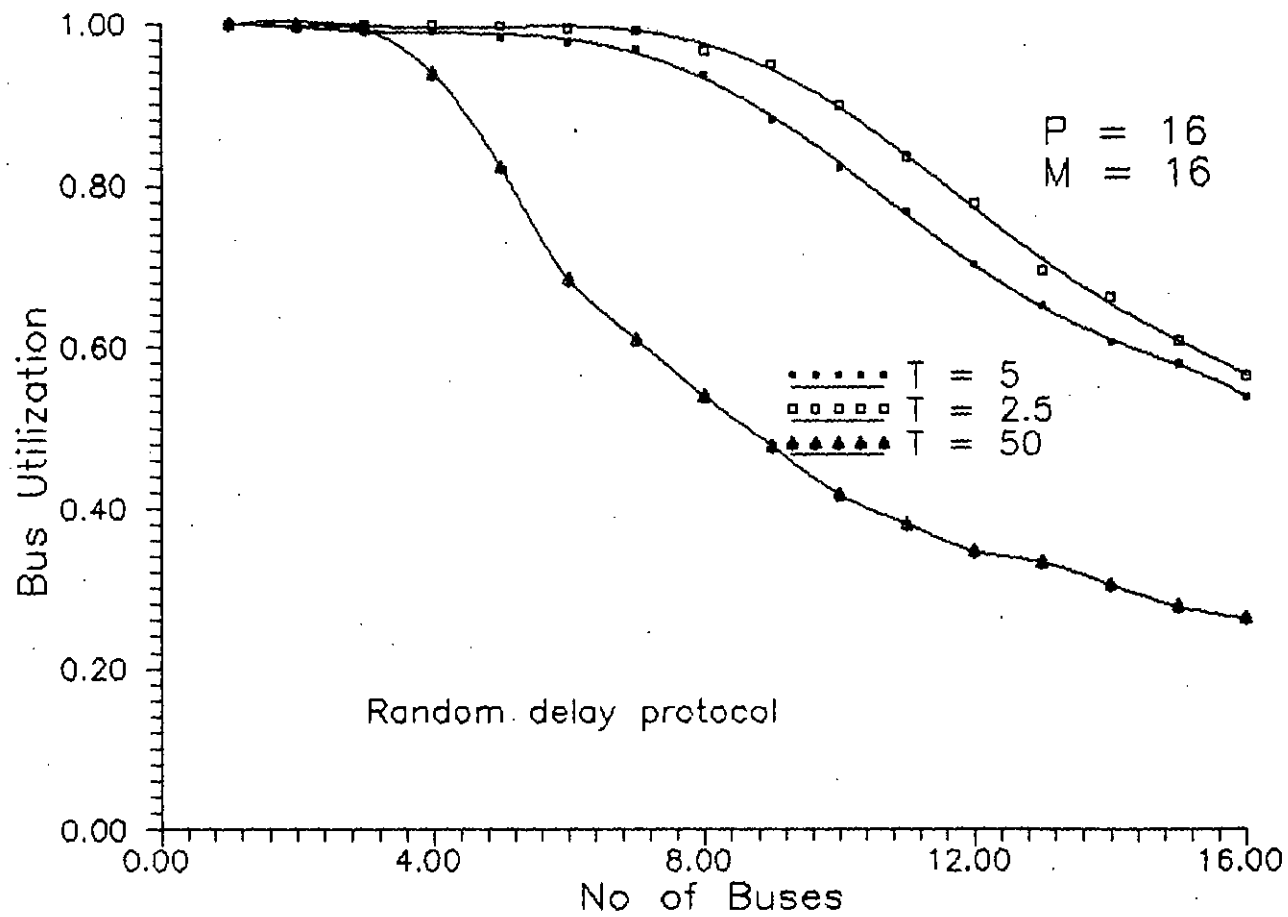


Figure 5.2.4: Bus utilization vs number of buses for asynchronous circuit switched system.

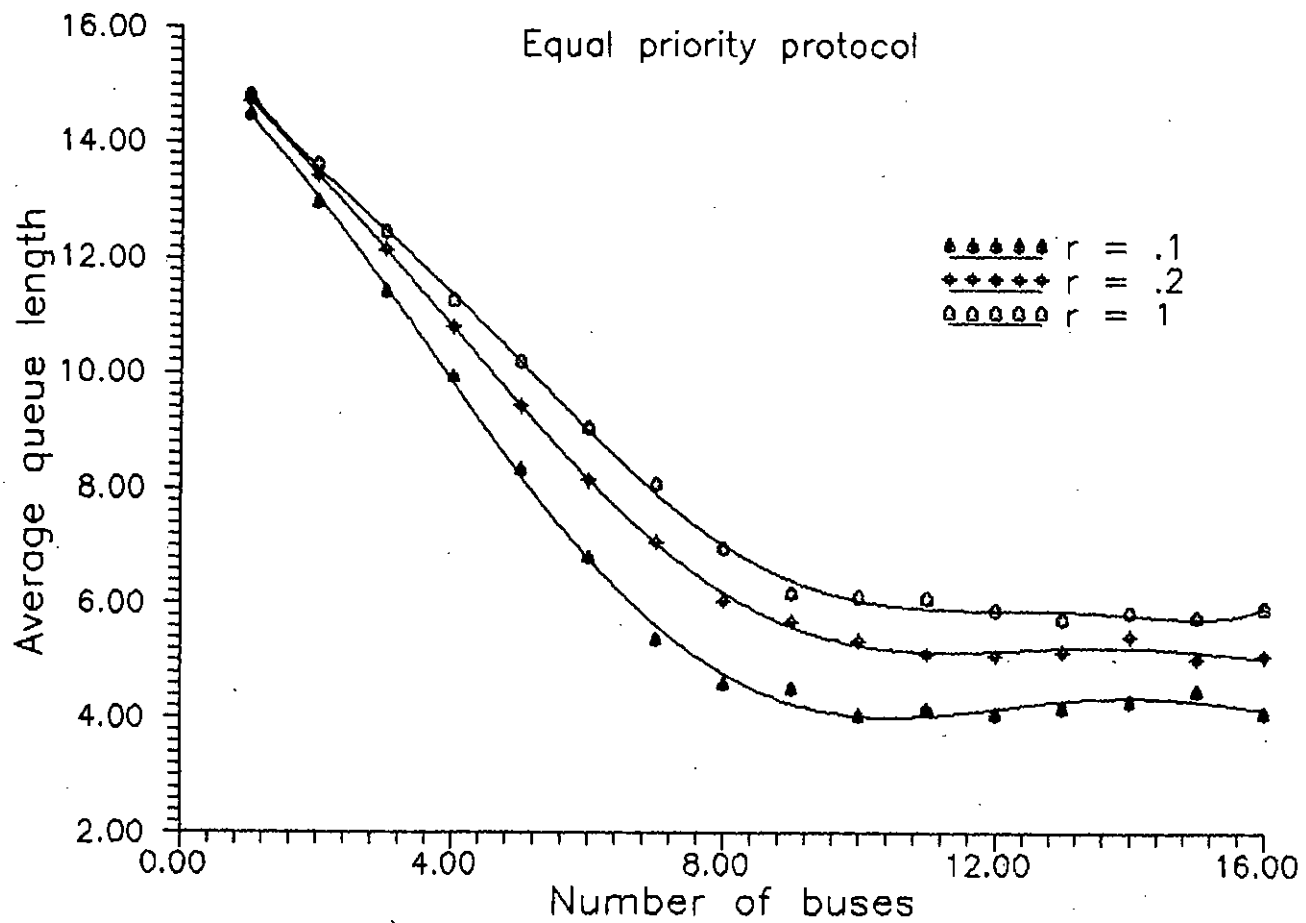


Figure 5.3.1: Average queue length vs number of buses for synchronous circuit switched system.

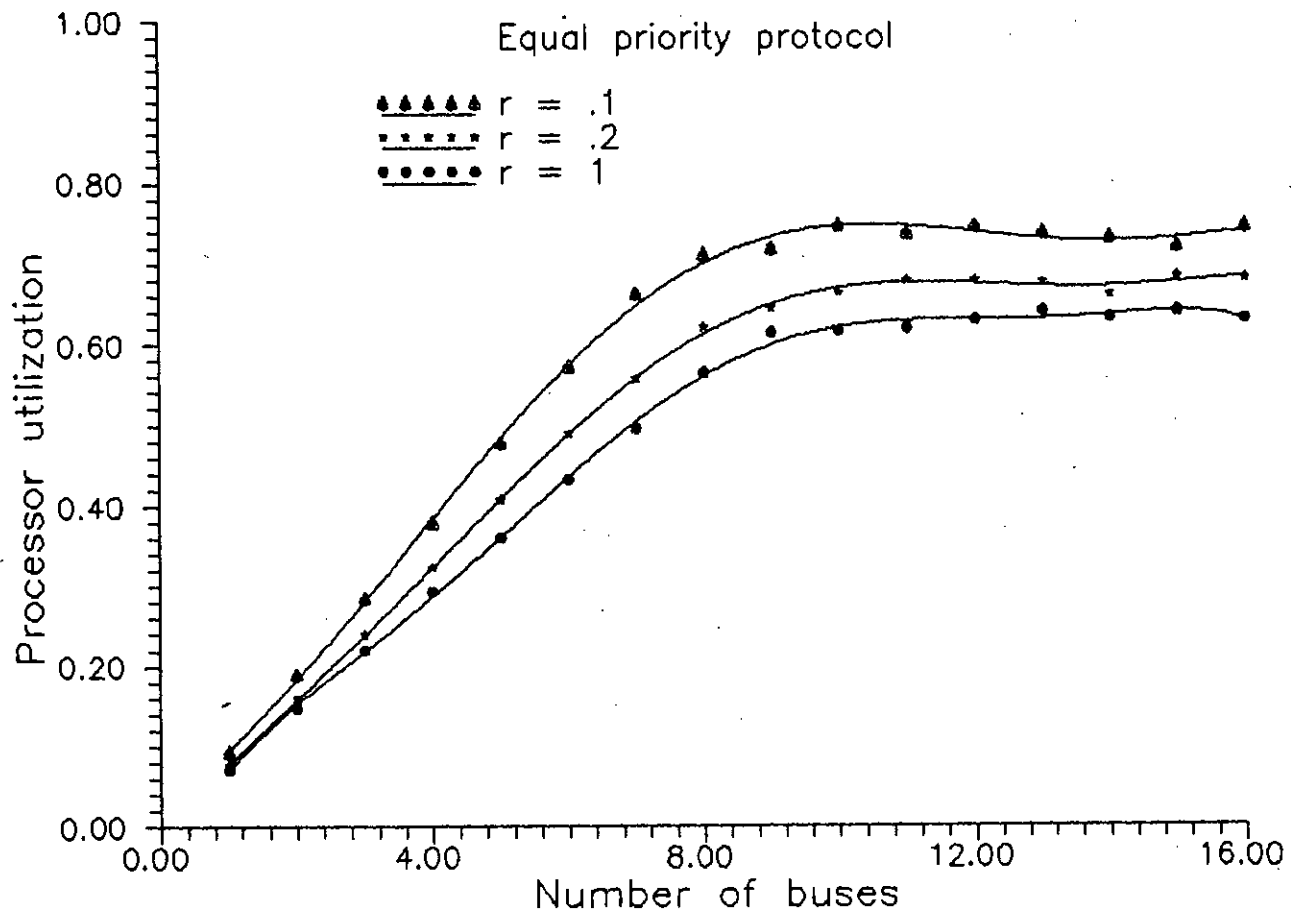


Figure 5.3.2: Processor utilization vs number of buses for synchronous circuit switched system.



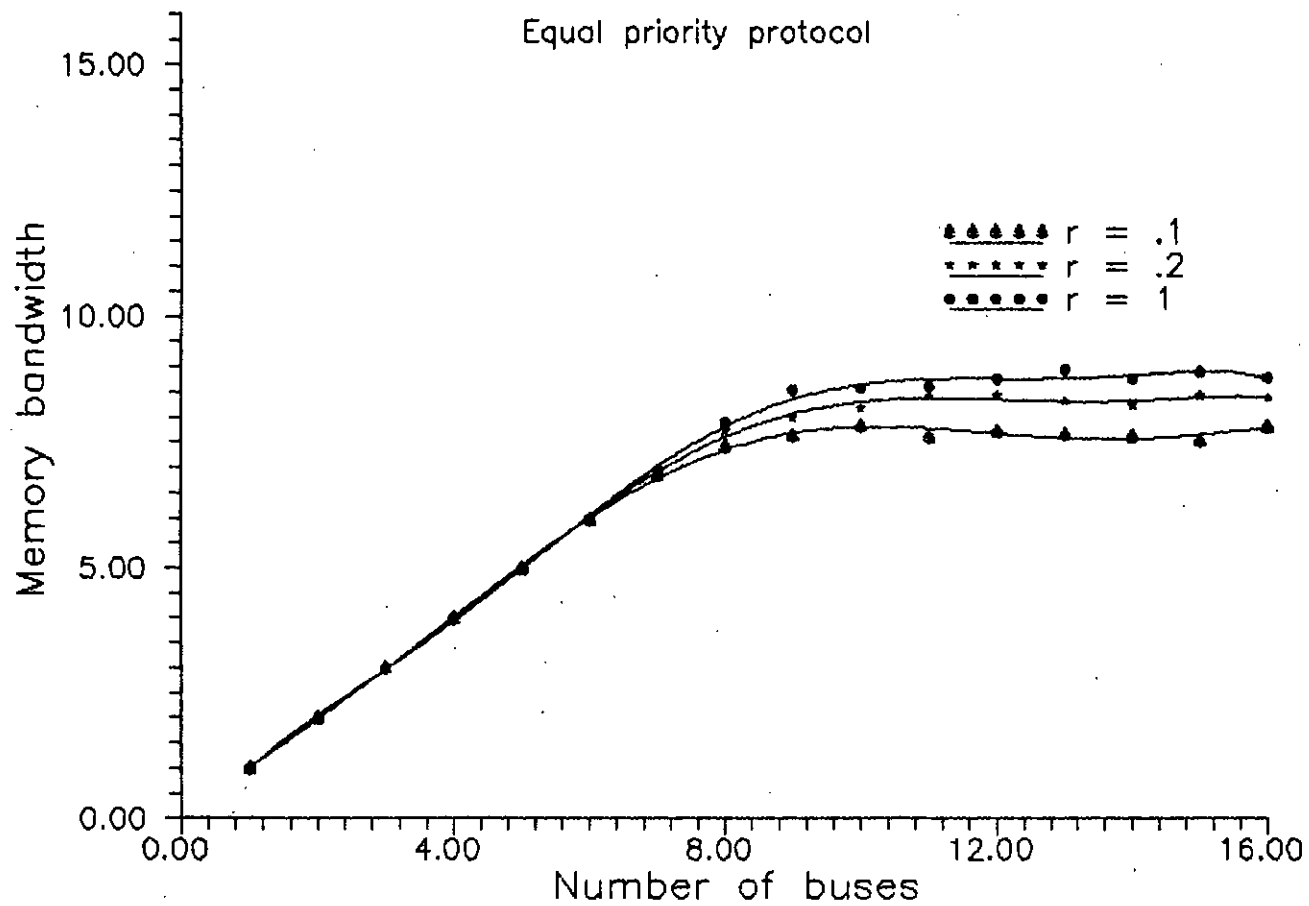


Figure 5.3.3: Memory bandwidth vs number of buses for synchronous circuit switched system.

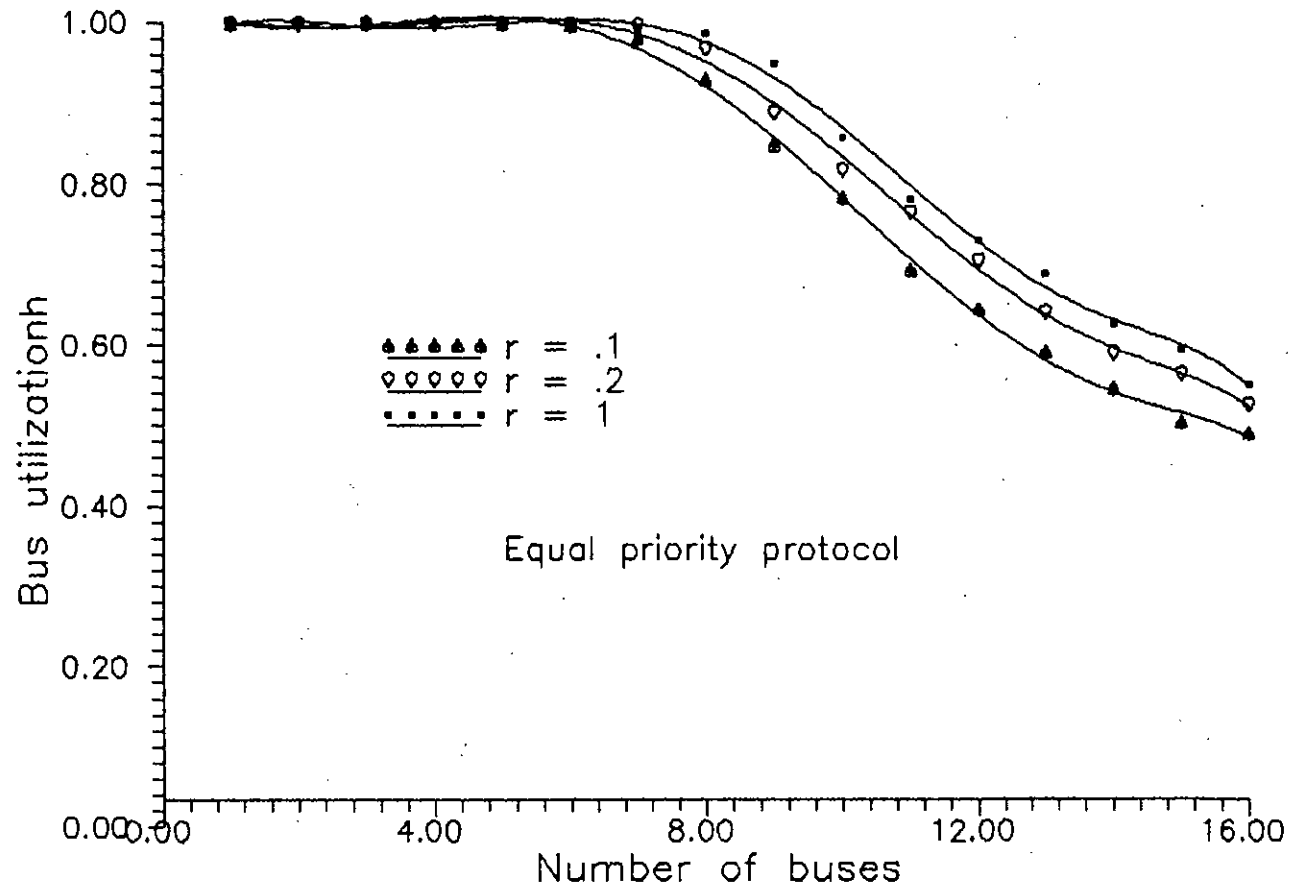


Figure 5.3.4: Bus utilization vs number of buses for circuit switched synchronous system.

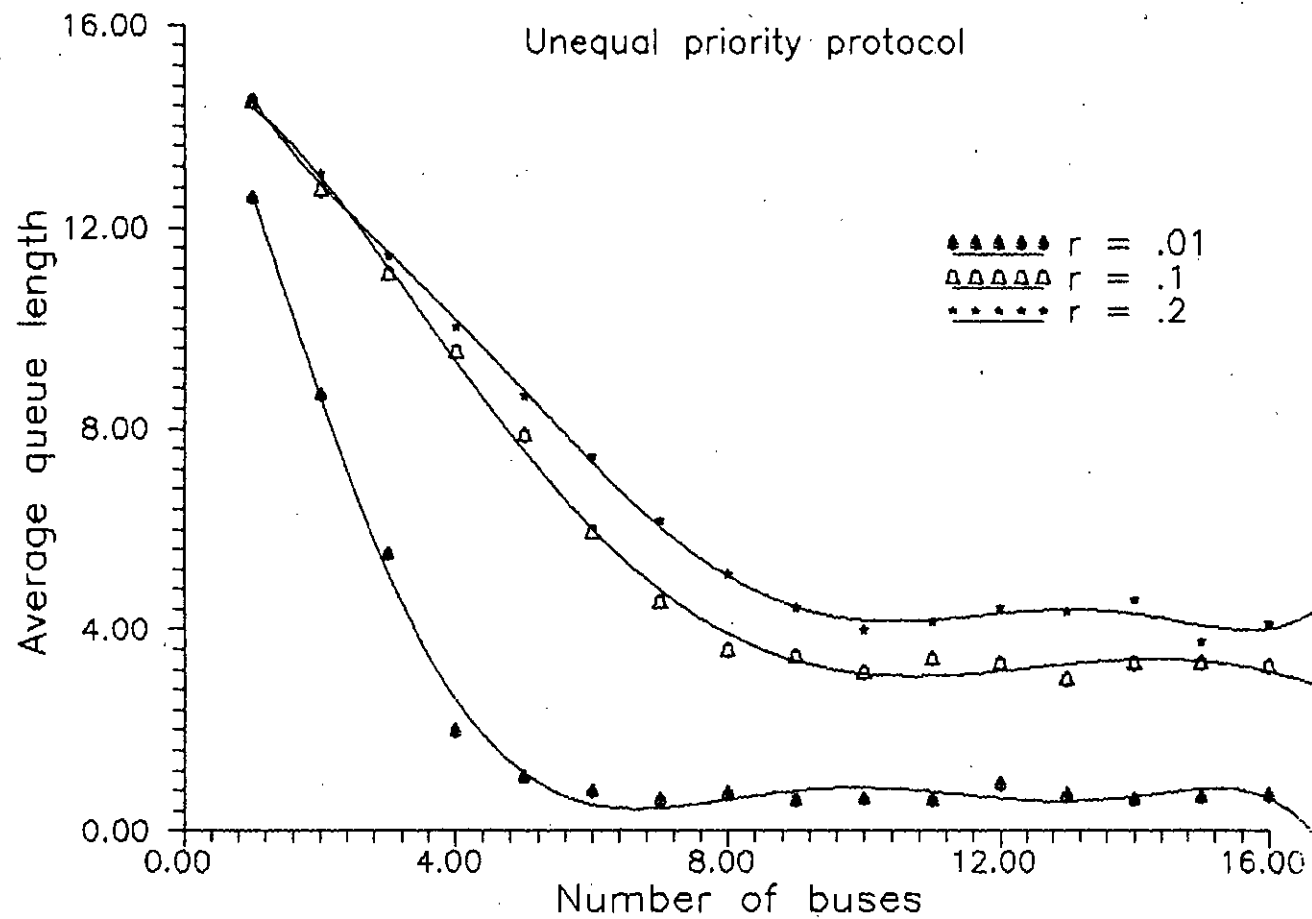


Figure 5.4.1: Average queue length vs number of buses for synchronous circuit switched system.

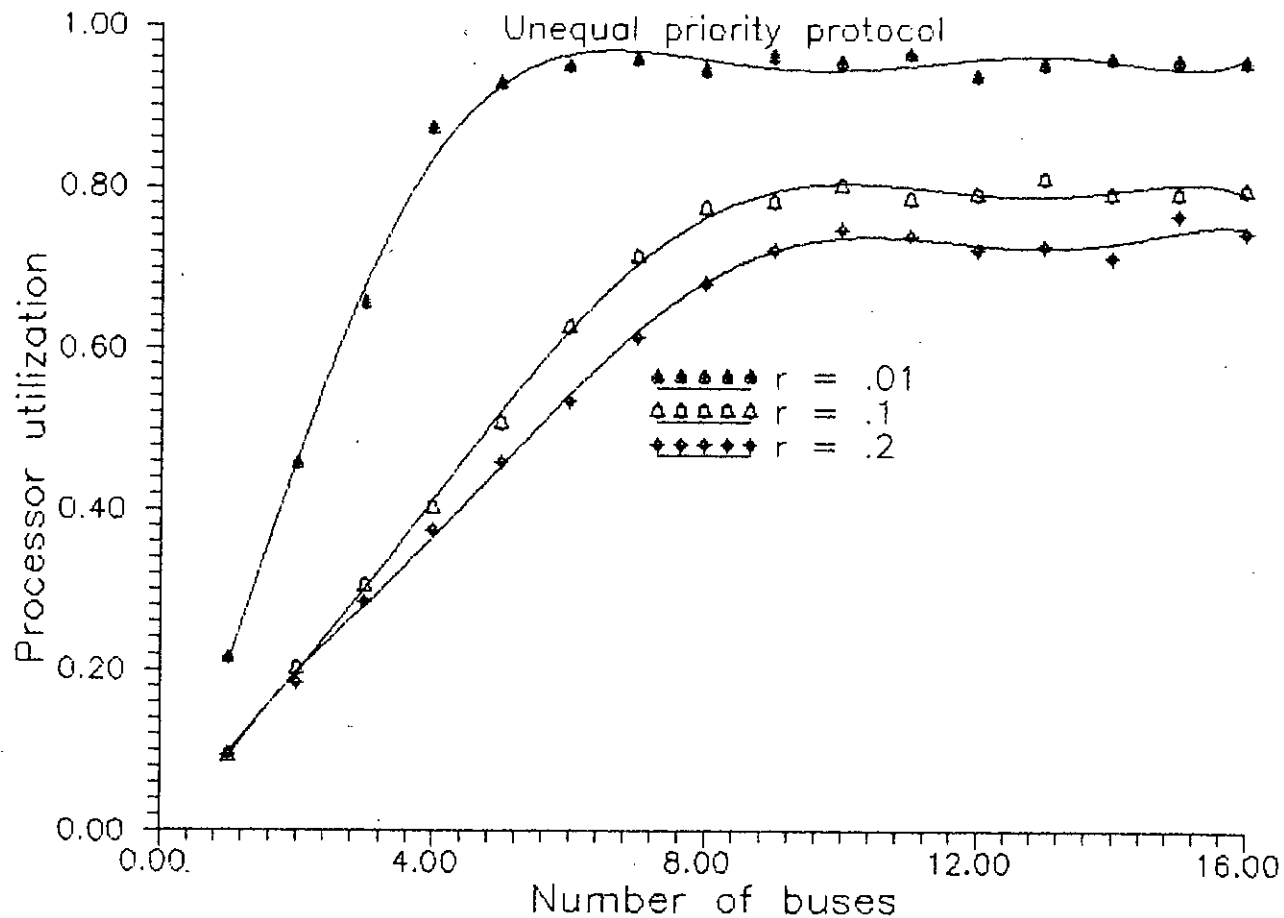


Figure 5.4.2; Processor utilization vs number of buses for synchronous circuit switched system.

