UNIT 1 MULTIPROCESSOR SYSTEMS

Structure **Page Nos.**

1.0 INTRODUCTION

A multiprocessor system is a collection of a number of standard processors put together in an innovative way to improve the performance / speed of computer hardware. The main feature of this architecture is to provide high speed at low cost in comparison to uniprocessor. In a distributed system, the high cost of multiprocessor can be offset by employing them on a computationally intensive task by making it compute server. The multiprocessor system is generally characterised by - *increased system throughput* and *application speedup* - parallel processing.

Throughput can be improved, in a *time-sharing environment*, by executing a number of unrelated user processor on different processors in parallel. As a result a large number of different tasks can be completed in a unit of time without explicit user direction. On the other hand application speedup is possible by creating a multiple processor scheduled to work on different processors.

The scheduling can be done in two ways:

1) *Automatic means*, by parallelising compiler.

2) *Explicit-tasking approach*, where each programme submitted for execution is treated by the operating system as an independent process.

Multiprocessor operating systems aim to support high performance through multiple CPUs. An important goal is to make the number of CPUs transparent to the application. Achieving such transparency is relatively easy because the communication between different (parts of) applications uses the same primitives as those in multitasking uni-processor operating systems. The idea is that all communication is done by manipulating data at shared memory locations, and that we only have to protect that data against simultaneous access. Protection is done through synchronization primitives like semaphores and monitors.

In this unit we will study multiprocessor coupling, interconnections, types of multiprocessor operating systems and synchronization.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- define a multiprocessor system;
- describe the architecture of multiprocessor and distinguish among various types of architecture;
- become familiar with different types of multiprocessor operating systems;
- discuss the functions of multiprocessors operating systems;
- describe the requirement of multiprocessor in Operating System, and
- discuss the synchronization process in a multiprocessor system.

1.2 MULTIPROCESSING AND PROCESSOR COUPLING

Multiprocessing is a general term for the use of two or more CPUs within a single computer system. There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined. The term multiprocessing is sometimes used to refer to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant. However, the term multiprogramming is more appropriate to describe this concept, which is implemented mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs. A system can be both multiprocessing and multiprogramming, only one of the two, or neither of the two. Processor Coupling is a type of multiprocessing. Let us see in the next section.

Processor Coupling

Tightly-coupled multiprocessor systems contain multiple CPUs that are connected at the bus level. These CPUs may have access to a central shared memory (SMP), or may participate in a memory hierarchy with both local and shared memory (NUMA). *The IBM p690 Regatta* is an example of a high end SMP system. Chip multiprocessors, also known as multi-core computing, involve more than one processor placed on a single chip and can be thought of the most extreme form of tightly-coupled multiprocessing. Mainframe systems with multiple processors are often tightly-coupled.

Loosely-coupled multiprocessor systems often referred to as clusters are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system. *A Linux Beowulf* is an example of a loosely-coupled system.

Tightly-coupled systems perform better and are physically smaller than loosely-coupled systems, but have historically required greater initial investments and may depreciate rapidly; nodes in a loosely-coupled system are usually inexpensive commodity computers and can be recycled as independent machines upon retirement from the cluster.

Power consumption is also a consideration. Tightly-coupled systems tend to be much more energy efficient than clusters. This is due to fact that considerable economies can be realised by designing components to work together from the beginning in tightly-coupled systems, whereas loosely-coupled systems use components that were not necessarily intended specifically for use in such systems.

1.3 MULTIPROCESSOR INTERCONNECTIONS

As learnt above, a multiprocessor can speed-up and can improve throughput of the computer system architecturally. The whole architecture of multiprocessor is based on the principle of parallel processing, which needs to synchronize, after completing a

stage of computation, to exchange data. For this the multiprocessor system needs an efficient mechanism to communicate. This section outlines the different architecture of multiprocessor interconnection, including:

- Bus-oriented System
- Crossbar-connected System
- Hyper cubes
- Multistage Switch-based System.

1.3.1 Bus-oriented System

Figure1 illustrates the typical architecture of a bus oriented system. As indicated, processors and memory are connected by a common bus. Communication between processors (P1, P2, P3 and P4) and with globally shared memory is possible over a shared bus. Other then illustrated many different schemes of bus-oriented system are also possible, such as:

1) Individual processors may or may not have private/cache memory.
2) Individual processors may or may not attach with input/output devices.
3) Input/output devices may be attached to shared bus.
4) Shared memory implemented in the form of multiple physical banks connected to the shared bus.