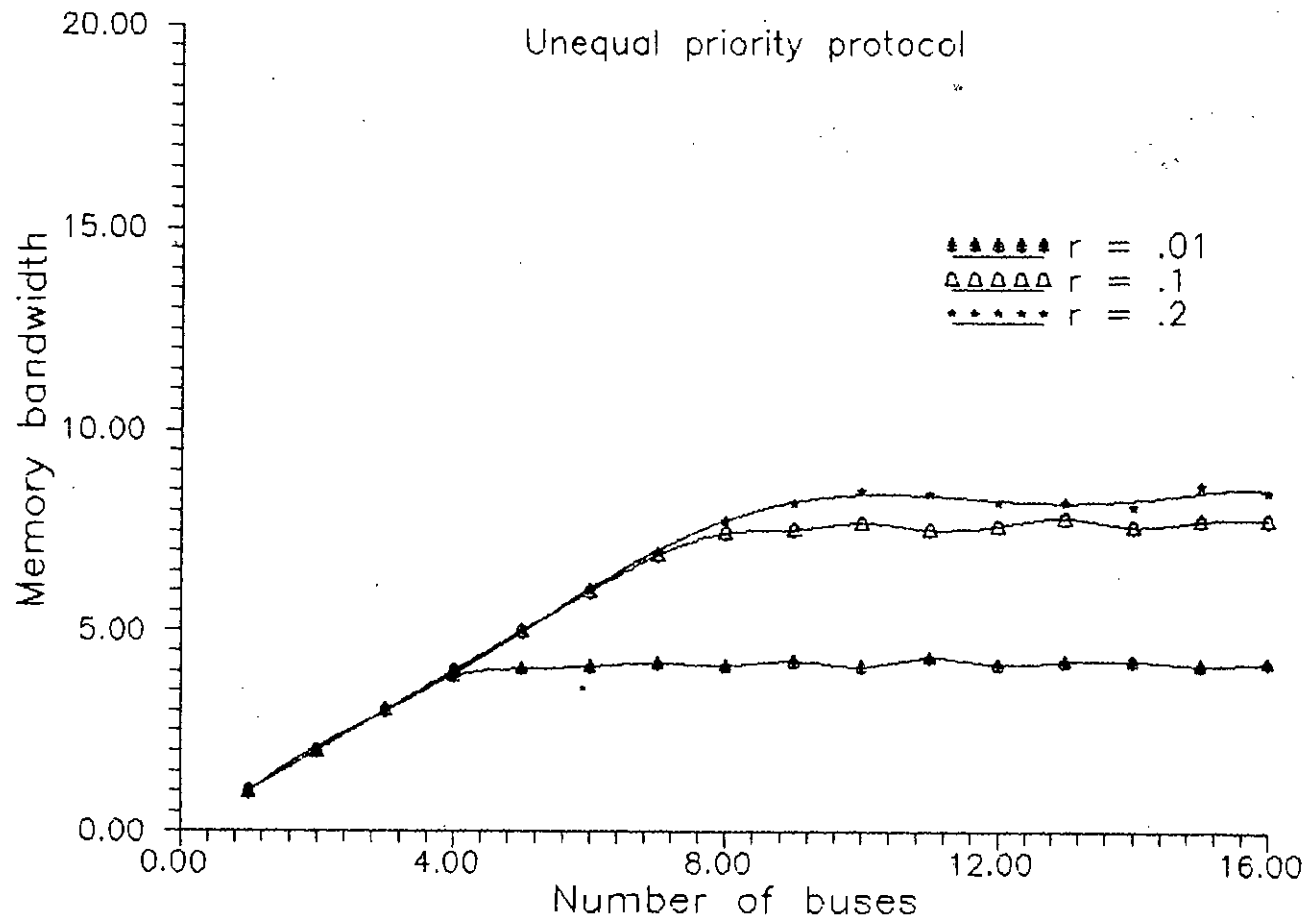


Figure 5.4.3: Memory bandwidth vs number of buses for synchronous circuit switched system.

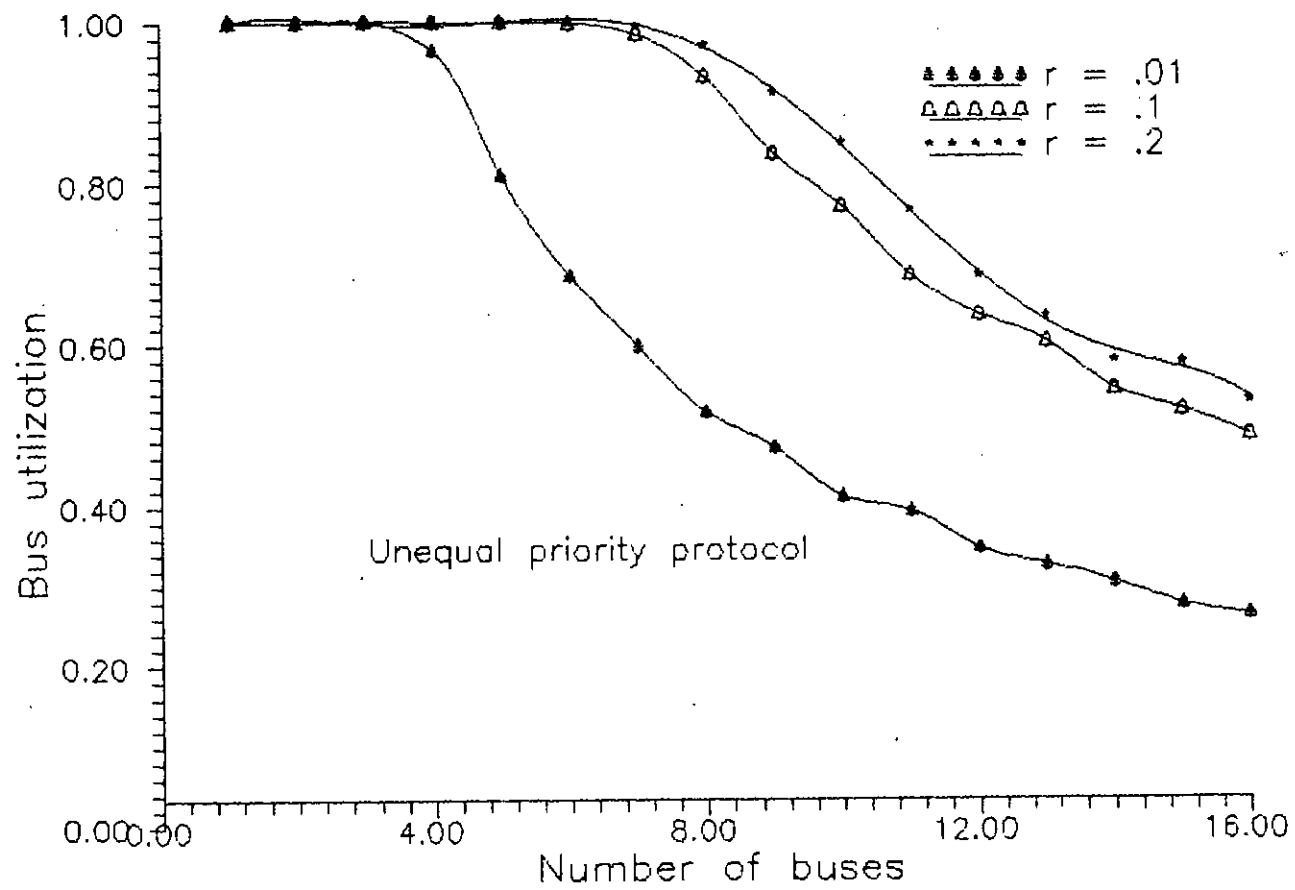


Figure 5.4.4: Bus utilization vs number of buses for synchronous circuit switched system.

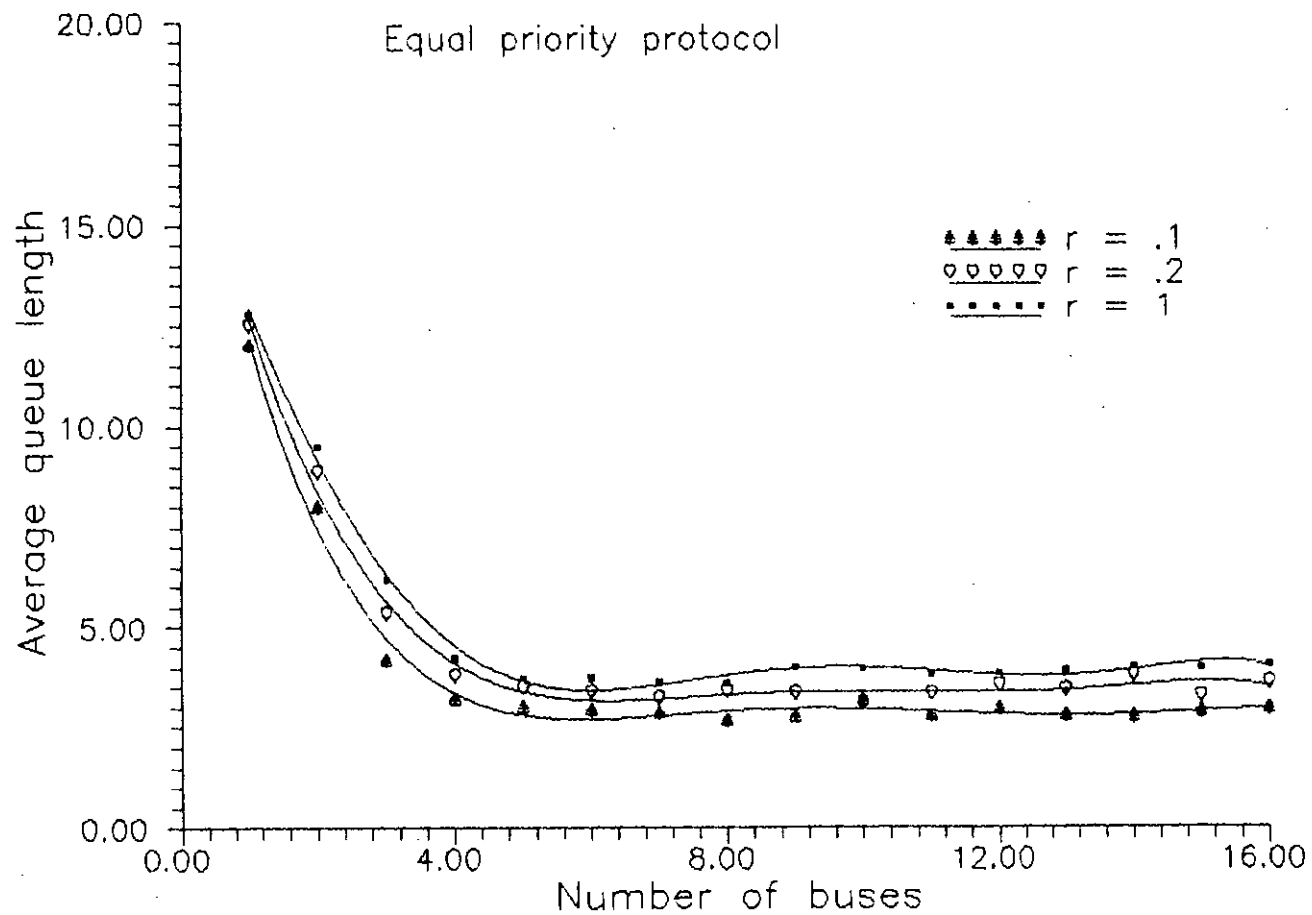


Figure 5.5.1: Average queue length vs number of buses for packet switched synchronous system.

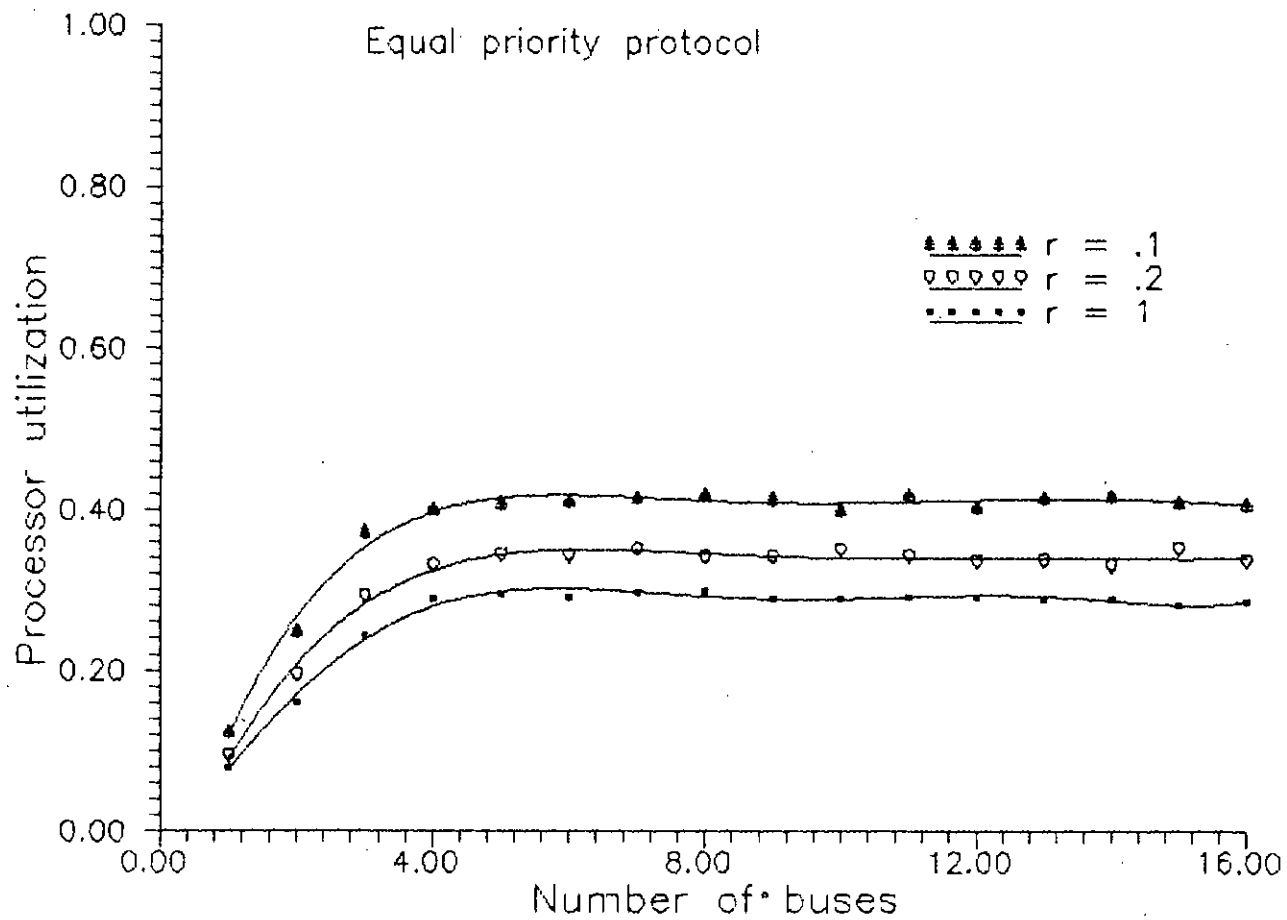


Figure 5.5.2: Processor utilization vs number of buses for packet switched synchronous system.



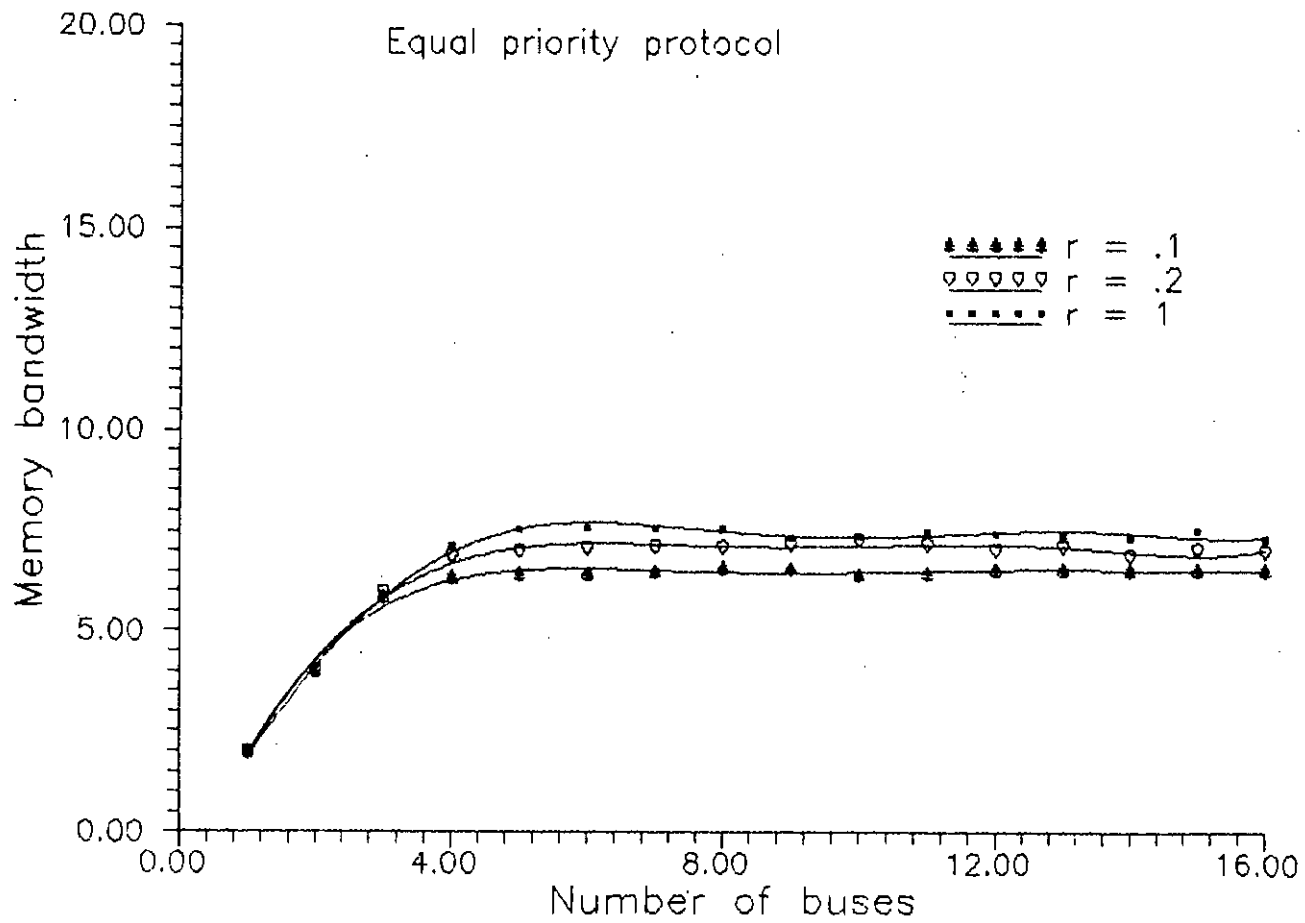


Figure 5.5.3: Memory bandwidth vs number of buses for packet switched synchronous system.

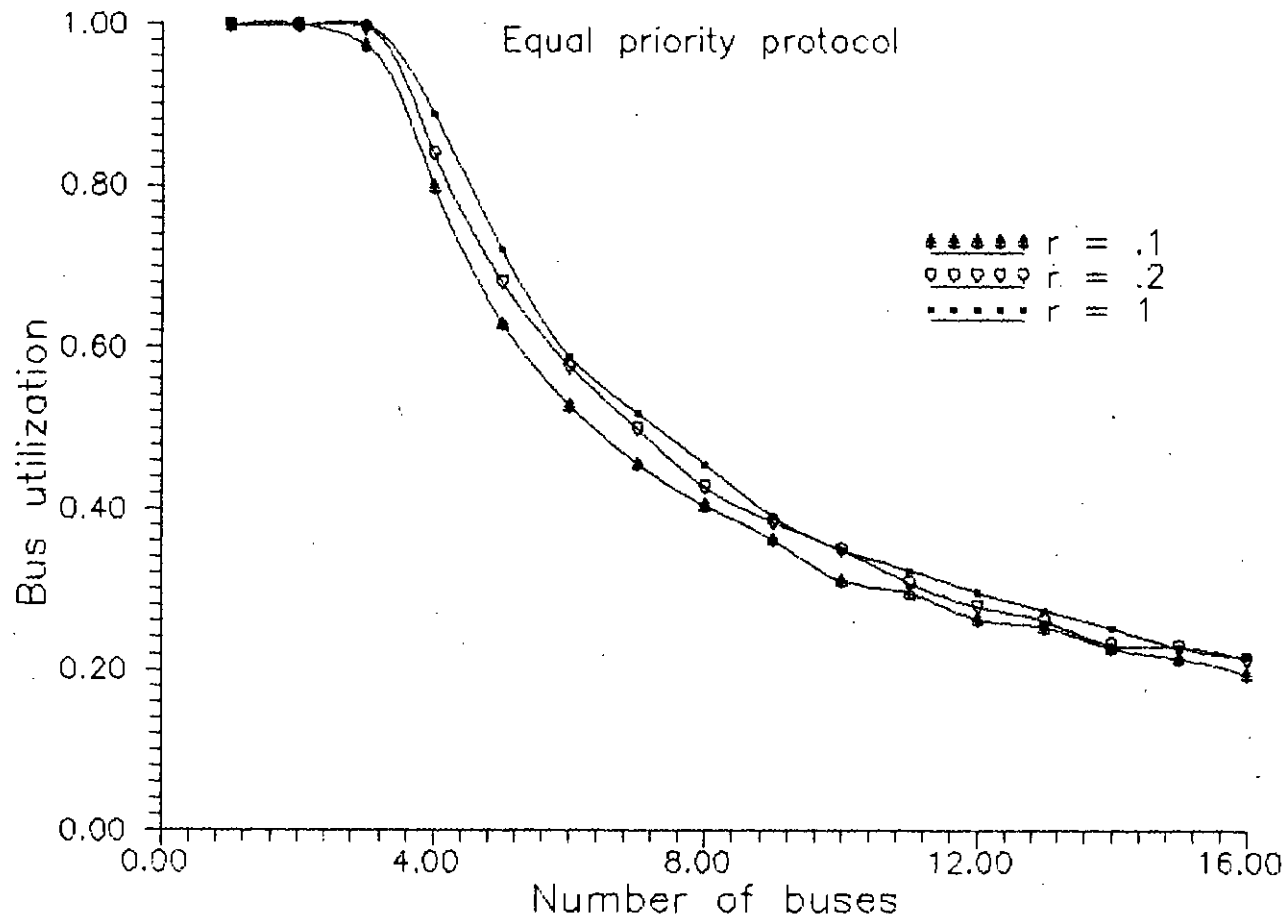


Figure 5.5.4: Bus utilization vs number of buses for packet switched synchronous system.

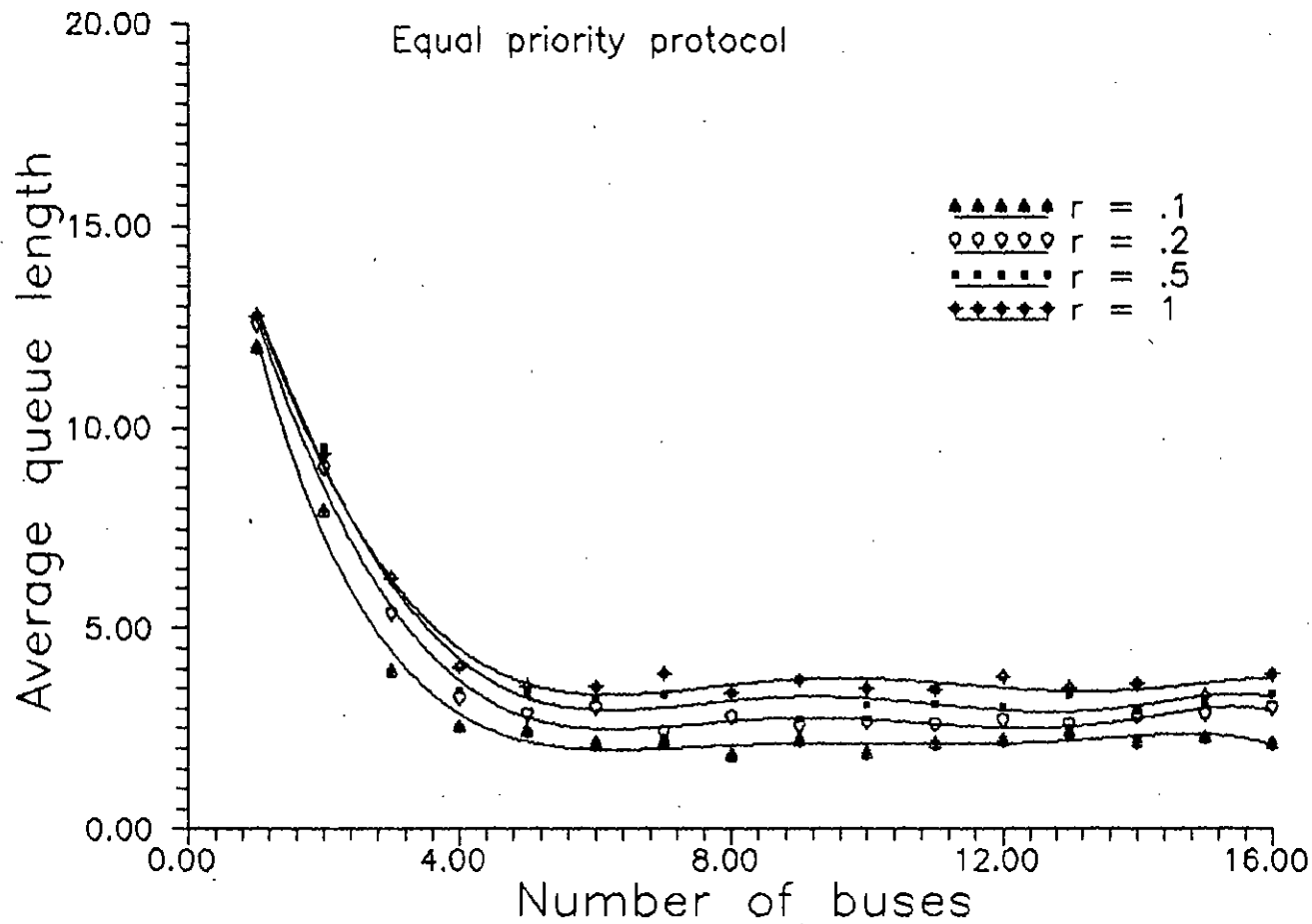


Figure 5.6.1: Average queue length vs number of buses for packet switched asynchronous system.

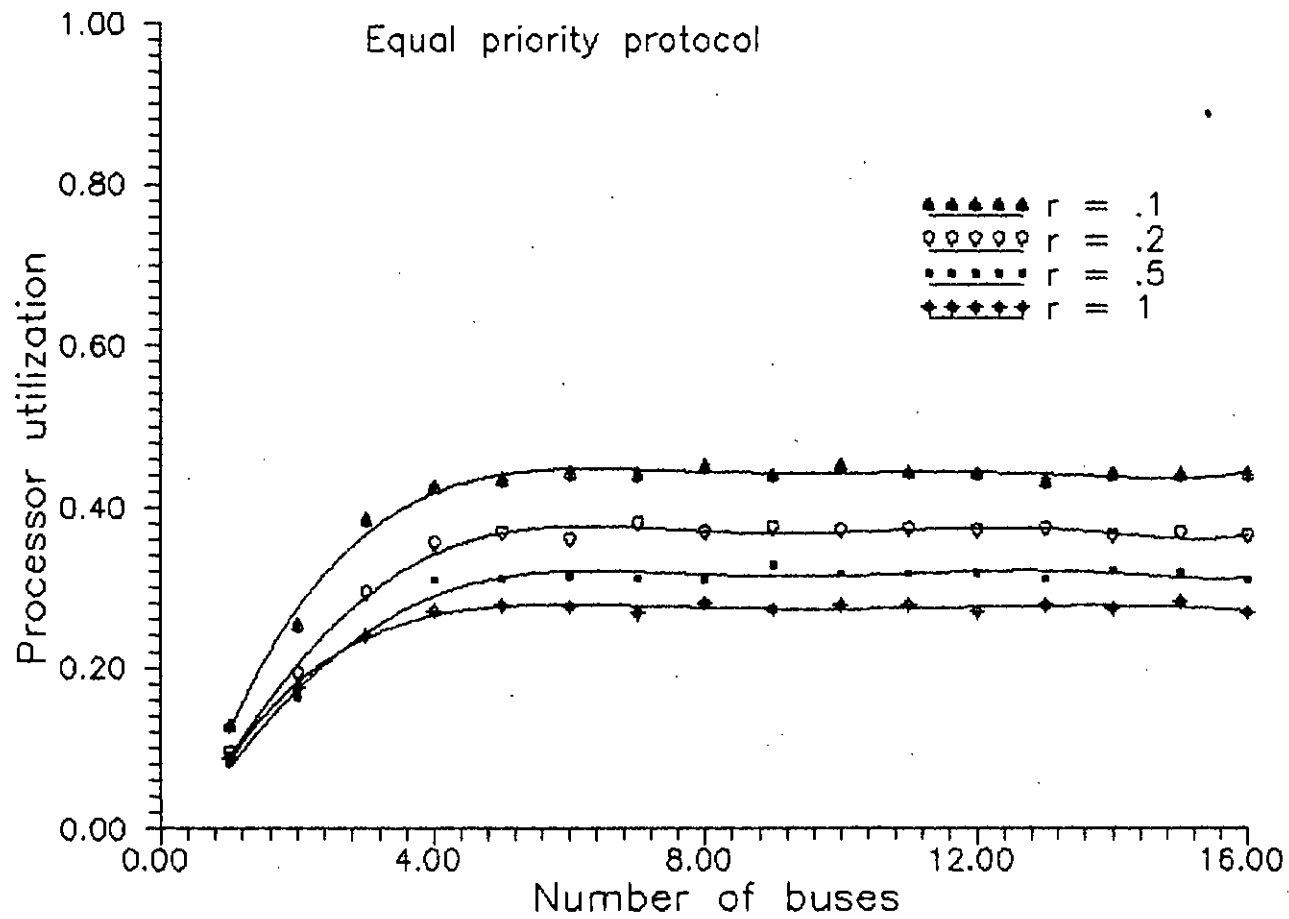


Figure 5.6.2: Processor utilization vs number of buses for packet switched asynchronous system.

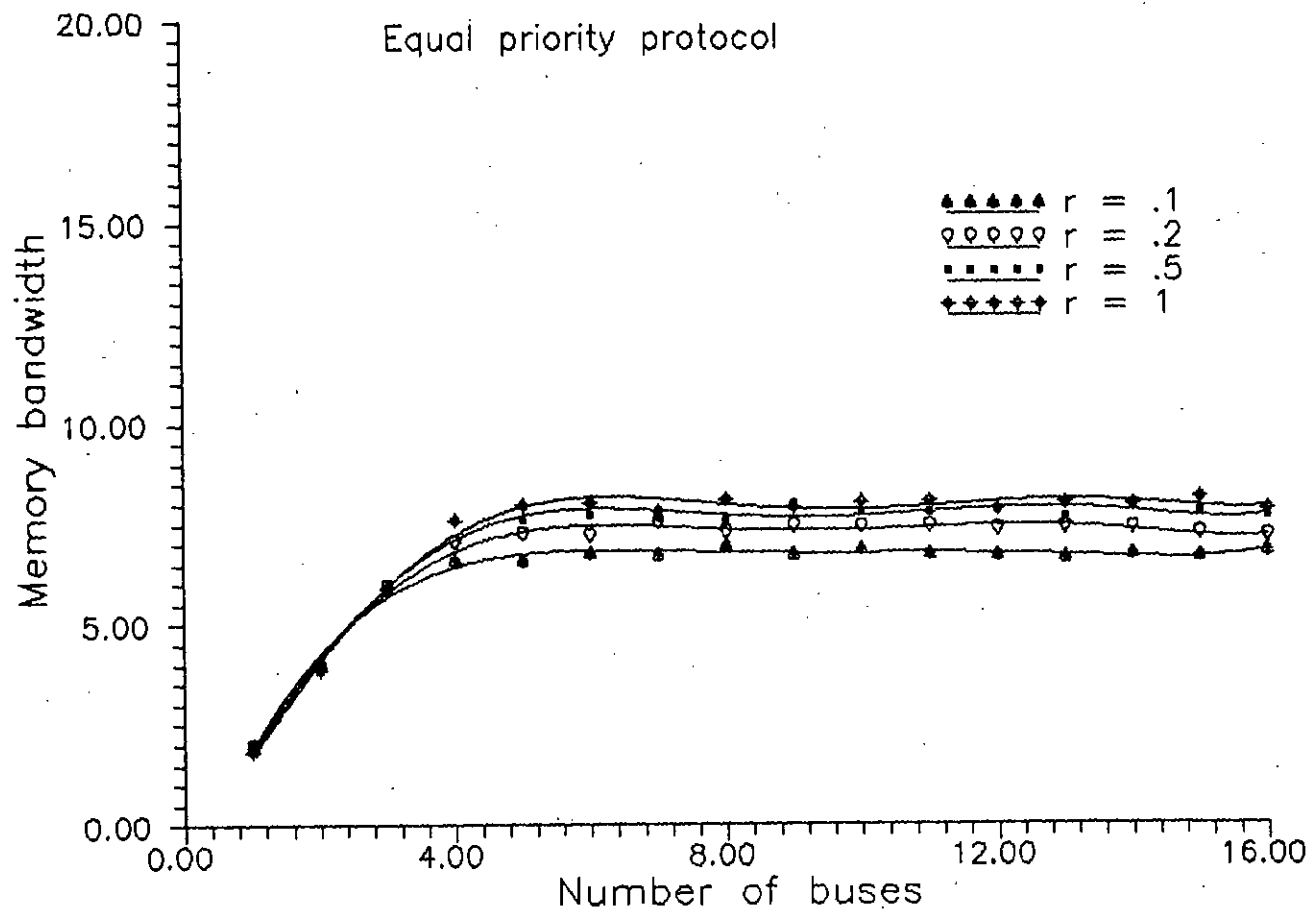


Figure 5.6.3: Memory bandwidth vs number of buses for packet switched asynchronous system.

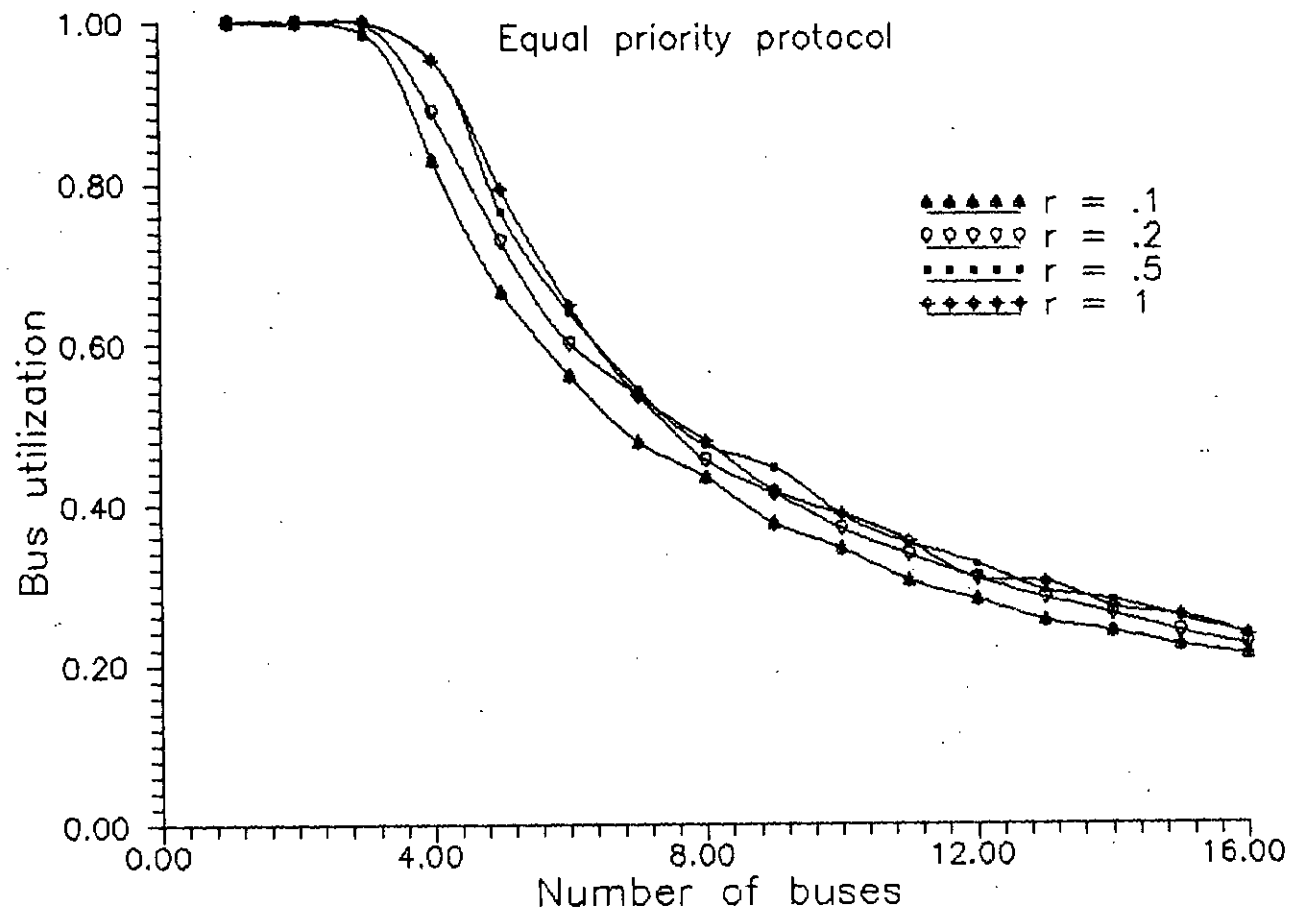


Figure 5.6.4: Bus utilization vs number of buses for packet switched asynchronous system.

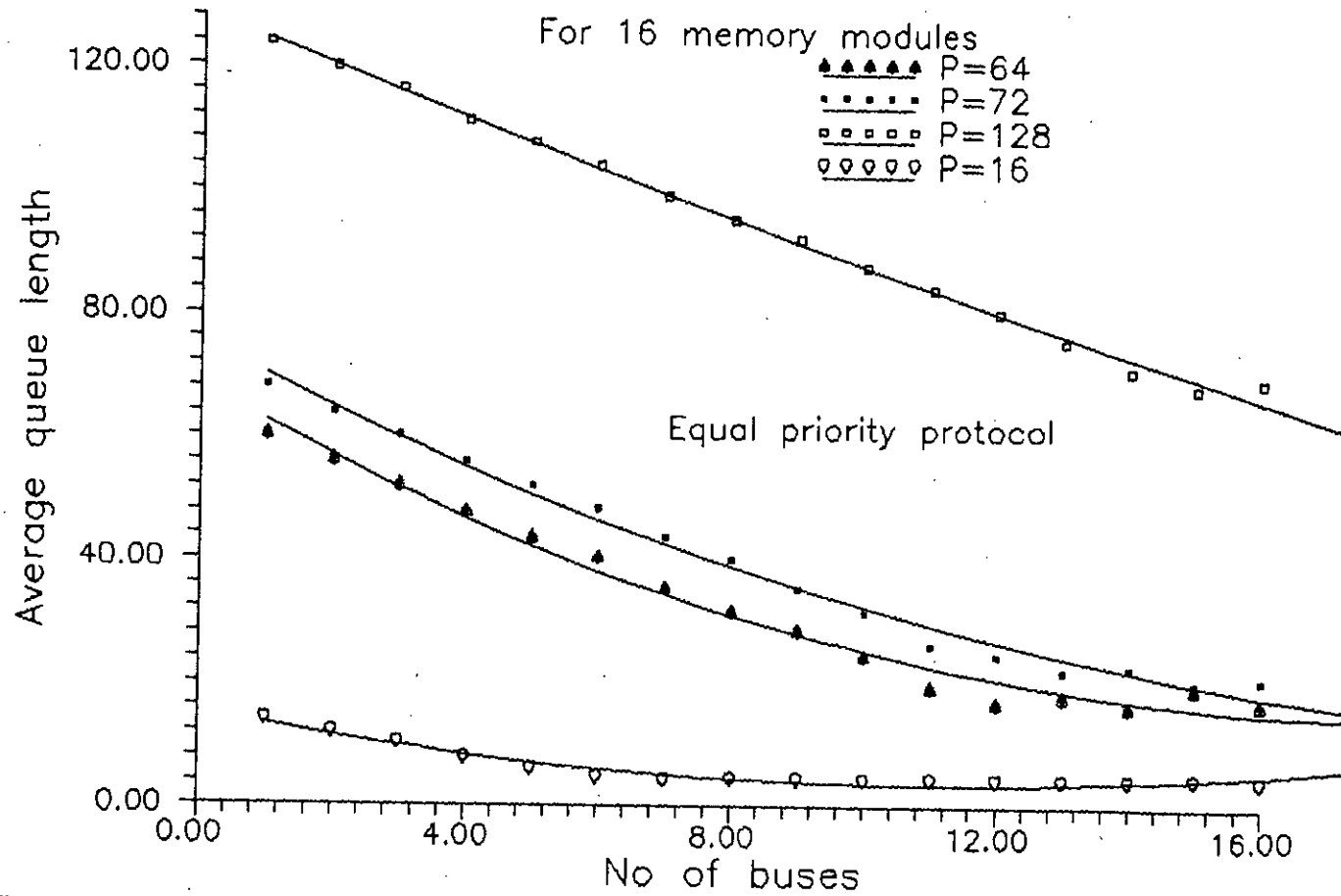


Figure 5.7.1: Average queue length vs number of buses with variable number of processors.

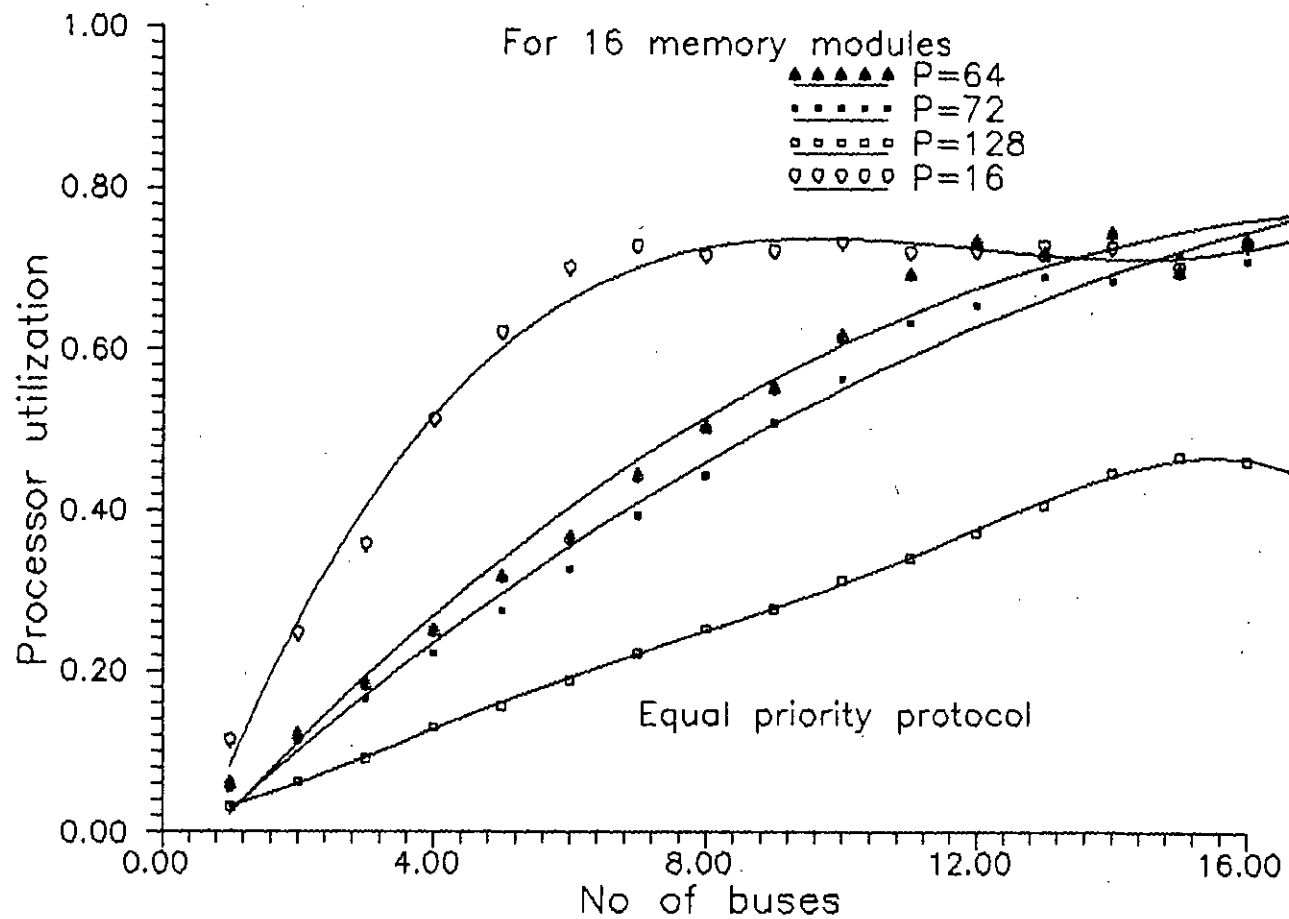


Figure 5.7.2: Processor utilization vs number of buses with variable number of processors.



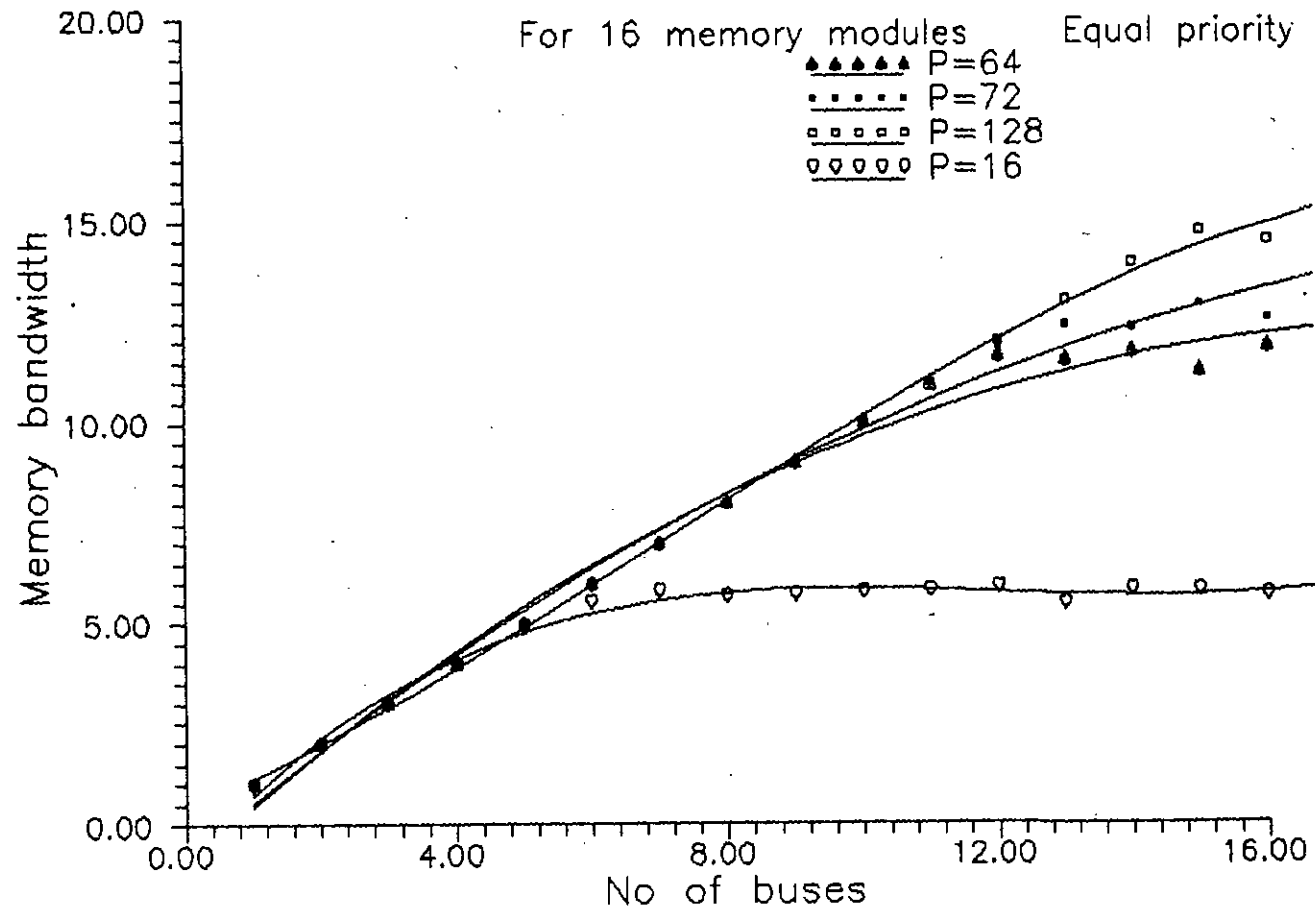


Figure 5.7.3: Memory bandwidth vs number of buses with variable number of processors.