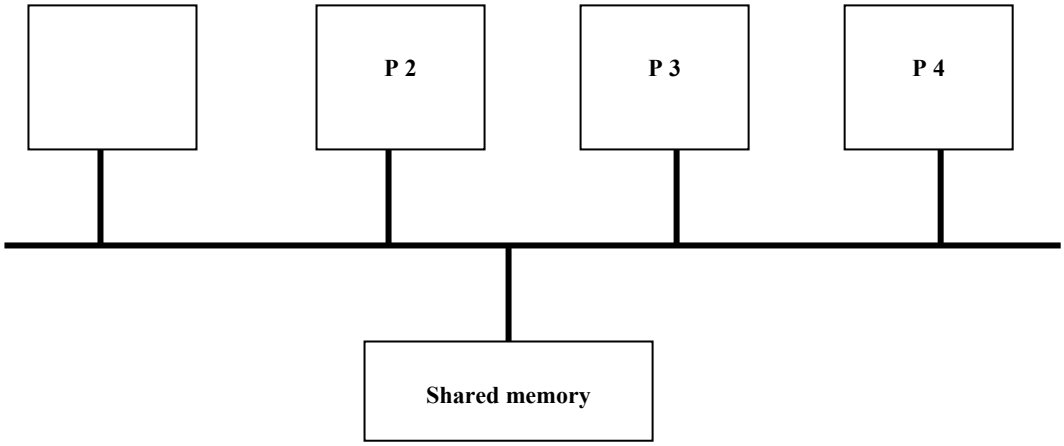

Figure 1. Bus-oriented multiprocessor interconnection

The above architecture gives rise to a problem of *contention* at two points, one is shared bus itself and the other is shared memory. Employing private/cache memory in either of two ways, explained below, the problem of contention could be reduced;