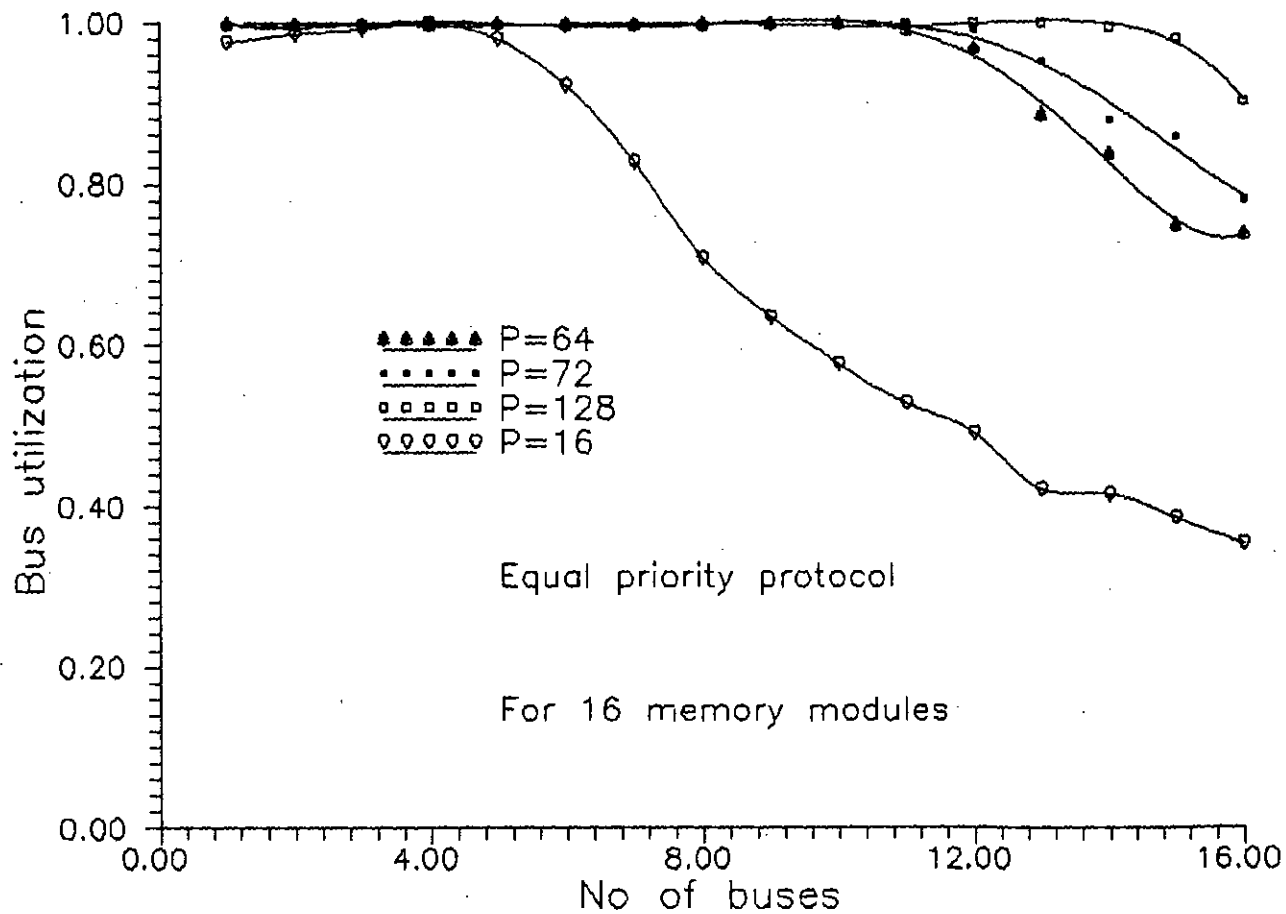


Figure 5.7.4: Bus utilization vs number of buses with variable number of processors.

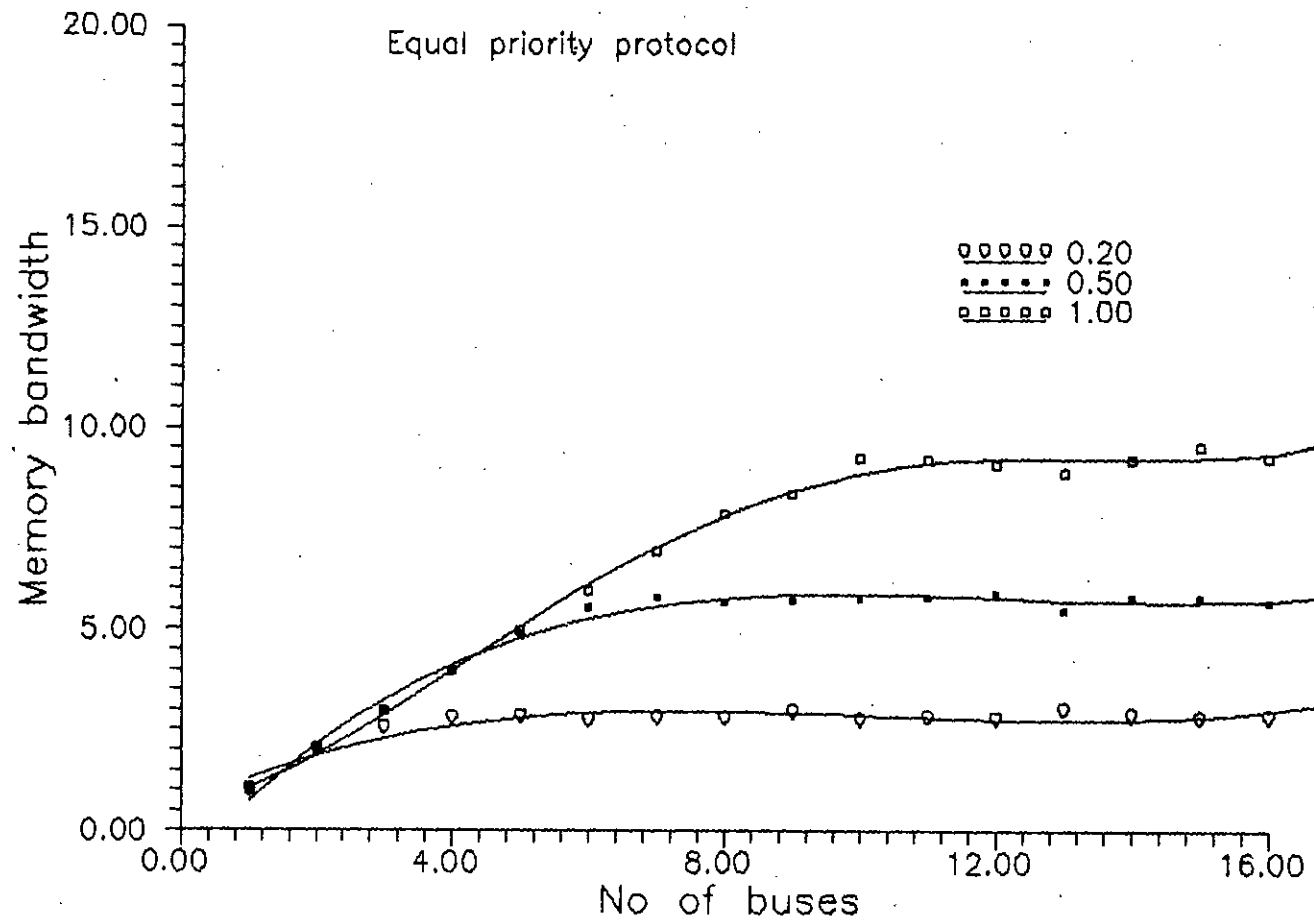


Figure 5.8 Memory bandwidth vs number of buses for circuit switched synchronous system.

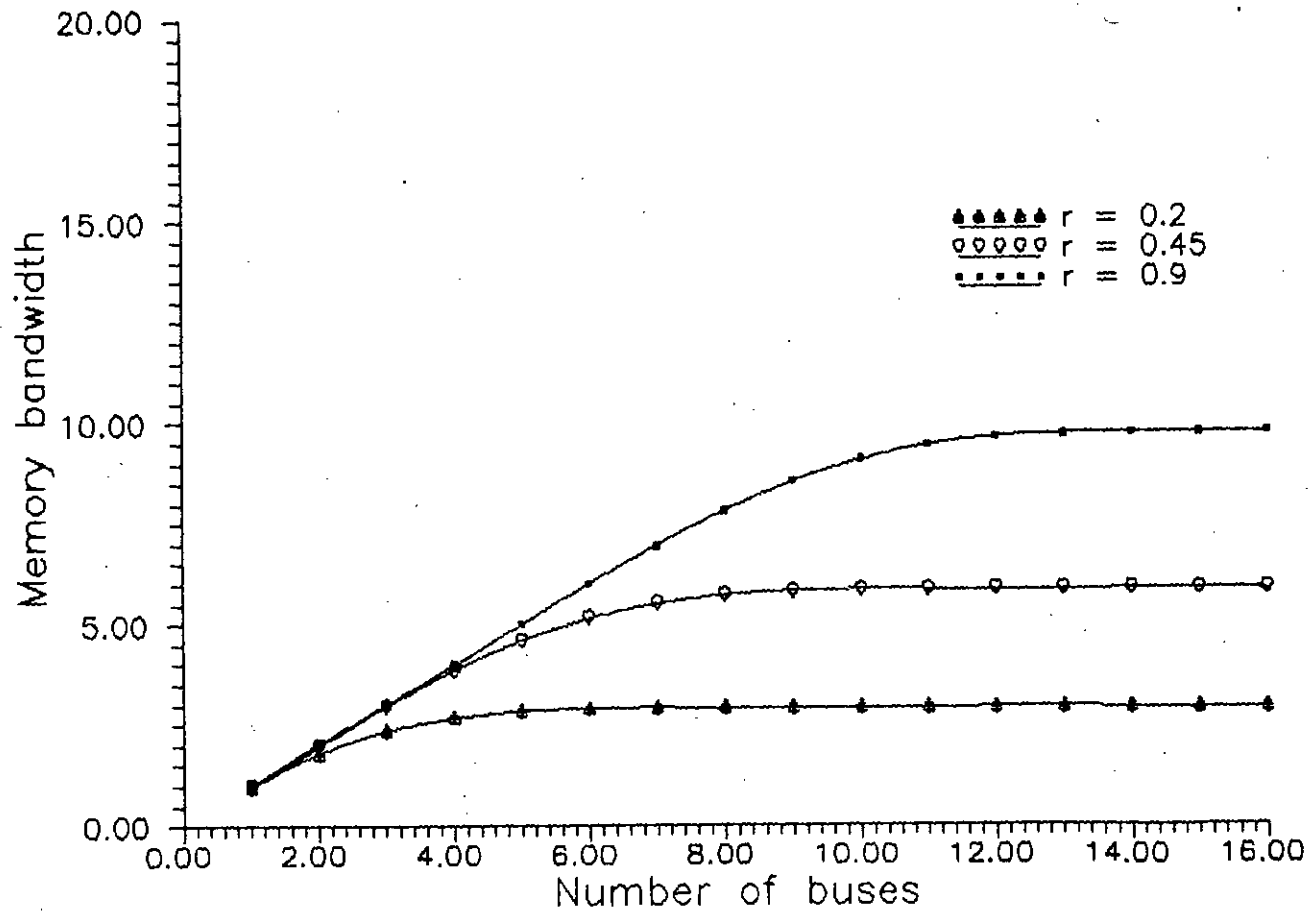


Figure 5.9: Memory bandwidth vs number of buses (By analysis).

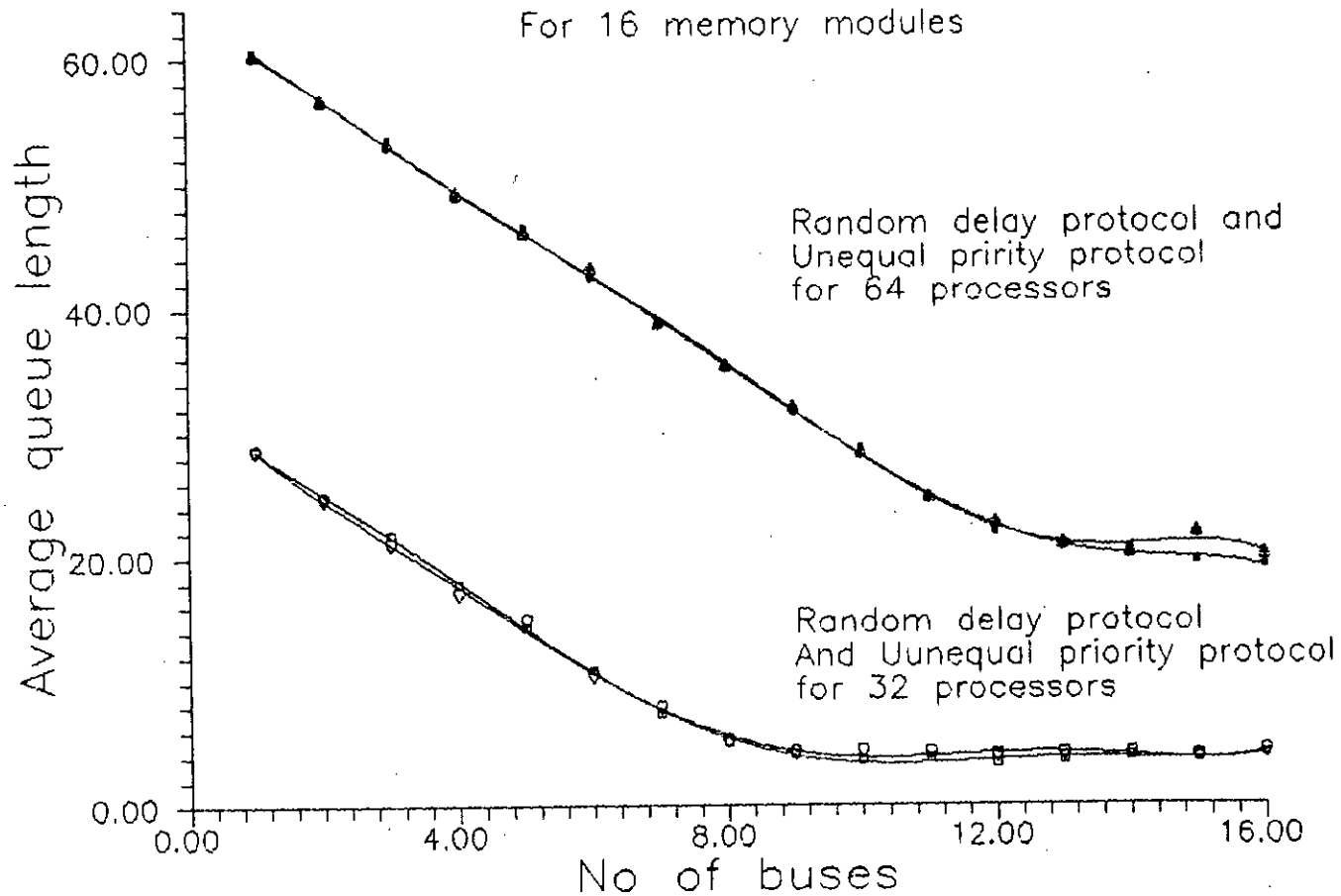


Figure 5.10.1: Average queue length vs number of buses for asynchronous circuit switched system.

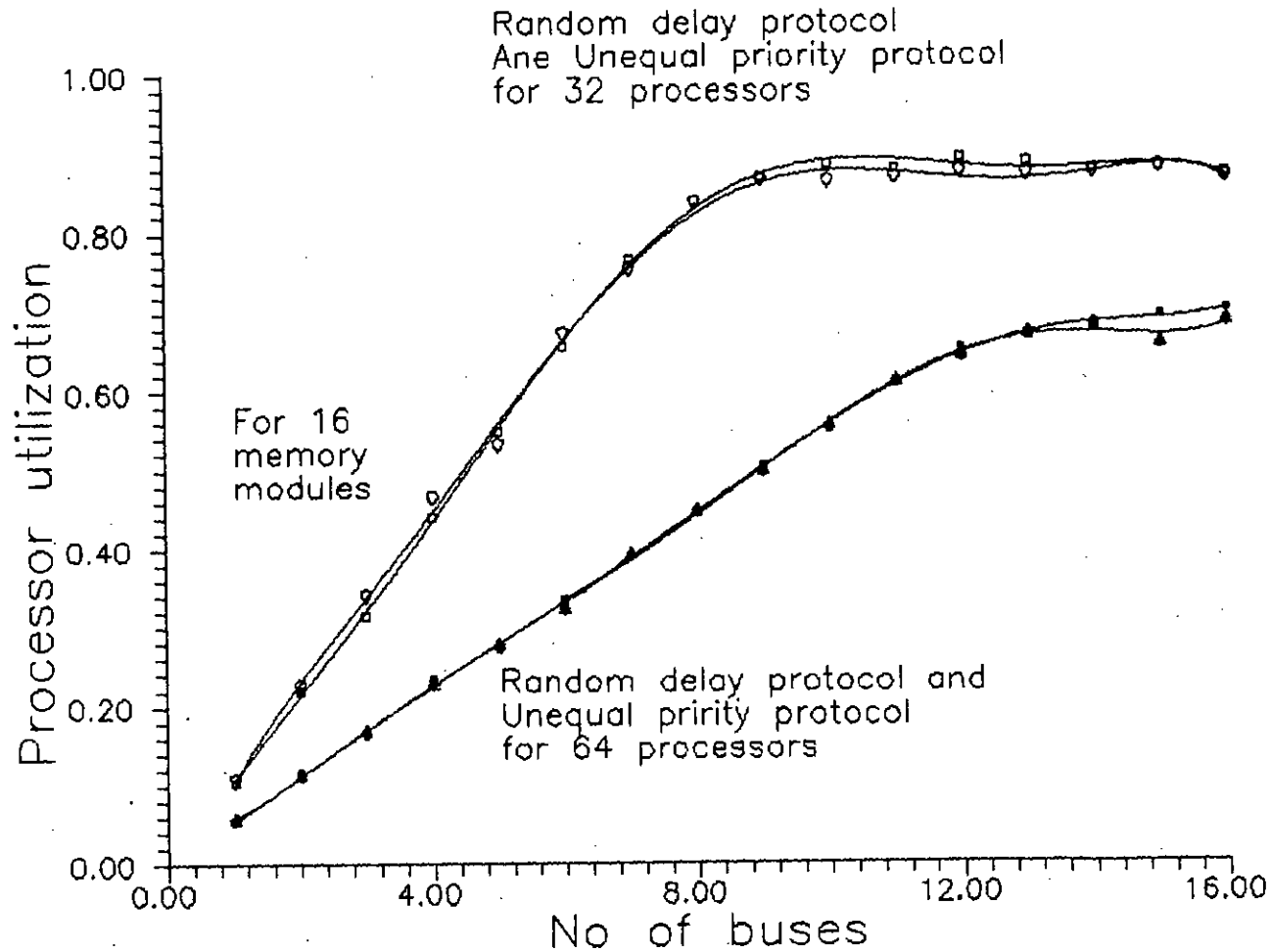


Figure 5.10.2: Processor utilization vs number of buses for asynchronous circuit switched system.

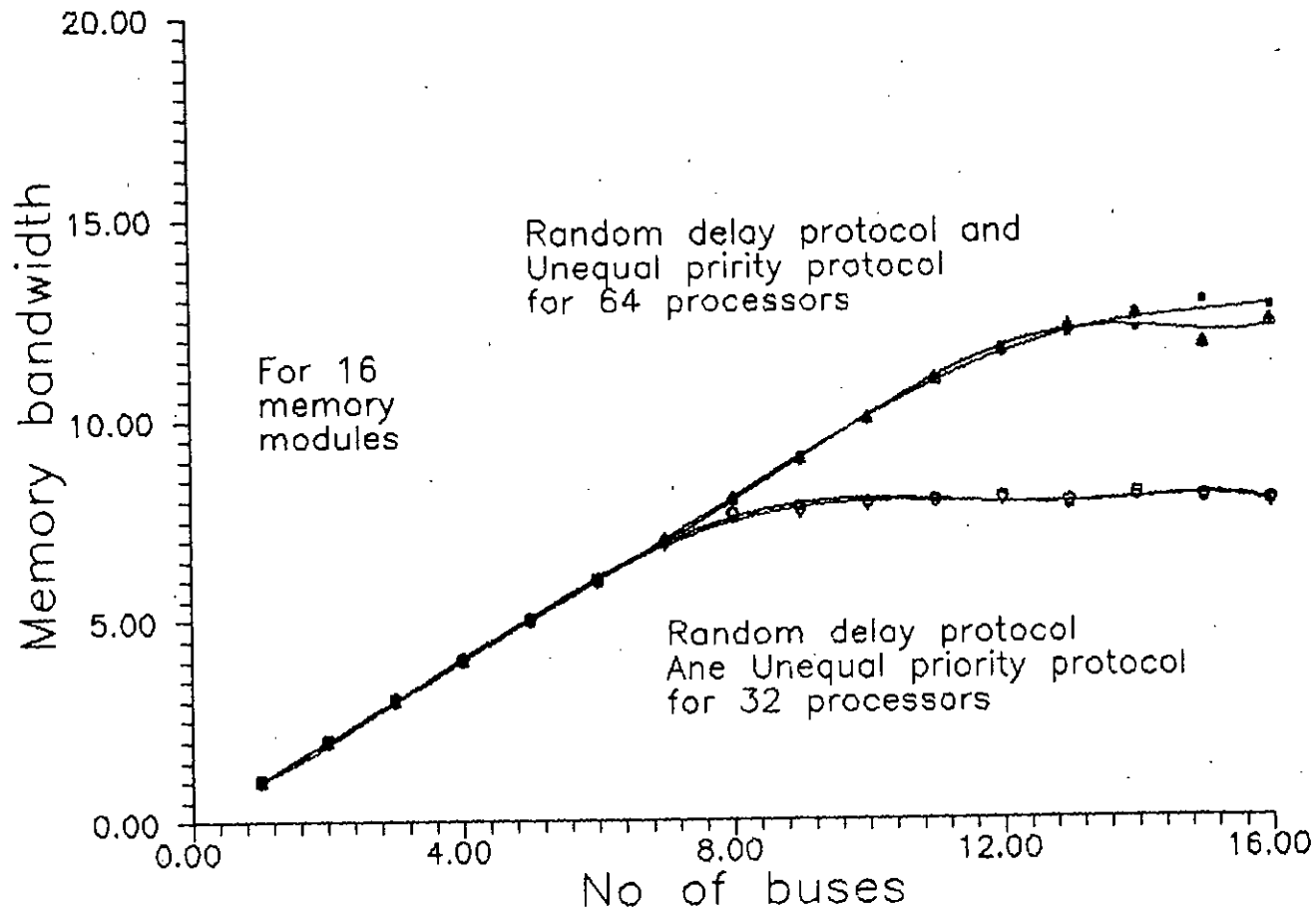


Figure 5.10.3: Memory bandwidth vs number of buses for asynchronous circuit switched system.

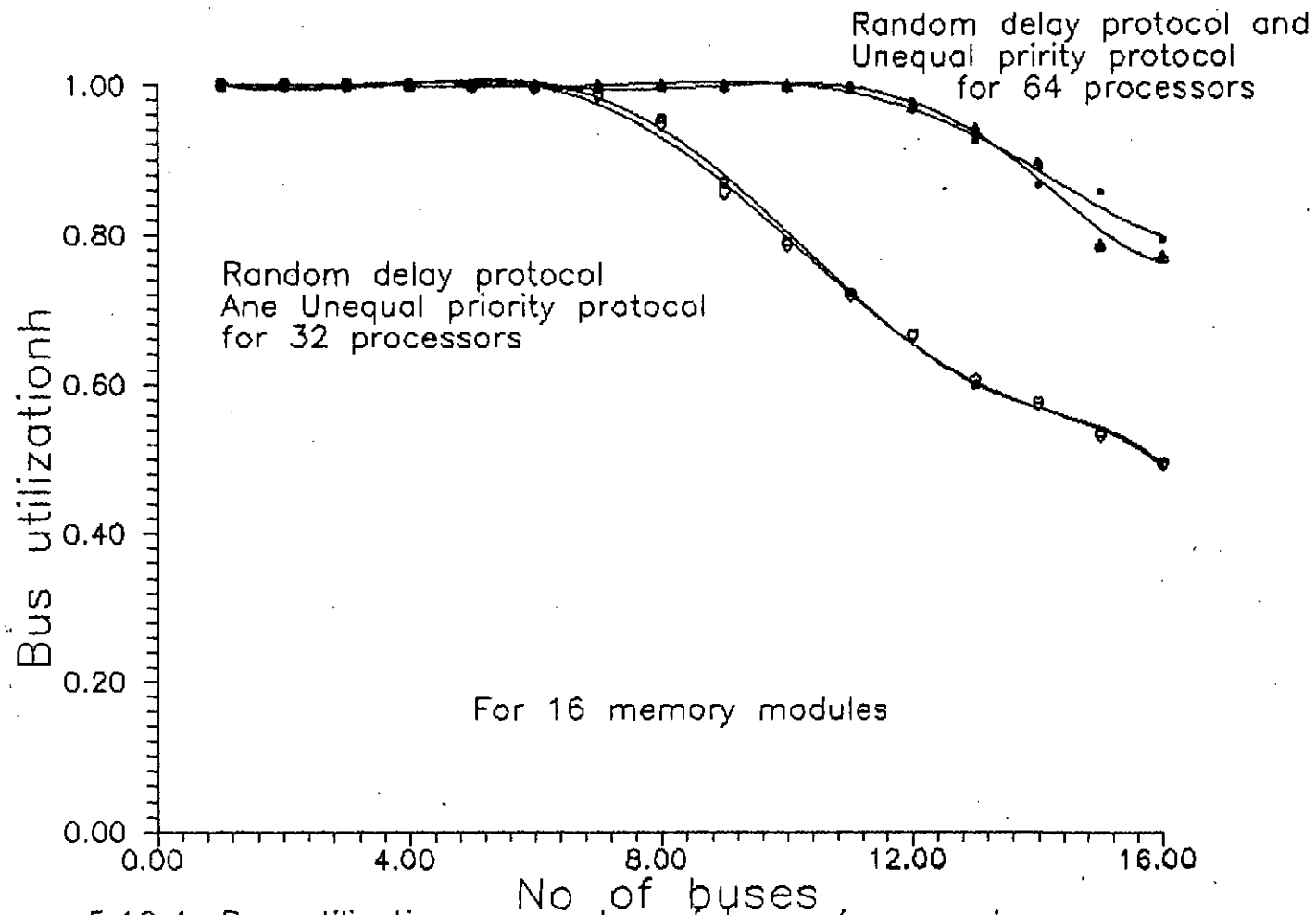


Figure 5.10.4: Bus utilization vs number of buses for asynchronous circuit switched system.



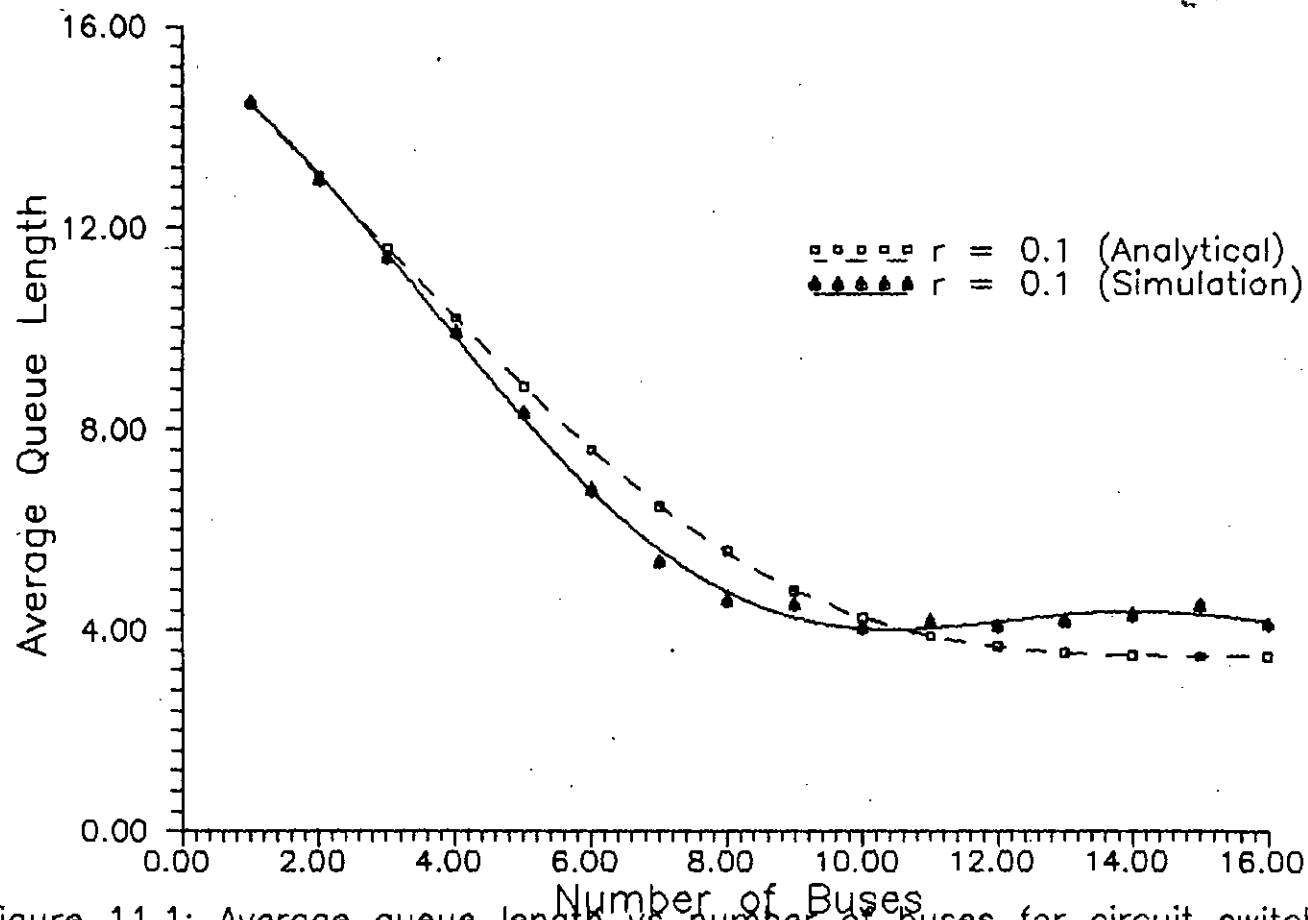


Figure 11.1: Average queue length vs number of buses for circuit switched synchronous system.

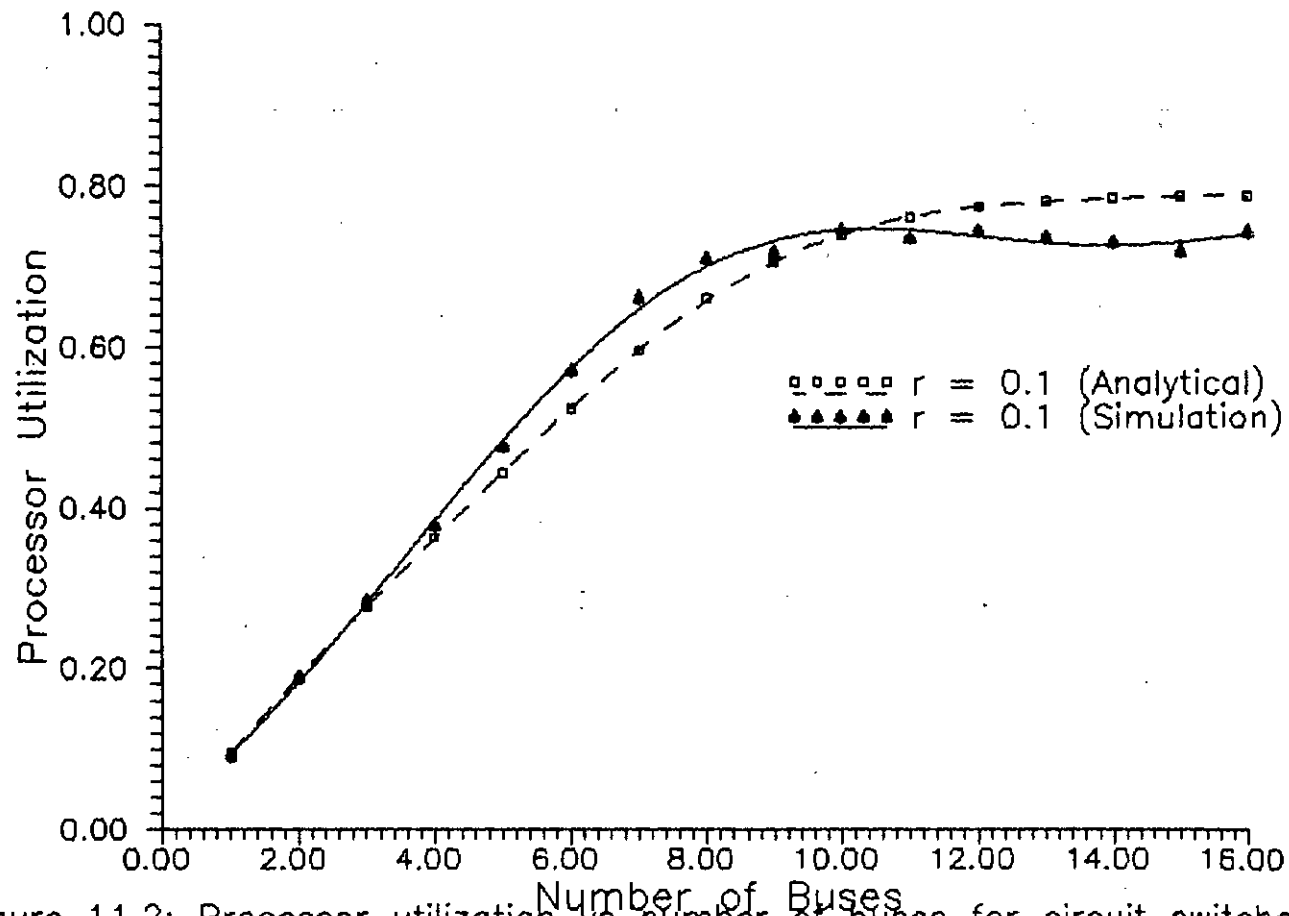


Figure 11.2: Processor utilization vs number of buses for circuit switched synchronous system.

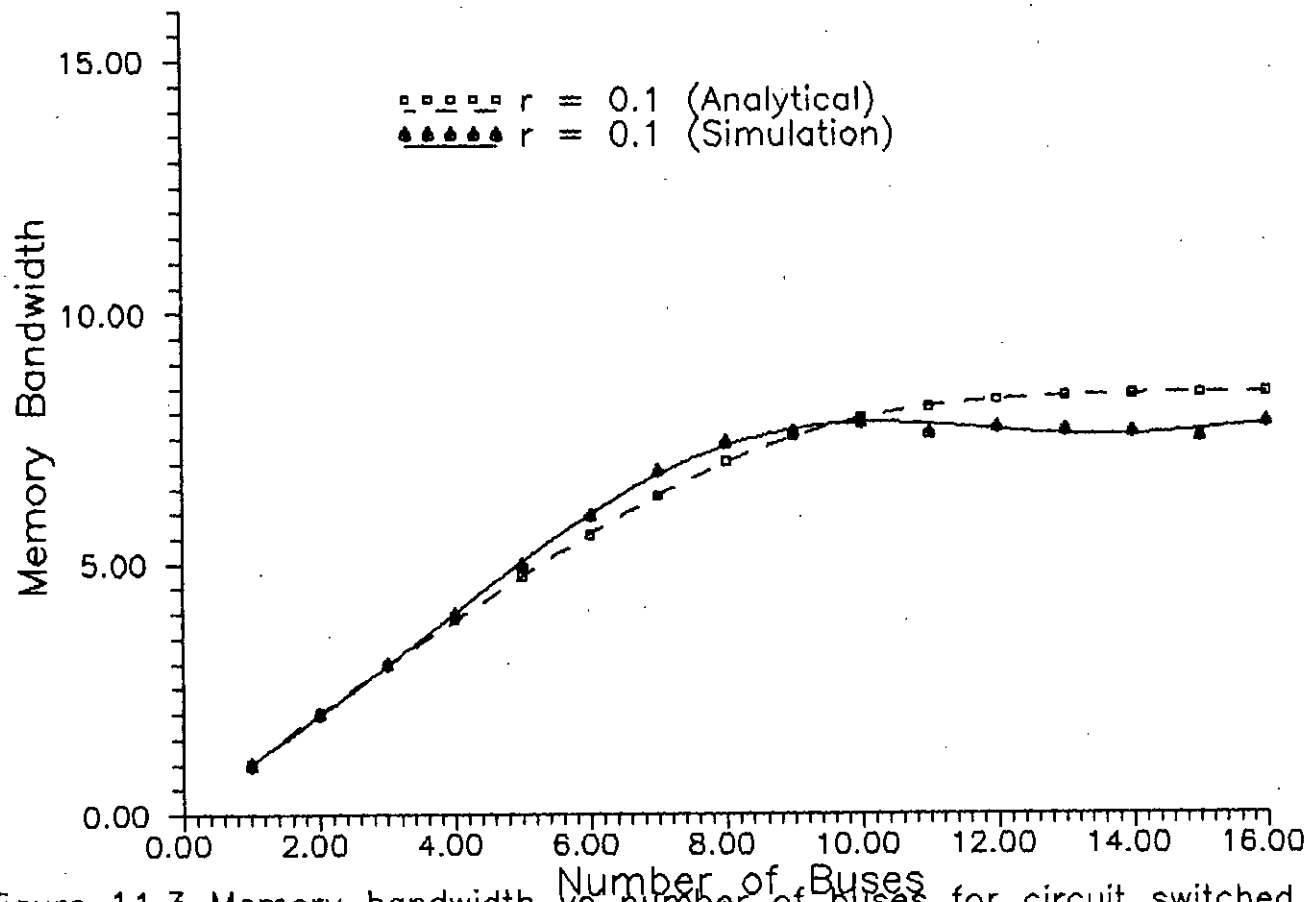


Figure 11.3 Memory bandwidth vs number of buses for circuit switched synchronous system.

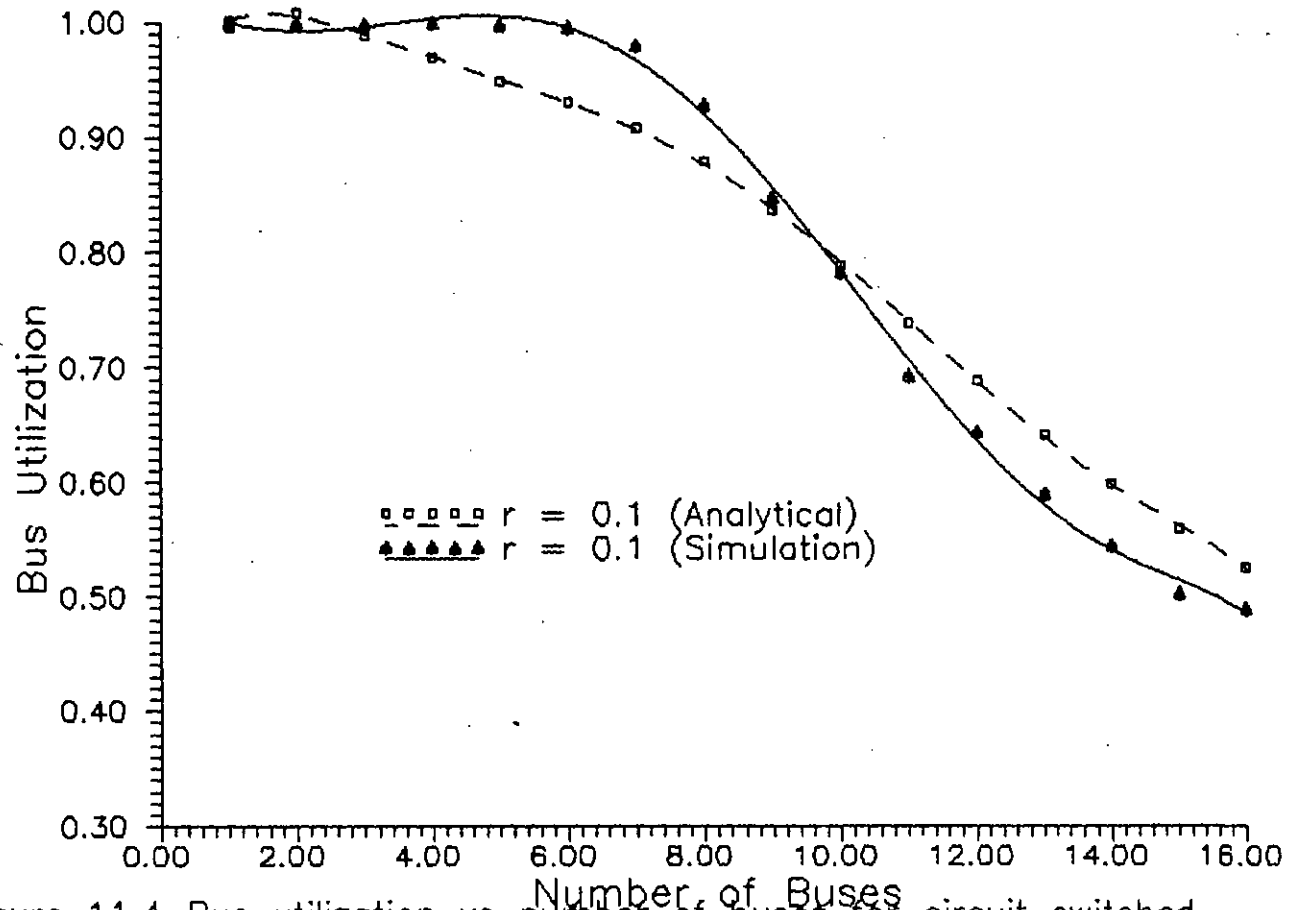


Figure 11.4 Bus utilization vs number of buses for circuit switched synchronous system.

when more than one processor request the same memory module simultaneously. If the interconnection network is multiple bus connection, then it is expected that as the number of buses increases performances will improve upto certain level. Because, when there is only one available bus then only one of the requesting processors may win in bus arbitration. Performance degradation occurs because the memory modules requested by the processors, may remain unoccupied, as they have failed in bus arbitration. That is there remain free memory modules because of unavailability of a bus. So average queue length become high, because processors have to wait for availability of both bus and memory module. As the number of buses are increased from one to two, bus conflict reduces. Average queue length decreases upto certain number of buses and then a saturation value is attained. After that value the increase of number of buses does not decrease average queue length. This is shown in fig 5.1.1, where it is seen that the saturation bus numbers are 9, 10 and 5 for processor think times of 2.5, 5, 50 slots respectively. It is possible to find out an optimum number of buses, because there is an average number of processors which generate request simultaneously in a specific system depending on average processing or think time of the processors of the system. And the optimum number of buses is a function of average number of processors and total number of memory modules. If think time is large then average number of processors which can generate request simultaneously decreases, that is, bus conflict decreases. So less number of buses can bring the system into

saturation region as in figure 5.1.1. There it is seen that when processor think time is 50 slots then only 5 buses can bring the system to crossbar performance level.

Also it is expected that processor utilization will increase with increase of number of buses [6]. Because then less number of processors have to wait in queue. But this increase in utilization also become flat after an optimum number of buses where bus conflict is almost absent. This is shown in processor utilization vs. number of buses curve in different graphs.

Also for memory bandwidth the same effect is seen, that is, memory bandwidth increases upto certain number of buses. This situation is shown in different graphs (Figure no. 5.1.3, 5.2.3) where memory bandwidth is plotted against number of buses. For bus utilization it is expected [6] that when there is only one bus then request rate is higher, so this bus is always busy. hence bus utilization remains almost 100 percent. When the number of buses increases bus utilization remains flat in 100 percent utilization upto certain number of buses. Then, after optimum number of buses bus utilization decreases slightly. Looking at figures 5.3.1 to 5.3.4 for probability of cache miss = 0.1, it is seen that optimum number of buses is 8. And at this value bus utilization becomes around 0.97 which is slightly less than 100 percent efficiency. After then if the number of buses is increased bus utilization gradually decreases and at 16 buses i.e. when there is physically total connection between processor and memory modules, it is seen that bus utilization is very low. In figure 5.3.4 it is 0.48 and in figure 5.1.4 it is 0.57. So it is seen

that in these cases approximately 50 percent of the buses are unutilized.

In the multiprocessor system it is possible to remove bus conflict totally. The simulation is carried out for finding the number of buses when there is no bus conflict. Average queue length, processor utilization, memory bandwidth and bus utilization are plotted against the number of buses, because for a system all these parameters are needed for determining the system performance [6]. For example, a circuit switched synchronous system with probability of cache miss of 0.2, it is required that memory bandwidth will be in the range of 5-6. Then from figure 5.3.3, for equal priority, it is found that the memory bandwidth of 5 occurs at number of buses equal to 5. So, these graphs can be used in system design. Bus requirements can be kept within the limit of cost by fixing other performance requirements. Figure 5.8 and figure 5.9 are for synchronous circuit switched system, where equal priority is considered. In figure 5.8 simulation is carried out for finding memory bandwidth as number of buses is varied with the probability of request at the beginning of system cycle as the parameter. In figure 5.9 the curves of figure 5.8 are validated by probabilistic analysis method. In figure 5.11.1 through 5.11.4 synchronous circuit switched system is validated by semi-Markov analytical method. In these figures, the probability generating a request or cache miss is considered to be 0.1. From both probabilistic and semi-Markov analysis it is seen that simulation results differ by 5-7 percent from the analytical solutions.

From figures 5.10.1 through 5.10.4 it is seen that for large number of processors random delay protocol and unequal priority protocol give the same performances. It may be said that random delay protocol is in some cases slightly better. From figure 5.10.1 for 32 processor the average queue length curve of random delay protocol in flat region is a little below than that of unequal priority protocol.

Study of figures 5.1.1 through 5.1.4 and figures 5.2.1 through 5.2.4 show that unequal priority protocol is better for lower processor think times, such as for  $T = 5$  or  $T = 2.5$ . But for larger think time random delay protocol is as good as unequal priority protocol.

Random delay protocol works well in some cases. When there are larger number of processors (32 or 64) request to a particular memory module is large compared to lower number of processors, say 16 processors. In the simulation with large number of processors (32 or 64) lower processing time is considered. So request rate is very high. For very high think time request rate is very low. So it can be concluded that random delay protocol works as good as unequal priority protocol for very low and very high request rate. For intermediate think time unequal priority protocol is better. The cause that random delay protocol works good for some region is that resubmission of request after random delay decreases the probability of collision among processors submitting requests simultaneously.

Random delay protocol is better than unequal priority protocol because it is closer to equal priority



protocol. With respect to hardware it is easier to implement [10]. When there is bus or memory conflict resubmission of request is required, which causes this protocol to become closer to equal priority protocol.

From figure 5.7.1 to figure 5.7.4 requirement of bus for optimum performance is very high for large number of processors. But from these figure it is possible to take a less number of buses while keeping performances in acceptable region. For example, if acceptable memory bandwidth for 64 processor and 16 memory module system is 8 then from figure 5.7.3 the number of buses needed is 9.

Performances in asynchronous system are better than those of synchronous system. Because in asynchronous system requests can be generated at any moment and so if resources are available processors do not have to wait for the next clock. But In synchronous system request can be generated only in the presence of clock. So a processor has to wait for a positive clock edge. This situation can be examined by comparing the graphs (figures 5.1.1-5.1.4 and 5.3.1-5.3.4) of synchronous and asynchronous systems.

Packet switched synchronous and asynchronous system are also studied. Figures 5.5.1 through 5.5.4 are for various performances of packet switched synchronous system. And figures 5.6.1 through 5.6.4 are for packet switched asynchronous system. In both synchronous and asynchronous systems equal priority protocol is considered. Because in a multiprocessor system if it is required that all the processors have equal priority then

this protocol is desirable.

From figure 5.5.1 through 5.5.4 for packet switched synchronous system, it is seen that probabilities of cache miss(or probability that a processor ends processing for non-cache system) of 0.1, 0.2 and 1.0 are used. From these figures it is seen that all the curves are closely spaced, which implies that there are little performance differences for the given probabilities of cache misses. But for circuit switched system(figure 5.4.1-5.4.4) these differences are not as small as in packet switched system. To see the effect of probability of cache miss more clearly (figures 5.6.1 through 5.6.4) four different cache misses namely 0.1, 0.2, 0.5, 1.0 are taken for packet switched asynchronous system. Here also differences in the performances are not so prominent as compared to the circuit switched system.

From figures 5.5.1 to 5.5.3 and figures 5.6.1 to 5.6.3, i.e. for packet switched synchronous and asynchronous system it is seen that absence of bus conflicts occurs at 4 buses. The performances attain saturation value at 4 buses and then remains flat. And from figures 5.5.4 and 5.6.4 it is seen that after 4 buses bus utilization gradually decreases and reaches around 0.2 at 16 buses (total connection).

In packet switched system when a processor submits requests, it first checks for a idle bus. If there is any idle bus and it wins in bus arbitration it occupies the bus. And when transfer of request through bus is complete then IMP\controller [5] tracks the requested memory status. If that memory module is

free and the processor wins in memory arbitration then memory operation is performed. So here it is not required for a requesting processor that both the requested memory and any of the buses to be free simultaneously. So there is less conflict in packet switched system and performances attain optimum value with less number of buses(4-5) .

Hardware design of arbiter of asynchronous circuit switched system is shown in figures 2.1 through 2.3 . If there are 8 processors, 8 memory modules and 4 buses, then total number of gates required for this arbiter is 1576 and the delay is 23d. If 16 processors are used then total number of gates required for this arbiter is 7184 and delay becomes 29d. Design of 2 to 1 arbiter of synchronous circuit switched system is shown given in [12]. Number of gates required for 8 to 1 memory arbiter is 98 and delay is 6d. And number of gates required for 16 to 1 memory arbiter is 210 with delay 8d. An improved design of 8 to 4 arbiter for bus arbitration is shown in figure 2.6 . The 8 to 4 arbiter in [12] requires 4 level logic. Total number of gates required are 248 and delay is 11d. In the design given in section 2.6, a lookahead approach is used and delay is reduced to 5d, but total number of gates required is 304. In [12] it is said that if this 8 to 4 arbiter is implemented using 3 level logic then delay can be reduced to 5d with only 268 gates. But in that case maximum fan-in of some gates becomes 8 whereas in the design presented here maximum fan-in is only 3.

## CHAPTER 6

### CONCLUSIONS AND SUGGESTIONS:

#### 6.1 Conclusion:

In the multiprocessor systems studied, it is seen that when there is only one time shared bus performance parameters are not so high. If number of buses is increased performance increases upto certain number of buses. Crossbar performance can be achieved with lower number of buses. For fast operation it is possible to increase number of processors to an optimum value. For this an adequate number of buses are also necessary. Performances observed are relatively better in asynchronous system, because conflicts are comparatively lower. But asynchronous system design is complex and its analytical model development requires a great deal of computations. Whereas synchronous system design is less complex and an equal priority assignment on all processors is easily possible with the aid of flip flops.

In packet switched system performances reach saturation values with only 3 to 5 buses (for 16 processor and 16 memory module) depending on cache misses. Whereas in circuit switched system saturation values are reached at 5 to 10 buses depending on cache misses. In packet switched system extra overhead required is intermediate message processor(IMP)/controller. In asynchronous circuit switched system with 16 processors and 16 memory modules, unequal priority protocol is better than random delay protocol when processing time is short. But for larger processing time (50 slot) random delay protocol is

as good as unequal priority protocol. With 16 memory modules, if number of processors are increased (32,64) then it is seen that random delay and equal priority protocol has almost equal performances. Random delay protocol is easy to implement and it introduces almost equal priority to processors [12]. In synchronous circuit switched system equal and unequal priority protocol is used. In unequal priority protocol lower priority processors have to wait in queue a longer time which is not desirable but this does not affect overall performances. Multiple bus system is fault tolerant and cost effective than MINs. Simulation and analytical results of the multiprocessor system differed by 5-6 percent only. Various simulation results could be helpful in actual system design. With specific load condition and depending on synchronous or asynchronous system and packet switched or circuit switched system, one can easily select the best number of buses for the system.

Delay in arbitration in 8 to 4 synchronous arbiter can be reduced as explained in arbiter design to  $5d$  ( $d$  is nominal gate delay). Similarly 4 to 2 arbiter design is also presented and from these other combinations can be made. Total number of gates required for designing arbiter of an asynchronous system with 8 processors and 4 buses is 1576 and it results in a delay of  $23d$ . Whereas for synchronous system with equal number of processors and buses total number of gates required is 1088 with delay  $11d$ .

## 6.2 Suggestions for Further Research:

1. Performance study of multi-bus multiprocessor system considering different cache protocols can be pursued as a continuation of this thesis work. The extent to which references have to be made to shared memory depends a lot on the cache algorithm used. A key aspect of this algorithm is the policy for updating the shared memory when a request is addressed to the cache. Copy back divides the references to shared memory into two categories: those due to cache misses and those due to the copy back process. A cache miss occurs when a processor generates a read or write request for data or instructions that cannot be satisfied by the cache.

2. Analytical modeling of multiple bus multiprocessor system for those connections not covered in chapter 3 can be pursued. It is possible to use semi-Markov model, product form solution or closed queuing model for analysis. Combinatorial analysis approach can also be taken. For analysis it is possible to use packet switched multiple bus multiprocessor system. Also in chapter 3 a simplified model of semi-Markov process is presented. It is required that this model is justified by analysis and simulation.

3. Performance analysis of multistage interconnection network can be studied. Multistage interconnection networks are cost effective. But in MIN there is only one path from a processor to a memory module, i.e. it is not fault tolerant. So it is necessary to make them fault tolerant. For this it would be useful to study Omega Network or Delta

Network.

4. The study of synchronization and program partitioning methods in multiprocessor system would be also interesting. In order to use multiprocessor effectively program should be partitioned so that all portions can be processed in different processors. Processes must be able to communicate and to synchronize with each other. For synchronization bit map method can be used [1]. Communication among processors can be better done by message passing method.

```

/*Simulation program for asynchronous circuit switched system
with random delay protocol */
#include <stdio.h>
#include <math.h>
#include <alloc.h>
#include <time.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
/* Different structures and globals are defined */
#define no 16
#define mo 16

typedef struct processor{
    unsigned n_e;
    unsigned m_no;
    unsigned b_no;
    unsigned time;
    }pro;

typedef struct buses {
    unsigned state;
    unsigned p_no;
    }bus;

typedef struct memory{
    unsigned state;
    unsigned p_no;
    }mem;

typedef struct controller {
    unsigned time;
    unsigned p_no;
    struct controller *next;
    }con;

unsigned ko,*q,*m1,*n1,*k1;
unsigned *ttot,rcount,t1,t2,q_last,n1_last,m1_last,k1_last;

/* Subroutine end of processing has described below. A processor comes
in this routine when it ends processing or there is a cache miss */

eop(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned *e_count)
{
/* *s1 is number of current processor. */
/* *t is current time. */
/* start is first element in the event queue */
/* *e_count is the number of simultaneous request */
    unsigned i,j,del;

j=random(mo); /* Select with equal priority any memory module */
/* Assuming equal memory reference probability */

```



```

/* If the required memory is busy then processor waits for next
   transition */
if(m[j].state ==1 )
{
    del=random(4)+1;
    p[*s1].time+=del;
    p[*s1].n_e=2;
    p[*s1].m_no=j;
    *q+=1;*n1-=1;
    printf("%d    %d    %d    %d    %d    -    -    -\n", *t,*q,*n1,*m1,*k1

    *t+=del;          /* Update the time */
    insert(s1,t,start); /* Insert the processor in event queue */
}
/* Also if there is no bus free then processor waits for next
   transition */
else
{
    for(i=0;i< ko;i++)
    {
        /* If there is any bus free */
        if(b[i].state == 0)
            break;
    }

if(i== ko)          /* There is no free bus */
{
    del=random(4)+1;
    *q+=1;*n1-=1;
    p[*s1].time+=del;
    p[*s1].m_no=j;
    p[*s1].n_e=2;
    printf("%d    %d    %d    %d    %d    -    -    -\n", *t,*q,*n1,*m1,*k1)

    *t+=del;
    insert(s1,t,start);
}
else          /* The processor has won in both arbitration */
{
    b[i].state=1;          /* Make the occupied bus busy */
    m[j].state=1;          /* Make the requested memory module busy */
    b[i].p_no = *s1;
    m[j].p_no = *s1;
    p[*s1].m_no=j;
    p[*s1].b_no=i;
    p[*s1].n_e=1;
    /* Make the processor busy in memory access */

    del=20;
    p[*s1].time+=del;
    *m1+=1;
    *k1+=1;
    printf("%d    %d    %d    %d    %d    %d    %d    %d \n",*t,*q,*n1,*m1,*k1,
    *s1,p[*s1].m_no,p[*s1].b_no);
}
}

```

```

    *m1+=1;
    *k1+=1;
    printf("%d    %d    %d    %d    %d    %d    %d %d\n",
           *t,*q,*n1,*m1,*k1,
           *s1,p[*s1].m_no,p[*s1].b_no);

    *t+=del;
    insert(s1,t,start);
  }
}

/* Subroutine processor releases memory is described below */

prm(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned *e_count)
{
  unsigned del;
  del=random(101)+1;          /* Return the processor to
  processing state */
  p[*s1].time+=del;
  p[*s1].n_e=0;
  b[p[*s1].b_no].state=0;    /* Make the occupied bus */
  m[p[*s1].m_no].state=0;    /* and memory module free */
  b[p[*s1].b_no].p_no=0;
  m[p[*s1].m_no].p_no=0;

  *k1-=1;*m1-=1;           /* Update number of busy bus */
                           /* busy memory module and busy and */
                           /* waited processors */
  printf("%d    %d    %d    %d    %d    %d    %d %d\n",
         *t,*q,*n1,*m1,*k1, *s1,p[*s1].m_no,p[*s1].b_no);

  *t+=del;
  insert(s1,t,start);
}
/* Subroutine end of next transition is described below. A
processor comes here when it fails in arbitration and
resubmits request after waiting arandom amount of time */

ent(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned *e_count)
{
  unsigned i,del;
  for(i=0;i< ko;i++)
  {
    if(b[i].state==0)
      break;
  }
  if(i == ko)
  {
    del=random(4)+1;
    p[*s1].time+=del;
    p[*s1].n_e=2;
  }
}

```

```

printf("%d      %d      %d      %d      %d      -      -      -\n",
        *t,*q,*n1,*m1,*k1);
*t+=del;
insert(s1,t,start);
}
else
{
    if(m[p[*s1].m_no].state==1)
    {
        del=random(4)+1;
        p[*s1].time+=del;
        p[*s1].n_e=2;
        printf("%d      %d      %d      %d      %d      -      -      -\n",
                *t,*q,*n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        *q-=1;

        del=20;
        p[*s1].time+=del;
        b[i].state=1;
        p[*s1].n_e=1;
        p[*s1].b_no=i;
        m[p[*s1].m_no].state=1;
        b[i].p_no = *s1;
        m[p[*s1].m_no].p_no = *s1;
        *n1+=1;*m1+=1;*k1+=1;
        printf("%d      %d      %d      %d      %d      %d %d %d\n",
                *t,*q,*n1,*m1,*k1,*s1,p[*s1].m_no,p[*s1].b_no);
        *t+=del;
        insert(s1,t,start);
    }
}
}

/* subroutine insert is stated below. After finish of an
event a processor is to be inserted in proper position
of the event queue */
insert(unsigned *s1,unsigned *t,con *start)
{
FILE *f2;
con *prev,*new,*current,*inter;
current=start->next;
if(*t < current->time)
{
start->p_no = *s1;
start->time = *t;
}
else
{
if((new=(con*)calloc(1,sizeof(con)))==NULL)
{
printf("No memory available for allocation \n");
}
}
}

```

```

        exit(1);
    }
    start->p_no=current->p_no;
    start->time=current->time;
    start->next=current->next;
    free(current);
    current=start->next;
    prev=start;
    while(current->next != NULL && *t >= current->time)
    {
        prev=current;
        inter=current->next;
        current=inter;
    }
    new->p_no = *s1;
    new->time = *t;
    if(current->next != NULL)
    {
        prev->next=new;
        new->next=current;
    }
    else
    {
        if(*t >= current->time)
        {
            current->next=new;
            new->next=NULL;
        }
        else
        {
            prev->next=new;
            new->next=current;
        }
    }
}
current=start;
f2=fopen("file2.dat","a+");
while(current)
{
    fprintf(f2,"          %u          %u\n" ,current->time,
            current->p_no);
    current=current->next;
}
fprintf(f2," END OF SERVICE \n");
fclose(f2);
}
/* subroutine for random number generation */
g_rand()
{
    double r,log();
    unsigned x;

    r=(double)(random(1000)/10000);
    r-=1;
    x=-log(-r)/log(.49);
}

```

```

    return(x);
    }

/* This subroutine is for calculating statistics of simulation
*/

result(unsigned *tq,unsigned *tn1,unsigned *tm1,unsigned
*tk1,unsigned *t)
{
    unsigned sub;
    if(rcount == 0)
    {
        t1=*t;
        q_last = *q;
        n1_last = *n1;
        m1_last = *m1;
        k1_last = *k1;
        rcount+=1;
    }
    else
    {
        t2 = *t;
        sub = t2-t1;
        *tq+= q_last*sub;
        *tn1+= n1_last*sub;
        *tm1+= m1_last*sub;
        *tk1+= k1_last*sub;

        t1=t2;
        q_last = *q;
        n1_last = *n1;
        m1_last = *m1;
        k1_last = *k1;
    }
}

/* Main program is started below */
main()
{
    FILE *f1;
    con *start,*prev,*new,*current;
    pro p[no];
    mem m[mo];
    bus b[16];

    unsigned *tq,*tn1,*tm1,*tk1;
    unsigned i,j,*s_c,*e_count,r[16],*s1,count,scr_cnt;
    unsigned *t;
    float ttq,ttn1,ttm1,ttk1;
    void *calloc();
    void *malloc();
    ko=1;

    clrscr();
    f1=fopen("file1.dat","a+");
    for(ko = 1;ko < 17;ko++)

```

```

{
rcount=0; t1=0; t2=0;
q_last=0; n1_last=0; m1_last=0; k1_last=0;
ttot=(unsigned*)malloc(sizeof(unsigned));
n1=(unsigned*)malloc(sizeof(unsigned));
m1=(unsigned*)malloc(sizeof(unsigned));
k1=(unsigned*)malloc(sizeof(unsigned));
q=(unsigned*)malloc(sizeof(unsigned));
t=(unsigned*)malloc(sizeof(unsigned));
s1=(unsigned*)malloc(sizeof(unsigned));
tq=(unsigned*)malloc(sizeof(unsigned));
tn1=(unsigned*)malloc(sizeof(unsigned));
tm1=(unsigned*)malloc(sizeof(unsigned));
tk1=(unsigned*)malloc(sizeof(unsigned));
e_count=(unsigned*)malloc(sizeof(unsigned));
s_c=(unsigned*)malloc(sizeof(unsigned));
*n1=18; *m1=0; *k1=0, *q=0; *ttot=3000;
*tq=0; *tn1=0; *tm1=0; *tk1=0;

if((start=( con*)calloc(1,sizeof( con)))==NULL)
{
printf("No memory available for allocation \n");
exit(1);
}

/* Random numbers are generated below */
randomize();
for(i=0;i < no;i++)
{
r[i]=random(101)+1;
}
/* Event queue is formed below */

start->next=NULL;
start->p_no=0;
start->time=r[0];
for(i=1;i<no;i++)
{
if((current=( con*)calloc(1,sizeof( con)))==NULL)
{
printf("No memory");
exit(1);
}
current->time=r[i];
current->p_no=i;
if(current->time < start->time)
{
prev=start;
current->next=prev;
start=current;
}
else
{
prev=start;
new=start->next;
}
}

```

```

while(new != NULL && current->time >= prev->time)
{
    if(current->time < new->time)
    {
        current->next=new;
        prev->next=current;
        prev=new;
    }
    else
    {
        prev=new;
        new=new->next;
    }
}
if(new == NULL)
{
    prev->next=current;
    current->next=NULL;
}
}
}
f1=fopen("file1.dat","a+");
current=start;
while(current)
{
    fprintf(f1,"          %u          %u\n" ,current->time,
            current->p_no);
    current=current->next;
}
fclose(f1);
/*initialization of arrays are done below */
for(i=0;i<no;i++)
{
    p[i].n_e=0;
    p[i].m_no=0;
    p[i].b_no=0;
    p[i].time=r[i]; /* A random starting time is for each
                    processor */
}
for(i=0;i<mo;i++)
{
    m[i].state=0;
    m[i].p_no=0;
}
for(i=0;i<ko;i++)
{
    b[i].state=0;
    b[i].p_no=0;
}
printf(" Time          P          P          B          M          P          M B\n");
printf("   Waiting      Busy          busy  Busy  no  no  no \n");

gotoxy(2,24);
printf("PRESS ANY KEY TO SEE NEXT PAGE");
/*Finding of current event\events are started below */

```

```

window(1,3,79,20);
scr_cnt=0;
clrscr();
gotoxy(1,1);
*t = start->time;
while(*t < *ttot)
{
*e_count=1;
current=start;
new=current->next;
while(current->time == new->time)
{
*e_count+=1;
current=new;
new=current->next;
}
/* Look the number of processors which submit request
simultaneously and check the condition for give them service
*/

while(*e_count)
{
*e_count-=1;
*s1=start->p_no;
*s_c=p[*s1].n_e;
switch(*s_c)
{
case 0:{
eop(s1,t,start,p,m,b,e_count);
break;
}
case 1:{
prm(s1,t,start,p,m,b,e_count);
break;
}
case 2:{
ent(s1,t,start,p,m,b,e_count);
break;
}
}
}/*switch*/
*t = start->time;
if(*e_count==0 && *t>1000)
result(tq,tn1,tm1,tk1,t);
scr_cnt+=1;
if(scr_cnt >=20)
{
getch();
clrscr();
scr_cnt=0;
}
}
}

getch();

```



```

window(1,1,79,24);
clrscr();
printf("%6d %6d %6d %6d \n",*tq,*tn1,*tml,*tk1);
*ttot-=1000;
ttq=((float)(*tq))/((float)(*ttot));
ttn1=((float)(*tn1))/((float)(*ttot));
ttml=((float)(*tml))/((float)(*ttot));
ttkl=((float)(*tk1))/((float)(*ttot));
printf("\n\tAverage Queue Length=%f\n",ttq );
printf("\n\tAverage No of busy Processor =%f\n",ttn1);
printf("\n\tAverage No of Busy Memory =%f\n",ttml);
printf("\n\tAverage No of Busy Bus =%f\n",ttkl);
printf("\n\tProcessor Utilization =%f\n",ttn1/no);
printf("\n\tMemory Bandwith=%f\n",ttml/mo);
printf("\n\tBus Utilization=%f\n",ttkl/ko);
getch();

*ttot+=1000;
fprintf(f1,"%u      %f      %f      %f      %f\n",ko,ttq,
ttn1/no,ttml,ttkl/ko);
} /* for different no. of buses */
fclose(f1);

}

```

```

/*This Simulation has been done for Performance Analysis of
Circuit Switched Synchronous System. Equal Priority protocol
has been used. */

```

```

#include <stdio.h>
#include <math.h>
#include <alloc.h>
#include <time.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
#define no 18
#define mo 18
#define PT 10
#define CT 20
#define CLT 5
typedef struct processor{
    unsigned n_e;
    unsigned m_no;
    unsigned b_no;
    unsigned time;
    }pro;

typedef struct buses {
    unsigned state;
    unsigned p_no;
    unsigned time;
    }bus;

typedef struct memory{
    unsigned state;
    unsigned p_no;
    unsigned time;
    }mem;

typedef struct controller {
    unsigned time;
    unsigned p_no;
    struct controller *next;
    }con;

FILE *fg;
unsigned *ko,*q,*m1,*n1,*k1,rcount,t1,t2,q_last,n1_last,
m1_last,k1_last, *ttot,pr;
/* Subroutine for memory update */

ud_m(mem m[],unsigned *t)
{
    unsigned i;
    for(i=0;i<mo;i++)
    {
        if(m[i].state==0)

```

```

    m[i].time = *t;
  }
}

```

/\* Subroutine for bus update \*/

```

ud_b(bus b[], unsigned *t)

```

```

{
  FILE *f3;
  unsigned i;
  for(i=0; i < *ko; i++)
  {
    if(b[i].state==0)
      b[i].time = *t;
  }

  f3=fopen("file3.dat","a+");
  for(i=0; i < *ko; i++)
  {
    fprintf(f3,"b[%u].time = %u    b[%u].state = %u \n",i,
      b[i].time,i,b[i].state);
  }
  fprintf(f3,"\n\n");
  fclose(f3);
}

```

/\* Subroutine for smallest bus time \*/

```

unsigned s_time(bus b[])
{

```

```

  unsigned smallest,i;
  i=0;
  smallest=b[i].time;
  for(i=1; i < *ko; i++)
  {
    if(smallest>b[i].time)
    {
      smallest=b[i].time;
      w=i;
    }
  }

```

```

  printf("\n*** smallest = %u    bus no.= %3d \n",smallest,w);
  return smallest;
}

```

/\* If a processor submit request at a time other than at the beginning of system cycle processor has to wait and the routine below takes account for that \*/

```

wait(unsigned *s1, unsigned *t, con *start, pro p[],
  unsigned clock)
{
  unsigned del,i;
  i=random(mo);
  del = clock - p[*s1].time;    /* wait del time */

```

```

p[*s1].time+=del;
*t+=del;
*q+=1; /* put the processor in queue */
*n1-=1;
p[*s1].m_no=i;
p[*s1].n_e = 2; /*it's next event will be end of next
transition */

/* Put the processor in appropriate position in event queue */
insert(s1,t,start);
}

/* When processor ends processing it has to perform the
following things */

eop(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
unsigned i,j,del;
j=random(mo);
/* If the required memory is busy then processor waits for
next transition */
fg = fopen("fileg.dat","a+");
fprintf(fg,"-eop- processor ended processing\n");
fprintf(fg,"*t = %u *s1 = %u m_no = %u mem->state = %u\n",
*t, *s1,j,m[j].state);
fclose(fg);
if(m[j].state ==1 )
{
fg=fopen("fileg.dat","a+");
fprintf(fg,"requested memory is busy\n");
del = clock - p[*s1].time;
fprintf(fg," del = %u\n",del);
p[*s1].time+=del;
p[*s1].n_e=2;
p[*s1].m_no=j;
*q+=1;*n1-=1;
if(pr==1)
printf("%3d %3d %3d %3d %3d - - -\n",*t, *q,
*n1,*m1,*k1);
*t+=del;
fprintf(fg,"processor %3d memory %3d bus %3d occupied",
*s1,p[*s1].b_no,p[*s1].m_no);
fclose(fg);
insert(s1,t,start);
}
/* Also if there is no bus free then processor waits for next
transition */
else{
for(i=0;i < *ko;i++)
{
if(b[i].state == 0)
break;
}
}
}

```

```

if(i == *ko)
{
    fg=fopen("fileg.dat","a+");
    fprintf(fg,"There is no free bus \n");
    del = clock - p[*s1].time;
    fprintf(fg,"here = %u\n",s_time(b));
    fprintf(fg,"del = %u\n",del);
    *q+=1;*n1-=1;
    p[*s1].time+=del;
    p[*s1].m_no=j;
    p[*s1].n_e=2;
    if(pr==1)
        printf("%3d %3d %3d %3d %3d      -      -      -\n",*t, *q,
            *n1,*m1,*k1);
    fclose(fg);
    *t+=del;
    insert(s1,t,start);
}

else
{
    fg=fopen("fileg.dat","a+");
    fprintf(fg,"It is a success\n occupied bus no = %u\n" ,i);
    b[i].state=1;
    m[j].state=1;
    b[i].p_no=*s1;
    m[j].p_no=*s1;
    p[*s1].m_no=j;
    p[*s1].b_no=i;
    p[*s1].n_e=1;
    del = CT;
    fprintf(fg," del = %u\n ",del);
    p[*s1].time+=del;
    m[j].time+=del;
    b[i].time+=del;
    fprintf(fg,"b[%u].time = %u del = %u\n",i,b[i].time,del);
    *m1+=1;
    *k1+=1;
    if(pr==1)
        printf("%3d %3d %3d %3d %3d %3d %3d %3d\n",
            *t,*q,*n1,*m1,*k1, *s1,p[*s1].m_no,p[*s1].b_no);
    fclose(fg);
    *t+=del;
    insert(s1,t,start);
}
}
fg=fopen("fileg.dat","a+");
fprintf(fg,"Insertion time = %u\n",*t);
fprintf(fg,"eop ended\n\n\n");
fclose(fg);
}

/* subroutine processor releases memory has started */
prm(unsigned *s1,unsigned *t,con *start,pro p[],
    mem m[],bus b[],unsigned clock)

```

```

{
unsigned del;
fg=fopen("fileg.dat","a+");
fprintf(fg,"-prm- processor releases memory\n");
fprintf(fg,"*t = %u *s1 = %u m_no = %u \n",*t,*s1,p[*s1].m_no);
fprintf(fg,"freed bus no = %u\n",p[*s1].b_no);
del=random(PT)+1;
fprintf(fg,"del = %u\n",del);
p[*s1].time+=del;
p[*s1].n_e=0;
b[p[*s1].b_no].state=0;
m[p[*s1].m_no].state=0;
b[p[*s1].b_no].p_no=0;
m[p[*s1].m_no].p_no=0;
*k1-=1;*m1-=1;
if(pr==1)
    printf("%3d %3d %3d %3d %3d %3d %3d %3d \n",
        *t,*q,*n1,*m1,*k1,*s1,p[*s1].m_no,p[*s1].b_no);
*t+=del;
fclose(fg);
insert(s1,t,start);
fg=fopen("fileg.dat","a+");
fprintf(fg,"Insertion time = %u\n",*t);
fprintf(fg,"prm ended\n\n\n");
fprintf(fg,"processor %3d memory %3d bus %3d released",
*s1,p[*s1].b_no,p[*s1].m_no);
fclose(fg);
}
/* subroutine end of next transition has started */
ent(unsigned *s1,unsigned *t,con *start,pro p[],
mem m[],bus b[],unsigned clock)
{
    extern unsigned *ko,*q,*n1,*m1,*k1;
    unsigned i,del,dell;
    fg=fopen("fileg.dat","a+");
    fprintf(fg,"-ent- end of next transition\n");
    fprintf(fg,"*t = %u *s1 = %u m_no = %u mem->state=%u\n",
        *t, *s1,p[*s1].m_no,m[p[*s1].m_no].state);
    fclose(fg);
    for(i=0;i < *ko;i++)
    {
        if(b[i].state==0)
            break;
    }
    if(i == *ko)
    {
        fg=fopen("fileg.dat","a+");
        fprintf(fg,"There is no bus free\n");
        dell= s_time(b);
        del = clock - p[*s1].time; /* dell - (*t); */
        fprintf(fg,"herel = %u\n",dell);
        fprintf(fg,"del = %u\n",del);
        fprintf(fg,"del = %u\n",del);
        p[*s1].time+=del;
        p[*s1].n_e=2;
    }
}

```

```

if(pr==1)
    printf("%3d %3d %3d %3d %3d      -      -      -\n",
           *t,*q,*n1,*m1,*k1);
fclose(fg);
*t+=del;
insert(s1,t,start);
}
else
{
    if(m[p[*s1].m_no].state==1)
    {
        fg=fopen("fileg.dat","a+");
        fprintf(fg,"requested memory is still busy\n");
        fprintf(fg,"and memory time = %u\n",m[p[*s1].m_no].time);
        del=clock - p[*s1].time;
        fprintf(fg,"del = %u\n",del);
        p[*s1].time+=del;
        p[*s1].n_e=2;
        if(pr==1)
            printf("%3d %3d %3d %3d %3d      -      -      -\n",
                   *t,*q,*n1, *m1,*k1);
        fclose(fg);
        *t+=del;
        insert(s1,t,start);
    }
else
{
    fg=fopen("fileg.dat","a+");
    fprintf(fg,"Now it is a success\n");
    fprintf(fg,"Occupied bus no = %u and smallest = %u\n",
           i,dell);
    *q-=1;
    del = CT;
    fprintf(fg,"del = %u\n",del);
    p[*s1].time+=del;
    b[i].state=1;
    m[p[*s1].m_no].time+=del;
    b[i].time+=del;
    p[*s1].n_e=1;
    p[*s1].b_no=i;
    m[p[*s1].m_no].state=1;
    b[i].p_no=*s1;
    m[p[*s1].m_no].p_no=*s1;
    *n1+=1;*m1+=1;*k1+=1;
    result(tq,tn1,tm1,tk1,t);
    if(pr==1)
        printf("%3d %3d %3d %3d %3d      %3d %3d %3d \n",
               *t,*q,*n1,*m1,*k1,*s1,p[*s1].m_no,p[*s1].b_no);
    fclose(fg);
    *t+=del;
    insert(s1,t,start);
}
}
}
fg=fopen("fileg.dat","a+");
fprintf(fg,"Insertion time = %u\n",*t);

```

```

fprintf(fg,"ent ended \n\n\n");
fclose(fg);
}

/* subroutine insert has started below */

insert(unsigned *s1,unsigned *t,con *start)
{
con *prev,*new,*current;
current=start->next;
if(*t < current->time)
{
start->p_no = *s1;
start->time = *t;
}
else
{
if((new=(con*)malloc(sizeof(con)))==NULL)
{
printf("No memory available for allocation \n");
exit(1);
}
start->p_no=current->p_no;
start->time=current->time;
start->next=current->next;
free(current);
current=start->next;
prev=start;
while(current->next != NULL && *t >= current->time)
{
prev=current;
current=current->next;
}
new->p_no=*s1;
new->time=*t;
if(current->next != NULL)
{
prev->next=new;
new->next=current;
}
else
{
if(*t >= current->time)
{
current->next=new;
new->next=NULL;
}
else
{
prev->next=new;
new->next=current;
}
}
}
}

```



```

fg=fopen("fileg.dat","a+");
current=start;
fprintf(fg,"statistics on insertion\n");
while(current)
{
fprintf(fg,"          %u          %u\n",
        current->time,current->p_no);
current=current->next;
}
fclose(fg);
}
/* subroutine for random number generation */
g_rand(num)
{
double r,log();
unsigned x;
randomize();
r=(double)(random(100)/100.00);
r-=1;
x=-log(-r)/log(.49);
return(x);
}

result(unsigned *tq,unsigned *tn1,unsigned *tm1,
        unsigned *tk1,unsigned *t)
{
unsigned sub;
if(rcount == 0)
{
t1 = *t;
q_last = *q;
n1_last = *n1;
m1_last = *m1;
k1_last = *k1;
rcount+=1;
}
else
{
t2 = *t;
sub=t2-t1;
*tq+=q_last * sub;
*tn1+=n1_last * sub;
*tm1+=m1_last * sub;
*tk1+=k1_last * sub;
/* ***** */
t1=t2;
q_last = *q;
n1_last = *n1;
m1_last = *m1;
k1_last = *k1;
}
}
}
/* For equal priority protocol */
eq_sort(con *tag,unsigned *e_count)

```

```

{
con *current,*inter,*first;
int i,store;
first = tag;
current = tag;
inter=(con*)malloc(sizeof(con));
for(i=*e_count;i>0;i--)
{
store=random(i);
while(store)
{
store-=1;
current=current->next;
}
inter->p_no=first->p_no;
first->p_no=current->p_no;
current->p_no=inter->p_no;
first=first->next;
current=first;
}
free(inter);
}
/* For palcing processors in appropriate position in event
queue */
sort(pro p[],con *start,unsigned *e_count)
{
FILE *fp1;
unsigned back,store,prm_cnt;
con *current,*inter,*tag;

back=0;prm_cnt=0;
tag=start;
current=start;
store=*e_count;
if((fp1=fopen("f1.dat","a+"))==NULL)
{
printf("file error");
exit(1);
}
while(*e_count)
{
*e_count-=1;
if(p[current->p_no].n_e==1 && back==0)
{
tag=tag->next;
current=current->next;
prm_cnt+=1;
}
else if(p[current->p_no].n_e==1 && back>0)
{
inter=(con*)malloc(sizeof(con));
inter->p_no=tag->p_no;
tag->p_no=current->p_no;
current->p_no=inter->p_no;
free(inter);
}
}
}

```

```

        prm_cnt+=1;
        tag=tag->next;
        current=current->next;
    }
    else
    {
        current=current->next;
        back+=1;
    }
}
fprintf(fp1, "\t\t%u      %u\n", store, prm_cnt);

current=start;

while(current)
{
    fprintf(fp1, "\t%u      %u      %u\n", p[current->p_no].n_e,
    current->time, current->p_no);
    current=current->next;
}
fclose(fp1);
*e_count =store;
*e_count-=prm_cnt;
eq_sort(tag, e_count);
*e_count=store;
}

/* Main program is started below */
main()
{
    FILE *fp;
    con *start,*prev,*new,*current;
    pro p[no];
    mem m[no];
    bus b[18];
    unsigned *tq,*tn1,*tm1,*tk1,clock,*e_count,num;
    unsigned i,j,*s_c,r[no],*sl,count,scr_cnt,*t;
    float ttq,ttn1,ttm1,ttk1;
    void *calloc();
    void *malloc();
    if ((ko=(unsigned*)malloc(sizeof(unsigned)))==NULL)
    {
        printf("No memory");
        exit(1);
    }
    if((fp=fopen("f.dat","a+"))==NULL)
    {
        printf("file error");
        exit(1);
    }

    *ko=0;
    pr=0;
    for(*ko = 1;(*ko) < 17;(*ko)++ )
    {

```

```

clrscr();
rcount=0; t1=0;t2=0; q_last=0; clock=CLT;
n1_last=0; m1_last=0; k1_last=0;
ttot=(unsigned*)malloc(sizeof(unsigned));
n1=(unsigned*)malloc(sizeof(unsigned));
m1=(unsigned*)malloc(sizeof(unsigned));
k1=(unsigned*)malloc(sizeof(unsigned));
q=(unsigned*)malloc(sizeof(unsigned));
t=(unsigned*)malloc(sizeof(unsigned));
s1=(unsigned*)malloc(sizeof(unsigned));
tq=(unsigned*)malloc(sizeof(unsigned));
tn1=(unsigned*)malloc(sizeof(unsigned));
tm1=(unsigned*)malloc(sizeof(unsigned));
tk1=(unsigned*)malloc(sizeof(unsigned));
e_count=(unsigned*)malloc(sizeof(unsigned));
s_c=(unsigned*)malloc(sizeof(unsigned));

*n1=no;*m1=0;*k1=0;*q=0;*ttot=3000;
*tq=0;*tn1=0;*tm1=0;*tk1=0;

if((start=( con*)malloc(sizeof( con)))==NULL)
{
    printf("No memory available for allocation \n");
    exit(1);
}

/* Random numbers are generated below */
randomize();
num = PT;
for(i=0;i < no;i++)
    r[i]=g_rand(num)+1;

/* Controllar linked list is formed below */
start->next=NULL;
start->p_no=0;
start->time=r[0];
for(i=1;i<no;i++)
{
    if((current=( con*)malloc(sizeof( con)))==NULL)
    {
        printf("No memory");
        exit(1);
    }
    current->time=r[i];
    current->p_no=i;
    if(current->time < start->time)
    {
        prev=start;
        current->next=prev;
        start=current;
    }
    else
    {
        prev=start;
        new=start->next;

```

```

while(new != NULL && current->time >= prev->time)
{
if(current->time < new->time)
{
current->next=new;
prev->next=current;
prev=new;
}
else
{
prev=new;
new=new->next;
}
}
if(new == NULL)
{
prev->next=current;
current->next=NULL;
}
}

fg=fopen("fileg.dat","a+");
fprintf(fg,"statistics on enter\n");
current=start;
while(current)
{
fprintf(fg,"          %u          %u\n",
          current->time,current->p_no);
current=current->next;
}
fclose(fg);
/*initialization of arrays are done below */
for(i=0;i< no;i++)
{
p[i].n_e=0;
p[i].m_no=0;
p[i].b_no=0;
p[i].time=r[i]; /* A random number */
}

for(i=0;i<mo;i++)
{
m[i].state=0;
m[i].p_no=0;
m[i].time=0;
}

for(i=0;i < (*ko);i++)
{
b[i].state=0;
b[i].p_no=0;
b[i].time=0;
}

```

```

if(pr==1)
{
printf("Time   In   busy   In   In   PE   MM   B \n");
printf("      Q     PE    bus  MM   no  no  no \n");
gotoxy(2,24);
printf("PRESS ANY KEY TO SEE NEXT PAGE");
/*Finding of current event\events are started below */
window(1,3,79,20);
scr_cnt=0;
clrscr();
gotoxy(1,1);
}
*t=start->time;
for(i=0;i<mo;i++)
    m[i].time = *t;
for(i=0;i < *ko;i++)
    b[i].time = *t;

while(*t < *ttot)
{
*e_count=1;
current=start;
new=current->next;
while(current->time == new->time && new != NULL)
{
*e_count+=1;
current=new;
new=current->next;
}

sort(p,start,e_count);
if(clock < *t)
{
while(clock < *t)
clock+= CLT;
}

if(*t == clock)
{
clock+=CLT;
ud_m(m,t);
ud_b(b,t);
while(*e_count)
{
*e_count-=1;
*s1=start->p_no;
*s_c=p[*s1].n_e;
switch(*s_c)
{
case 0:
{
eop(s1,t,start,p,m,b,clock);
break;
}
}
}
}

```

```

case 1:
{
    prm(s1,t,start,p,m,b,clock);
    break;
}
case 2:
{
    ent(s1,t,start,p,m,b,clock);
    break;
}
}

*t=start->time;
if(*e_count == 0)
{
    if(*t>1000)
        result(tq,tn1,tm1,tk1,t);
    ud_m(m,t);
    ud_b(b,t);
}
scr_cnt+=1;
if(pr==1)
{
    if(scr_cnt == 17)
    {
        getch();
        scr_cnt=0;
        clrscr();
        gotoxy(1,1);
    }
}

fg=fopen("fileg.dat","a+");
fprintf(fg,"\nb[0]=%3d  b[1]=%3d  b[2]=%3d  b[4]=%3d\n",
b[0].state,b[1].state,b[2].state,b[3].state);
fprintf(fg,"\ntb[0]=%3d  tb[1]=%3d  tb[2]=%3d  tb[4]=%3d\n",
b[0].time,b[1].time,b[2].time,b[3].time);
fprintf(fg,"*****\n");
fclose(fg);
}
else
{
while(*e_count)
{
    *e_count-=1;
    *s1=start->p_no;
    wait(s1,t,start,p,clock);
    *t=start->time;
}
if(*t>1000)
    result(tq,tn1,tm1,tk1,t);
}

```

```

}

getch();
window(1,1,79,24);
clrscr();
*ttot-=1000;
ttq=((float)(*tq))/((float)(*ttot));
ttn1=((float)(*tn1))/((float)(*ttot));
ttm1=((float)(*tml))/((float)(*ttot));
ttk1=((float)(*tk1))/((float)(*ttot));
printf("\n\tAverage Queue Length=%f\n",ttq );
printf("\n\tAverage No of busy Processor =%f\n",ttn1);
printf("\n\tAverage No of Busy Memory =%f\n",ttm1);
printf("\n\tAverage No of Busy Bus =%f\n",ttk1);
printf("\n\tProcessor Utilization =%f\n",ttn1/no);
printf("\n\tMemory Bandwidth=%f\n",ttm1/mo);
printf("\n\tBus Utilization=%f\n",ttk1/(*ko));
*ttot+=1000;
getch();
fprintf(fp,"%u      %f      %f      %f      %f\n",*ko,
ttq,ttn1/no,ttm1,ttk1/(*ko));

}
fclose(fp);
}

```



```

/*This Simulation program is for Packet Switched Synchronous
  System. Equal priority protocol is used */
#include <stdio.h>
#include <math.h>
#include <alloc.h>
#include <time.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
#define no 16
#define mo 16
#define PT 1
#define BUST 5
#define MEMT 20
#define CINC 5
typedef struct processor
  {
    unsigned n_e;
    unsigned m_no;
    unsigned b_no;
    unsigned time;
  }pro;

typedef struct buses {
    unsigned state;
    unsigned p_no;
    unsigned time;
  }bus;

typedef struct memory{
    unsigned state;
    unsigned p_no;
    unsigned time;
    unsigned q;
  }mem;

typedef struct controller {
    unsigned time;
    unsigned p_no;
    struct controller *next;
  }con;

FILE *fg;
unsigned *ko,*q,*m1,*n1,*k1,rcount,t1,t2,q_last,n1_last,
  m1_last,k1_last;
unsigned *ttot,q_flast,q_blast,*qbf,*qbb,*q,pr;
Subroutine for memory update

ud_m(mem m[],unsigned *t)
{
  unsigned i;

```

```

for(i=0;i<mo;i++)
{
    if(m[i].state==0)
        m[i].time = *t;
}
}

/* Subroutine for bus update */

ud_b(bus b[],unsigned *t)
{
    FILE *f3;
    unsigned i;
    for(i=0;i < *ko;i++)
    {
        if(b[i].state==0)
            b[i].time = *t;
    }

    f3=fopen("file3.dat","a+");
    for(i=0;i < *ko;i++)
    {
        fprintf(f3,"b[%u].time = %u    b[%u].state = %u \n",i,
            b[i].time,i,b[i].state);
    }
    fprintf(f3,"\n\n");
    fclose(f3);

}

/* Subroutine for smallest bus time */
unsigned s_time(bus b[])
{
    unsigned smallest,i;
    i=0;
    smallest=b[i].time;
    for(i=1;i < *ko;i++)
    {
        if(smallest>b[i].time)
        {
            smallest=b[i].time;
            /* w=i; */
        }
    }

    printf("\n*** smallest = %u    bus no.= %3d \n",smallest,w);
    return smallest;
}
/* ***** */

wait(unsigned *s1,unsigned *t,con *start,pro p[],
    unsigned clock)

```

```

{
    unsigned del,i;
    i=random(mo);
    del = clock - p[*s1].time;
    p[*s1].time+=del;
    p[*s1].n_e=2;
    p[*s1].m_no=i;
    *qbf+=1;
    *q+=1;
    *t+=del;*n1-=1;
    insert(s1,t,start);
}

/* After think time processor comes here */
eop(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
    bus b[],unsigned clock)
{
    unsigned i,del;
    for(i=0;i<*ko;i++)
    {
        if(b[i].state==0)
            break;
    }
    if(i==*ko)
    {
        del= clock-p[*s1].time; /*CINC increment of clock*/
        p[*s1].time+=del;
        *qbf+=1;*n1-=1;
        *q+=1;
        p[*s1].n_e=2; /*entbf()*/
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,*n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        del= BUST;
        p[*s1].time+=del;
        p[*s1].n_e=1;/*eobf*/
        *k1+=1;
        b[i].state=1;
        b[i].p_no=*s1;
        p[*s1].b_no=i;
        b[i].time+=del;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
}
}

```

```

/* After transferring informations through bus
processor comes here */
eobf(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
    unsigned i,del;
    b[p[*s1].b_no].state=0;
    *n1-=1; /* whether or not free memory is found processor
            remains idle */
    *k1-=1; /* and bus is freed */
    i=random(mo);
    if(m[i].state==1)
    {
        del=clock-p[*s1].time;
        p[*s1].time+=del;
        p[*s1].n_e=3;
        p[*s1].m_no=i;
        m[i].q+=1;
        *q+=1;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        m[i].state=1;
        m[i].p_no=*s1;
        p[*s1].m_no=i;
        p[*s1].n_e=4; /*prm*/
        del=MEMT; /*memory access time */
        p[*s1].time+=del;
        m[i].time+=del;
        *m1+=1;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
}

/* If once in forward direction busy bus condition is
found processor comes here */

entbf(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
    unsigned i,del;
    for(i=0;i<*ko;i++)
    {
        if(b[i].state==0)
            break;
    }
}

```

```

if(i==*ko)
{
    del=CINC; /*increment of clock*/
    p[*s1].time+=del;
    p[*s1].n_e=2; /*entbf()*/
    if(pr==1)
        printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,*n1,
            *m1,*k1);
    *t+=del;
    insert(s1,t,start);
}
else
{
    del= BUST;
    p[*s1].time+=del;
    p[*s1].n_e=1; /*eobf*/
    *k1+=1; *n1+=1;
    *qbf-=1; /* this processor was in bus queue */
    *q-=1;
    b[i].state=1;
    b[i].p_no=*s1;
    p[*s1].b_no=i;
    b[i].time+=del;
    if(pr==1)
        printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
            *n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
}
}

/* If once required memory is found in busy state then
processor comes here */
entm(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
    unsigned i,del;
    i=p[*s1].m_no;
    if(m[i].state==1)
    {
        del=clock-p[*s1].time;
        p[*s1].time+=del;
        p[*s1].n_e=3;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        m[i].state=1;
        m[i].p_no=*s1;
        p[*s1].m_no=i;
        p[*s1].n_e=4; /*prm*/
    }
}

```

```

del=MEMT; /*memory access time */
p[*s1].time+=del;
m[i].time+=del;
m[i].q-=1;
*q-=1;
*m1+=1;
if(pr==1)
    printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,
           *q,*n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
}
}

/* When memory access ends processor comes here to see
if there is any bus free */
prm(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
    unsigned i,j,del;
    j=p[*s1].m_no;
    *m1-=1; /* memory goes to idle state */
    for(i=0;i<*ko;i++)
    {
        if(b[i].state==0)
            break;
    }
    if(i==*ko)
    {
        del=clock-p[*s1].time; /*CINC is increment of clock*/
        p[*s1].time+=del;
        *qbb+=1;
        *q+=1;
        p[*s1].n_e=5; /*entbb()*/
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                   *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        del= BUST;
        p[*s1].time+=del;
        p[*s1].n_e=8; /*eobb*/
        *k1+=1;
        *n1+=1;
        b[i].state=1;
        b[i].p_no=*s1;
        p[*s1].b_no=i;
        b[i].time+=del;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                   *n1,*m1,*k1);
        *t+=del;
    }
}

```

```

    insert(s1,t,start);
}
}

/* If in backward direction busy bus condition is
found then processor comes here */

entbb(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
    unsigned i,j,del;
    for(i=0;i<*ko;i++)
    {
        if(b[i].state==0)
            break;
    }
    if(i==*ko)
    {
        del=CINC; /*increment of clock*/
        p[*s1].time+=del;
        p[*s1].n_e=5; /*entbb()*/
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,*n1,
                *m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        del= BUST;
        p[*s1].time+=del;
        p[*s1].n_e=6; /*eobb*/
        *k1+=1; *qbb-=1;
        *q-=1;
        b[i].state=1;
        *n1+=1;
        b[i].p_no=*s1;
        p[*s1].b_no=i;
        b[i].time+=del;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
}

/* In backward direction after getting bus processor
comes in thinking state */
eobb(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
    unsigned i,del;

```

```

b[p[*s1].b_no].p_no=0;
del=random(PT) + 1;
p[*s1].time+=del;
p[*s1].n_e=0;
*k1-=1;
if(pr==1)
    printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
           *n1,*m1,*k1);
*t+=del;
insert(s1,t,start);
}

```

/\* subroutine insert has started below \*/

```

insert(unsigned *s1,unsigned *t,con *start)
{
con *prev,*new,*current;
current=start->next;
if(*t < current->time)
{
start->p_no = *s1;
start->time = *t;
}
else
{
if((new=(con*)malloc(sizeof(con)))==NULL)
{
printf("No memory available for allocation \n");
exit(1);
}
start->p_no=current->p_no;
start->time=current->time;
start->next=current->next;
free(current);
current=start->next;
prev=start;
while(current->next != NULL && *t >= current->time)
{
prev=current;
current=current->next;
}
new->p_no=*s1;
new->time=*t;
if(current->next != NULL)
{
prev->next=new;
new->next=current;
}
else
{
if(*t >= current->time)
{
current->next=new;
new->next=NULL;
}
}
}

```



```

    }
else
    {
        prev->next=new;
        new->next=current;
    }
}

fg=fopen("fileg.dat","a+");
current=start;
fprintf(fg,"statistics on insertion\n");
while(current)
{
    fprintf(fg,"          %u          %u\n",current->time,
            current->p_no);
    current=current->next;
}
fclose(fg);
}

result(unsigned *tq,unsigned *tqf,unsigned *tqb,unsigned *tn1,
unsigned *tm1, unsigned *tk1,unsigned *t)
{
    unsigned sub;
    if(rcount == 0)
    {
        t1 = *t;
        q_last = *q;
        q_flast= *qbf;
        q_blast= *qbb;
        n1_last = *n1;
        m1_last = *m1;
        k1_last = *k1;
        rcount+=1;
    }
else
    {
        t2 = *t;
        sub=t2-t1;
        *tq+=q_last * sub;
        *tqf+=q_flast * sub;
        *tqb+=q_blast * sub;
        *tn1+=n1_last * sub;
        *tm1+=m1_last * sub;
        *tk1+=k1_last * sub;
        t1=t2;
        q_last = *q;
        q_flast = *qbf;
        q_blast = *qbb;
        n1_last = *n1;
        m1_last = *m1;
        k1_last = *k1;
    }
}

```

```

}
/* This sor is for equal priority protocol */
eq_sort(con *tag,unsigned *e_count)
{
    con *current,*inter,*first;
    int i,store;
    first = tag;
    current = tag;
    inter=malloc(sizeof(con));
    for(i=*e_count;i>0;i--)
    {
        store=random(i);
        while(store)
        {
            store-=1;
            current=current->next;
        }
        inter->p_no=first->p_no;
        first->p_no=current->p_no;
        current->p_no=inter->p_no;
        first=first->next;
        current=first;
    }
    free(inter);
}

/* This sort is for arrangement in event queue */
sort(pro p[],con *start,unsigned *e_count,mem m[], bus b[])
{
    FILE *fp1;
    unsigned back,store,prm_cnt;
    con *current,*inter,*tag;

    back=0;prm_cnt=0;
    tag=start;
    current=start;
    store=*e_count;
    if((fp1=fopen("f1.dat","a"))==NULL)
    {
        printf("file error");
        exit(1);
    }
    while(*e_count)
    {
        *e_count-=1;
        if((p[current->p_no].n_e==1 || p[current->p_no].n_e==4 ||
            p[current->p_no].n_e==6)&& back==0)
        {
            if(p[current->p_no].n_e==1 || p[current->p_no].n_e==6)
                b[p[current->p_no].b_no].state=0;
            else
                m[p[current->p_no].m_no].state=0;
            tag=tag->next;
            current=current->next;
            prm_cnt+=1;
        }
    }
}

```

```

    }
    else if((p[current->p_no].n_e==1 || p[current->p_no].n_e==4
           || p[current->p_no].n_e==8 )&& back>0)
    {
        if(p[current->p_no].n_e==1 || p[current->p_no].n_e==8)
            b[p[current->p_no].b_no].state=0;
        else
            m[p[current->p_no].m_no].state=0;

        inter=(con*)malloc(sizeof(con));
        inter->p_no=tag->p_no;
        tag->p_no=current->p_no;
        current->p_no=inter->p_no;
        free(inter);
        prm_cnt+=1;
        tag=tag->next;
        current=current->next;
    }
    else
    {
        current=current->next;
        back+=1;
    }
}
fprintf(fp1, "\t\t%u      %u\n", store, prm_cnt);

current=start;

while(current)
{
    fprintf(fp1, "\t%u      %u      %u\n", p[current->p_no].n_e,
    current->time, current->p_no);
    current=current->next;
}
fclose(fp1);
eq_sort(tag, e_count);
*e_count=store;
}
/* Main program has started below */

main()
{
    FILE *fp;
    con *start, *prev, *new, *current;
    pro p[no];
    mem m[mo];
    bus b[16];
    unsigned *tq, *tn1, *tm1, *tk1, clock, *tqb, *tqf;
    unsigned i, j, *s_c, *e_count, r[no], *s1, count, scr_cnt, *t;
    float ttq, ttn1, ttm1, ttk1, ttqb, ttqf, ttmq;
    void *calloc();
    void *malloc();
    if ((ke=(unsigned*)malloc(sizeof(unsigned)))==NULL)
    {
        printf("No memory");
    }
}

```

```

    exit(1);
}
if((fp=fopen("f.dat","a+"))==NULL)
{
    printf("file error");
    exit(1);
}

*ko=0;
pr=0;
for(*ko = 1;(*ko) < 17;(*ko)++ )
{
    clrscr();
    rcount=0; t1=0;t2=0; q_flast=0; q_blast=0;q_last=0;clock=5;
    scr_cnt=0;n1_last=0; m1_last=0; k1_last=0;
    qbb=malloc(sizeof(unsigned));
    qbf=malloc(sizeof(unsigned));
    tqb=malloc(sizeof(unsigned));
    tqf=malloc(sizeof(unsigned));
    ttot=(unsigned*)malloc(sizeof(unsigned));
    n1=(unsigned*)malloc(sizeof(unsigned));
    m1=(unsigned*)malloc(sizeof(unsigned));
    k1=(unsigned*)malloc(sizeof(unsigned));
    q=(unsigned*)malloc(sizeof(unsigned));
    t=(unsigned*)malloc(sizeof(unsigned));
    s1=(unsigned*)malloc(sizeof(unsigned));
    tq=(unsigned*)malloc(sizeof(unsigned));
    tn1=(unsigned*)malloc(sizeof(unsigned));
    tm1=(unsigned*)malloc(sizeof(unsigned));
    tk1=(unsigned*)malloc(sizeof(unsigned));
    e_count=(unsigned*)malloc(sizeof(unsigned));
    s_c=(unsigned*)malloc(sizeof(unsigned));

    *n1=no;*m1=0;*k1=0;*qbb=0;*qbf=0;*q=0;*ttot=3000;
    *tq=0;*tqf=0;*tqb=0;*tn1=0;*tm1=0;*tk1=0;

    if((start=( con*)malloc(sizeof( con)))==NULL)
    {
        printf("No memory available for allocation \n");
        exit(1);
    }

    /* Random numbers are generated below */
    randomize();
    for(i=0;i < no;i++)
        r[i]=random(PT)+1;

    /* Controllar linked list is formed below */
    start->next=NULL;
    start->p_no=0;
    start->time=r[0];
    for(i=1;i<no;i++)
    {
        if((current=( con*)malloc(sizeof( con)))==NULL)
        {

```

```

printf("No memory");
exit(1);
}
current->time=r[i];
current->p_no=i;
if(current->time < start->time)
{
prev=start;
current->next=prev;
start=current;
}
else
{
prev=start;
new=start->next;
while(new != NULL && current->time >= prev->time)
{
if(current->time < new->time)
{
current->next=new;
prev->next=current;
prev=new;
}
else
{
prev=new;
new=new->next;
}
}
if(new == NULL)
{
prev->next=current;
current->next=NULL;
}
}
}
fg=fopen("fileg.dat","a+");
fprintf(fg,"statistics on enter\n");
current=start;
while(current)
{
fprintf(fg,"          %u          %u\n",current->time,
current->p_no);
current=current->next;
}
fclose(fg);
/*initialization of arrays are done below */
for(i=0;i<no;i++)
{
p[i].n_e=0;
p[i].m_no=0;
p[i].b_no=0;
p[i].time=r[i]; /* A random number */
}
for(i=0;i<mo;i++)

```

```

{
    m[i].state=0;
    m[i].p_no=0;
    m[i].time=0;
    m[i].q=0;
}

for(i=0;i < (*ko);i++)
{
    b[i].state=0;
    b[i].p_no=0;
    b[i].time=0;
}

if(pr==1)
{
    printf("    Time    PE    BUSY    In    In    \n");
    printf("          no    PE    MEM    BUS    \n");
    gotoxy(2,24);
    printf("PRESS ANY KEY TO SEE NEXT PAGE");
    /*Finding of current event\events are started below */
    window(1,3,79,20);
    scr_cnt=0;
    clrscr();
    gotoxy(1,1);
}

*t=start->time;
for(i=0;i<mo;i++)
    m[i].time = *t;
for(i=0;i < *ko;i++)
    b[i].time = *t;

while(*t < *ttot){
    *e_count=1;
    current=start;
    new=current->next;
    while(current->time == new->time && new != NULL)
    {
        *e_count+=1;
        current=new;
        new=current->next;
    }
    sort(p,start,e_count,m,b);
    if(*t==clock)
    {
        clock+=5;
        while(*e_count)
        {
            *e_count-=1;
            *s1=start->p_no;
            *s_c=p[*s1].n_e;
            switch(*s_c)
            {
                case 0:

```

```

    {
        eop(s1,t,start,p,m,b,clock);
        break;
    }
    case 1:
    {
        eobf(s1,t,start,p,m,b,clock);
        break;
    }
    case 2:
    {
        entbf(s1,t,start,p,m,b,clock);
        break;
    }
    case 3:
    {
        entm(s1,t,start,p,m,b,clock);
        break;
    }
    case 4:
    {
        prm(s1,t,start,p,m,b,clock);
        break;
    }
    case 5:
    {
        entbb(s1,t,start,p,m,b,clock);
        break;
    }
    case 6:
    {
        eobb(s1,t,start,p,m,b,clock);
        break;
    }
}
*t=start->time;
if(clock < *t)
{
    while(clock < *t)
        clock+=CINC;
}
if(*e_count == 0)
{
    if(*t > 1000)
        result(tq,tqf,tqb,tn1,tm1,tk1,t);
    ud_m(m,t);
    ud_b(b,t);
}
scr_cnt+=1;
if(pr==1)
{
    if(scr_cnt == 17)
    {
        getch();
        scr_cnt=0;
    }
}

```

```

        clrscr();
        gotoxy(1,1);
    }
}

fg=fopen("fileg.dat","a+");
fprintf(fg,"\nb[0]=%3d  b[1]=%3d  b[2]=%3d
b[4]=%3d\n",b[0].state,b[1].state,b[2].state,b[3].state);
fprintf(fg,"\ntb[0]=%3d  tb[1]=%3d  tb[2]=%3d  tb[4]=%3d\n",
        b[0].time, b[1].time,b[2].time,b[3].time);
fprintf(fg,"*****\n");
fclose(fg);
}
}
else
{
while(*e_count)
{
    *e_count-=1;
    *s1=start->p_no;
    wait(s1,t,start,p,clock);
    *t=start->time;
}
if(*t >1000)
    result(tq,tqf,tqb,tn1,tm1,tk1,t);
}
}
getch();
window(1,1,79,24);
clrscr();
ttq=0;
for(i=0;i<mo;i++)
ttq+=m[i].q; /*
*ttot-=1000;
ttq=((float)(*tq))/((float)(*ttot));
tn1=((float)(*tn1))/((float)(*ttot));
tm1=((float)(*tm1))/((float)(*ttot));
tk1=((float)(*tk1))/((float)(*ttot));
ttqb=((float)(*tqb))/((float)(*ttot));
ttqf=((float)(*tqf))/((float)(*ttot));
*ttot+=1000;
printf("\n\tAverage Queue Length=%f\n",ttq );
printf("\n\tAverage forward Queue Length=%f\n",ttqf );
printf("\n\tAverage backward Queue Length=%f\n",ttqb );
printf("\n\tAverage No of busy Processor =%f\n",tn1);
printf("\n\tAverage No of Busy Memory =%f\n",tm1);
printf("\n\tAverage No of Busy Bus =%f\n",tk1);
printf("\n\tProcessor Utilization =%f\n",tn1/no);
printf("\n\tMemory Bandwidth=%f\n",tm1/mo);
printf("\n\tBus Utilization=%f\n",tk1/(*ko));
getch();
fprintf(fp,"%u      %f      %f      %f      %f\n",*ko,ttq,
tn1/no,tm1,tk1/(*ko));
}
}

```



```

/* Simulation program for asynchronous packet switched system
   with equal priority protocol. */
#include <stdio.h>
#include <math.h>
#include <alloc.h>
#include <time.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
#define no 16
#define mo 16
#define BUST 5
#define MEMT 20
#define CINC 5
#define PT 1
typedef struct processor{
    unsigned n_e;
    unsigned m_no;
    unsigned b_no;
    unsigned time;
    }pro;

typedef struct buses {
    unsigned state;
    unsigned p_no;
    unsigned time;
    }bus;

typedef struct memory{
    unsigned state;
    unsigned p_no;
    unsigned time;
    unsigned q;
    }mem;

typedef struct controller {
    unsigned time;
    unsigned p_no;
    struct controller *next;
    }con;

unsigned *ko,*q,*m1,*n1,*k1,rcount,t1,t2,q_last,n1_last;
unsigned *ttot,q_flast,q_blast,*qbf,*qbb,*q,pr;
/* Subroutine for memory update */

ud_m(mem m[],unsigned *t)
{
    unsigned i;
    for(i=0;i<mo;i++)
    {
        if(m[i].state==0)
            m[i].time = *t;
    }
}

```

```

}

/* Subroutine for bus update */
ud_b(bus b[],unsigned *t)
{
/*FILE *f3;*/
unsigned i;
for(i=0;i < *ko;i++)
{
if(b[i].state==0)
b[i].time = *t;
}
}

/* Subroutine for smallest bus time */
unsigned s_time(bus b[])
{
unsigned smallest,i;
i=0;
smallest=b[i].time;
for(i=1;i < *ko;i++)
{
if(smallest>b[i].time)
{
smallest=b[i].time;
}
}
return smallest;
}

/* After think time processor comes here */
eop(unsigned *s1,unsigned *t,con *start,pro p[],mem m[],
bus b[],unsigned clock)
{
unsigned i,del;
for(i=0;i<*ko;i++)
{
if(b[i].state==0)
break;
}
if(i==*ko)
{
del= s_time(b) - p[*s1].time; /*CINC increment of clock*/
p[*s1].time+=del;
*qbf+=1;*n1-=1;
*q+=1;
p[*s1].n_e=2; /*entbf()*/
if(pr==1)
printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
*n1,*m1,*k1);
*t+=del;
insert(s1,t,start);
}
}

```

```

}
else
{
    del= BUST;
    p[*s1].time+=del;
    p[*s1].n_e=1; /* next event is eobf*/
    *k1+=1;
    b[i].state=1;
    b[i].p_no=*s1;
    p[*s1].b_no=i;
    b[i].time+=del;
    if(pr==1)
        printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
}
}

/* After transferring informations through bus processor
comes here */
eobf(unsigned *s1,unsigned *t,con *start,pro p[],
mem m[],bus b[],unsigned clock)
{
    unsigned i,del;
    *n1-=1; /* whether or not free memory is found
processor remains idle */
    *k1-=1; /* and bus is freed */
    i=random(mo);
    if(m[i].state==1)
    {
        del=m[i].time - p[*s1].time;
        /* until that memory is freed */
        p[*s1].time+=del;
        p[*s1].n_e=3;
        p[*s1].m_no=i;
        m[i].q+=1;
        *q+=1;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                    *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        m[i].state=1;
        m[i].p_no=*s1;
        p[*s1].m_no=i;
        p[*s1].n_e=4; /*next event is prm*/
        del=MEMT; /*memory access time */
        p[*s1].time+=del;
        m[i].time+=del;
        *m1+=1;
    }
}

```

```

    if(pr==1)
        printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
            *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
}

/* If once in forward direction busy bus condition is found
processor comes here */

entbf(unsigned *s1,unsigned *t,con *start,pro p[],
mem m[],bus b[],unsigned clock)
{
    unsigned i,del;
    for(i=0;i<*ko;i++)
    {
        if(b[i].state==0)
            break;
    }
    if(i==*ko)
    {
        del=s_time(b) - *t; /* until that bus is freed */
        p[*s1].time+=del;
        p[*s1].n_e=2; /*entbf()*/
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
    else
    {
        del= BUST;
        p[*s1].time+=del;
        p[*s1].n_e=1; /*eobf*/
        *k1+=1;*n1+=1;
        *qbf-=1; /* this pe was in bus queue */
        *q-=1;
        b[i].state=1;
        b[i].p_no=*s1;
        p[*s1].b_no=i;
        b[i].time+=del;
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
}

/* If once required memory is found in busy state then
processor comes here */
entm(unsigned *s1,unsigned *t,con *start,pro p[],
mem m[],bus b[],unsigned clock)

```

```

{
  unsigned i,del;
  i=p[*s1].m_no;
  if(m[i].state==1)
  {
    del= m[i].time - p[*s1].time;
    /* until that memory is freed */
    p[*s1].time+=del;
    p[*s1].n_e=3;
    if(pr==1)
      printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
              *n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
  }
  else
  {
    m[i].state=1;
    m[i].p_no=*s1;
    p[*s1].m_no=i;
    p[*s1].n_e=4; /*prm*/
    del=MEMT; /*memory access time */
    p[*s1].time+=del;
    m[i].time+=del;
    m[i].q-=1;
    *q-=1;
    *m1+=1;
    if(pr==1)
      printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
              *n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
  }
}

/* When memory access ends processor comes here to
see if there is any bus free */
prm(unsigned *s1,unsigned *t,con *start,pro p[],
mem m[],bus b[],unsigned clock)
{
  unsigned i,j,del;
  j=p[*s1].m_no;
  *m1-=1; /* memory goes to idle state */
  for(i=0;i<*ko;i++)
  {
    if(b[i].state==0)
      break;
  }
  if(i==*ko)
  {
    del=s_time(b) - p[*s1].time; /* until a bus is freed */
    p[*s1].time+=del;
    *qbb+=1;
    *q+=1;
    p[*s1].n_e=5; /*entbb()*/
  }
}

```

```

        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
else
{
    del= BUST;
    p[*s1].time+=del;
    p[*s1].n_e=6; /* next event is eobb*/
    *k1+=1;
    *n1+=1;
    b[i].state=1;
    b[i].p_no=*s1;
    p[*s1].b_no=i;
    b[i].time+=del;
    if(pr==1)
        printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
            *n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
}
}
/* If in backward direction busy bus condition is
found then processor comes here */
entbb(unsigned *s1,unsigned *t,con *start,pro p[],
mem m[],bus b[],unsigned clock)
{
    unsigned i,j,del;
    for(i=0;i<*ko;i++)
    {
        if(b[i].state==0)
            break;
    }
    if(i==*ko)
    {
        del= s_time(b) - *t; /* until a bus is freed */
        p[*s1].time+=del;
        p[*s1].n_e=5; /*entbb()*/
        if(pr==1)
            printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
                *n1,*m1,*k1);
        *t+=del;
        insert(s1,t,start);
    }
else
{
    del= BUST;
    p[*s1].time+=del;
    p[*s1].n_e=6; /*eobb*/
    *k1+=1;*qbb--=1;
    *q--=1;
    *n1+=1;
    b[i].state=1;
}
}

```

```

b[i].p_no=*s1;
p[*s1].b_no=i;
b[i].time+=del;
if(pr==1)
    printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
           *n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
}
}

/* In backward direction after getting bus processor
comes in thinking state */
eobb(unsigned *s1,unsigned *t,con *start,pro p[],
mem m[],bus b[],unsigned clock)
{
    unsigned i,del;
    del=random(PT)+1;
    p[*s1].time+=del;
    p[*s1].n_e=0;
    *k1-=1;
    if(pr==1)
        printf("%5u %5u %5u %5u %5u %5u \n",*t,*s1,*q,
               *n1,*m1,*k1);
    *t+=del;
    insert(s1,t,start);
}

    /* subroutine insert has started below */

insert(unsigned *s1,unsigned *t,con *start)
{
    con *prev,*new,*current;
    current=start->next;
    if(*t < current->time)
    {
        start->p_no = *s1;
        start->time = *t;
    }
else
    {
        if((new=( con*)malloc(sizeof( con)))==NULL)
        {
            printf("No memory available for allocation \n");
            exit(1);
        }
        start->p_no=current->p_no;
        start->time=current->time;
        start->next=current->next;
        free(current);
        current=start->next;
        prev=start;
        while(current->next != NULL && *t >= current->time)
        {
            prev=current;

```

```

    current=current->next;
}
new->p_no=*s1;
new->time=*t;
if(current->next != NULL)
{
    prev->next=new;
    new->next=current;
}
else
{
    if(*t >= current->time)
    {
        current->next=new;
        new->next=NULL;
    }
    else
    {
        prev->next=new;
        new->next=current;
    }
}
}
/* Results and statistics are calculated in the
   routine result */
result(unsigned *tq,unsigned *tqf,unsigned *tqb,
        unsigned *tn1,unsigned *tm1, unsigned *tk1,
        unsigned *t)
{
    unsigned sub;
    if(rcount == 0)
    {
        t1 = *t;
        q_last = *q;
        q_flast= *qbf;
        q_blast= *qbb;
        n1_last = *n1;
        m1_last = *m1;
        k1_last = *k1;
        rcount+=1;
    }
    else
    {
        t2 = *t;
        sub=t2-t1;
        *tq+=q_last * sub;
        *tqf+=q_flast * sub;
        *tqb+=q_blast * sub;
        *tn1+=n1_last * sub;
        *tm1+=m1_last * sub;
        *tk1+=k1_last * sub;
        t1=t2;
        q_last = *q;
        q_flast = *qbf;
        q_blast = *qbb;
    }
}

```



```

    n1_last = *n1;
    m1_last = *m1;
    k1_last = *k1;
}
}
/* For equal priority assignment to all the processors */
eq_sort(con *tag,unsigned *e_count)
{
    con *current,*inter,*first;
    int i,store;
    first = tag;
    current = tag;
    inter=malloc(sizeof(con));
    for(i=*e_count;i>0;i--)
    {
        store=random(i);
        while(store)
        {
            store-=1;
            current=current->next;
        }
        inter->p_no=first->p_no;
        first->p_no=current->p_no;
        current->p_no=inter->p_no;
        first=first->next;
        current=first;
    }
    free(inter);
}
/* To place the processors in proper position in
   event queue */
sort(pro p[],con *start,unsigned *e_count,mem m[],bus b[])
{
    unsigned back,store,prm_cnt;
    con *current,*inter,*tag;
    back=0;prm_cnt=0;
    tag=start;
    current=start;
    store=*e_count;
    while(*e_count)
    {
        *e_count-=1;
        if((p[current->p_no].n_e==1 || p[current->p_no].n_e==4 ||
p[current->p_no].n_e==8 && back==0))
        {
            if(p[current->p_no].n_e==1 || p[current->p_no].n_e==8)
                b[p[current->p_no].b_no].state=0;
            else
                m[p[current->p_no].m_no].state=0;
            tag=tag->next;
            current=current->next;
            prm_cnt+=1;
        }
        else if((p[current->p_no].n_e==1 || p[current->p_no].n_e==4||
p[current->p_no].n_e==6) && back>0)

```

```

    {
        if(p[current->p_no].n_e==1 || p[current->p_no].n_e==6)
            b[p[current->p_no].b_no].state=0;
        else
            m[p[current->p_no].m_no].state=0;
        inter=(con*)malloc(sizeof(con));
        inter->p_no=tag->p_no;
        tag->p_no=current->p_no;
        current->p_no=inter->p_no;
        free(inter);
        prm_cnt+=1;
        tag=tag->next;
        current=current->next;
    }
    else
    {
        current=current->next;
        back+=1;
    }
}
current=start;
*e_count=store;
eq_sort(start,e_count);
*e_count=store;
}
/* Main program is started below */

main()
{
    FILE *fp;
    con *start,*prev,*new,*current;
    pro p[no];
    mem m[no];
    bus b[16];
    unsigned *tq,*tn1,*tm1,*tk1,clock,*tqb,*tqf;
    unsigned i,j,*s_c,*e_count,r[no],*sl,count,scr_cnt,*t;
    float ttq,ttn1,ttm1,ttk1,ttqb,ttqf,ttmq;
    void *calloc();
    void *malloc();
    if ((ko=(unsigned*)malloc(sizeof(unsigned)))==NULL)
    {
        printf("No memory");
        exit(1);
    }
    if((fp=fopen("fpa.dat","a"))==NULL)
    {
        printf("file error");
        exit(1);
    }
    *ko=0;
    pr=0;
    for(*ko =1;(*ko) < 17;(*ko)++ )
    {
        clrscr();
        rcount=0; t1=0;t2=0; q_flast=0; q_blast=0;q_last=0;

```

```

clock=5;scr_cnt=0;
n1_last=0; m1_last=0; k1_last=0;
qbb=malloc(sizeof(unsigned));
qbf=malloc(sizeof(unsigned));
tqb=malloc(sizeof(unsigned));
tqf=malloc(sizeof(unsigned));
ttot=(unsigned*)malloc(sizeof(unsigned));
n1=(unsigned*)malloc(sizeof(unsigned));
m1=(unsigned*)malloc(sizeof(unsigned));
k1=(unsigned*)malloc(sizeof(unsigned));
q=(unsigned*)malloc(sizeof(unsigned));
t=(unsigned*)malloc(sizeof(unsigned));
s1=(unsigned*)malloc(sizeof(unsigned));
tq=(unsigned*)malloc(sizeof(unsigned));
tn1=(unsigned*)malloc(sizeof(unsigned));
tm1=(unsigned*)malloc(sizeof(unsigned));
tk1=(unsigned*)malloc(sizeof(unsigned));
e_count=(unsigned*)malloc(sizeof(unsigned));
s_c=(unsigned*)malloc(sizeof(unsigned));
*n1=no;*m1=0;*k1=0;*qbb=0;*qbf=0;*q=0;*ttot=30000;
*tq=0;*tqf=0;*tqb=0;*tn1=0;*tm1=0;*tk1=0;
if((start=(con*)malloc(sizeof(con)))==NULL)
{
printf("No memory available for allocation \n");
exit(1);
}

/* Random numbers are generated below */
randomize();
for(i=0;i < no;i++)
r[i]=random(PT)+1;
/* Controllar linked list is formed below */
start->next=NULL;
start->p_no=0;
start->time=r[0];
for(i=1;i<no;i++)
{
if((current=(con*)malloc(sizeof(con)))==NULL)
{
printf("No memory");
exit(1);
}
current->time=r[i];
current->p_no=i;
if(current->time < start->time)
{
prev=start;
current->next=prev;
start=current;
}
else
{
prev=start;
new=start->next;
while(new != NULL && current->time >= prev->time)

```

```

{
if(current->time < new->time)
{
current->next=new;
prev->next=current;
prev=new;
}
else
{
prev=new;
new=new->next;
}
}
if(new == NULL)
{
prev->next=current;
current->next=NULL;
}
}
}
/*initialization of arrays are done below */
for(i=0;i< no;i++)
{
p[i].n_e=0;
p[i].m_no=0;
p[i].b_no=0;
p[i].time=r[i];
}
for(i=0;i<no;i++)
{
m[i].state=0;
m[i].p_no=0;
m[i].time=0;
m[i].q=0;
}

for(i=0;i < (*ko);i++)
{
b[i].state=0;
b[i].p_no=0;
b[i].time=0;
}
if(pr==1)
{
printf("    Time    PE    BUSY    In    In    \n");
printf("                no    PE    MEM    BUS    \n");
gotoxy(2,24);
printf("PRESS ANY KEY TO SEE NEXT PAGE");
/*Finding of current event\events are started below */
}
window(1,3,79,20);
scr_cnt=0;
clrscr();
gotoxy(1,1);
*t=start->time;

```

```

for(i=0;i<mo;i++)
    m[i].time = *t;
for(i=0;i < *ko;i++)
    b[i].time = *t;
while(*t < *ttot)
{
    *e_count=1;
    current=start;
    new=current->next;
    while(current->time == new->time && new != NULL)
    {
        *e_count+=1;
        current=new;
        new=current->next;
    }
    sort(p,start,e_count,m,b);
    while(*e_count)
    {
        *e_count-=1;
        *s1=start->p_no;
        *s_c=p[*s1].n_e;
        switch(*s_c)
        {
            case 0:
            {
                eop(s1,t,start,p,m,b,clock);
                break;
            }
            case 1:
            {
                eobf(s1,t,start,p,m,b,clock);
                break;
            }
            case 2:
            {
                entbf(s1,t,start,p,m,b,clock);
                break;
            }
            case 3:
            {
                entm(s1,t,start,p,m,b,clock);
                break;
            }
            case 4:
            {
                prm(s1,t,start,p,m,b,clock);
                break;
            }
            case 5:
            {
                entbb(s1,t,start,p,m,b,clock);
                break;
            }
        }
        case 6:
        {

```

```

    eobb(s1,t,start,p,m,b,clock);
    break;
}
}
*t=start->time;
if(*e_count == 0)
{
    if(*t>1000)
        result(tq,tqf,tqb,tn1,tml,tk1,t);
    ud_m(m,t);
    ud_b(b,t);
}
scr_cnt+=1;
if(pr==1)
{
    if(scr_cnt == 17)
    {
        getch();
        scr_cnt=0;
        clrscr();
        gotoxy(1,1);
    }
}
}
}
clrscr();
*ttot-=1000;
ttq=((float)(*tq))/((float)(*ttot));
ttn1=((float)(*tn1))/((float)(*ttot));
ttml=((float)(*tml))/((float)(*ttot));
ttk1=((float)(*tk1))/((float)(*ttot));
ttqb=((float)(*tqb))/((float)(*ttot));
ttqf=((float)(*tqf))/((float)(*ttot));
*ttot+=1000;
printf("\n\tAverage Queue Length=%f\n",ttq );
printf("\n\tAverage No of busy Processor =%f\n",ttn1);
printf("\n\tAverage No of Busy Memory =%f\n",ttml);
printf("\n\tAverage No of Busy Bus =%f\n",ttk1);
printf("\n\tProcessor Utilization =%f\n",ttn1/no);
printf("\n\tMemory Bandwidth=%f\n",ttml/mo);
printf("\n\tBus Utilization=%f\n",ttk1/*ko));
getch();
fprintf(fp,"%u      %f      %f      %f      %f\n",
*ko,ttq,ttn1/no,ttml,ttk1/*ko));
}
fclose(fp);
}

```

## REFERENCES

- [1] M. Auguin and F. Boeri, "Presentation of the LASSY'S Work in the field of Parallel Architecture" , Supercomputing, Ed. A. Lichniewsky, C. Saguez, North-Holland, 1987.
- [2] Kai. Hwang and Faye A. Briggs, " Computer Architecture and Parallel Processing" , McGraw-Hill Book Company, 1985.
- [3] G. Jack Lipovski and Miroslaw Malek, " Parallel Computing Theory and Comparisons", John Wiley & Sons, 1987.
- [4] John P. Hayes, "Computer Architecture and Organisation ", McGraw-Hill Book Company, 1978.
- [5] L.N.Bhuyan, Q. Yang and D.P. Agrawal, " Performance of Multiprocessor Interconnection Networks", IEEE Computer, February 1989, pp. 25-37 .
- [6] M.H.Chowdhury and Md.S.Alam, "Shared Bus Multiprocessor," AIT-BUET Joint Conference Proceedings", 25th August, 1990, pp. 42-65.
- [7] Andrew S. Tanenbaum, "Computer Networks", Prentice Hall of India Private Limited, New Delhi 1987.
- [8] T.N.Mudge and H.B. Al-Sadoun, " A Semi-Markov Model for the Performance of Multiple Bus Systems", IEEE Trans. on Comput., Oct. 1985, pp. 934-942.
- [9] T.N. Mudge, J.P. Hayes and D.C. Winsor, "Multiple Bus Architectures", IEEE Computer, June 1987, pp. 42-48 .

- [10] D. M. Taub, "Arbitration and Control Acquisition in the proposed IEEE 896 Futurebus", IEEE Micro, August 1984, pp. 28-41.
- [11] F. El Guibaly, "Design and Analysis of Arbitration Protocols", IEEE Trans. on Comput., vol.C38, no.2, Feb. , 1989. pp. 161-171.
- [12] S.M. Mahmud and M.S.U. Alam, " A new arbitration circuit for synchronous multiple bus multiprocessor system", Electrical and Computer Engineering Department, Wayne State University. U.S.A .
- [13] J.H.Patel,"Analysis of Multiprocessors with private cache memories",IEEE Trans. on Comput.vol C31, April 1982, pp. 296-304.
- [14] A.Fukuda, "Equilibrium Point Analysis of Memory Interference in Multiprocessor System " IEEE Trans. on comput., Vol. C37, No. 5, May 1988.
- [15] J.K. Fisher, D.D.Gajski, M.Y.Wu, "Programming Environments for Multiprocessors", Supercomputing, Ed. A.Lichnewsky, C. Saguez, North-Holland, 1987.
- [16] M.A.Marson and M.Gerla,"Markov Model for Multiple Bus Multiprocessor System",IEEE Trans. on Comput. vol C-31, March 1982, pp. 239-248.



- [17] L.N. Bhuyan, "An Analysis of Processor Memory Interconnection Network", IEEE Trans. on Comput., Vol. C-34, No. 3, Mar. 1985, pp. 279-283.
- [18] N. Roberts, D.F. Anderson, R.M. Deal, M.S. Garet, W.A. Shaefer, "Introduction To Computer Simulation: The System Dynamics Approach", Addison-Wesley Publishing Company, 1983
- [19] A.A.B. Pritsker, C.D. Pegden, " Introduction to Simulation and SLAM", A Halsted Press Book, John Wiley & Sons, New York, 1979.