- with shared memory; and
- with cache associated with each individual processor

1) **With shared memory**

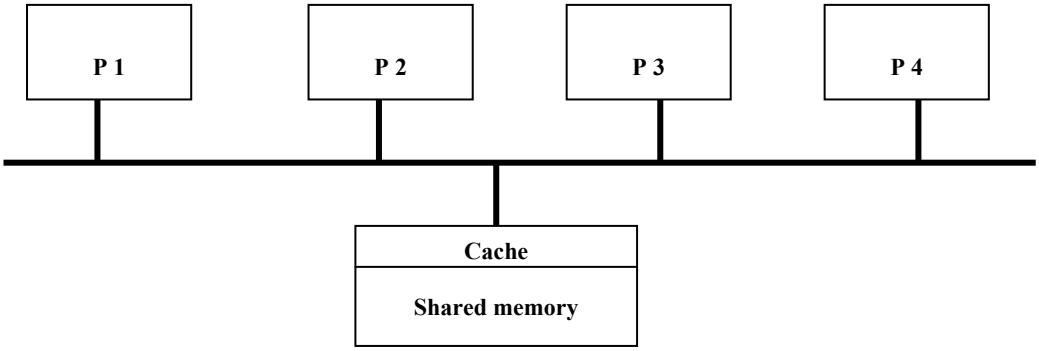

Figure 2: With shared Memory

2) **Cache associated with each individual processor.**

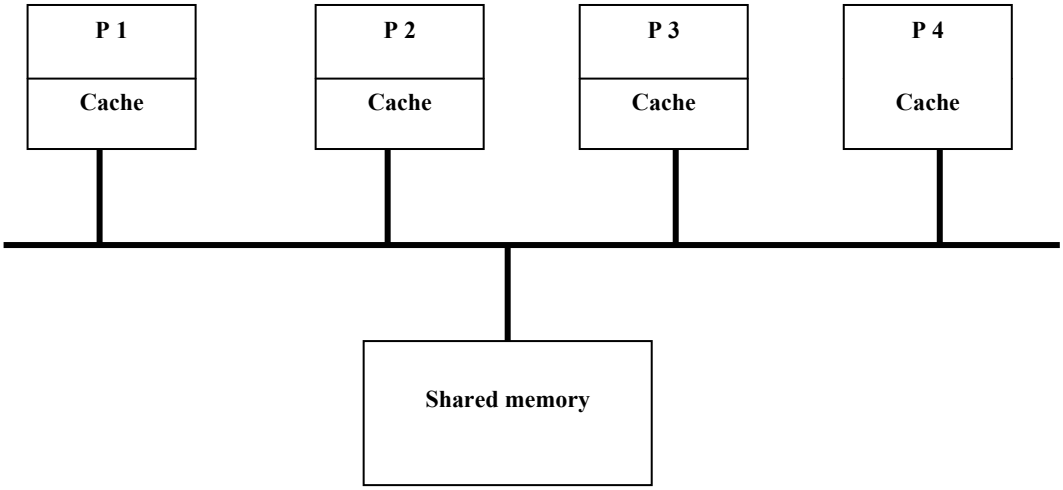

Figure 3: Individual processors with cache

The second approach where the cache is associated with each individual processor is the most popular approach because it reduces contention more effectively. Cache attached with processor an capture many of the local memory references for example, with a cache of 90% hit ratio, a processor on average needs to access the shared memory for 1 (one) out of 10 (ten) memory references because 9 (nine) references are already captured by private memory of processor. In this case where memory access is uniformly distributed a 90% cache hit ratio can support the shared bus to speed-up 10 times more than the process without cache. The negative aspect of such an arrangement arises when in the presence of multiple cache the shared writable data are cached. In this case Cache Coherence is maintained to consistency between multiple physical copies of a single logical datum with each other in the presence of update activity. Yes, the cache coherence can be maintained by attaching additional hardware or by including some specialised protocols designed for the same but unfortunately this special arrangement will increase the bus traffic and thus reduce the benefit that processor caches are designed to provide.

Cache coherence refers to the integrity of data stored in local caches of a shared resource. Cache coherence is a special case of memory coherence.

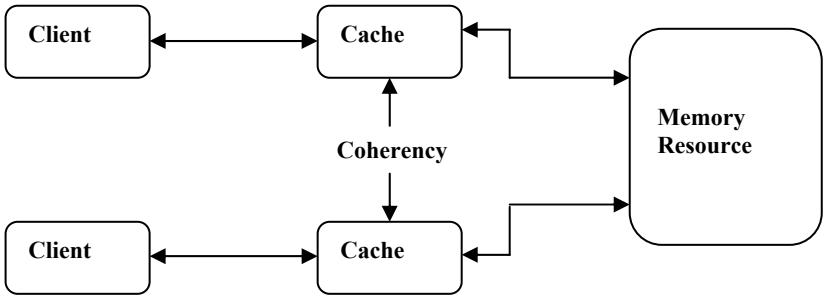

Figure 4: Multiple Caches of Common Resource

When clients in a system, particularly CPUs in a multiprocessing system, maintain caches of a common memory resource, problems arise. Referring to the *Figure 4*, if the top client has a copy of a memory block from a previous read and the bottom client changes that memory block, the top client could be left with an invalid cache of memory without it knowing any better. Cache coherence is intended to manage such conflicts and maintain consistency between cache and memory.

Let us discuss some techniques which can be employed to decrease the impact of bus and memory saturation in bus-oriented system.

1) **Wider Bus Technique**: As suggested by its name a bus is made wider so that more bytes can be transferred in a single bus cycle. In other words, a wider

parallel bus increases the bandwidth by transferring more bytes in a single bus cycle. The need of bus transaction arises when lost or missed blocks are to fetch into his processor cache. In this transaction many system supports, block-level reads and writes of main memory. In the similar way, a missing block can be transferred from the main memory to his processor cache only by a single main-memory (block) read action. The advantage of *block level access to memory* over *word-oriented busses* is the amortization of overhead addressing, acquisition and arbitration over a large number of items in a block. Moreover, it also enjoyed specialised burst bus cycles, which makes their transport more efficient.

2) **Split Request/Reply Protocols**: The memory request and reply are split into two individual works and are treated as separate bus transactions. As soon as a processor requests a block, the bus released to other user, meanwhile it takes for memory to fetch and assemble the related group of items.

The bus-oriented system is the simplest architecture of multiprocessor system. In this way it is believed that in a tightly coupled system this organisation can support on the order of 10 processors. However, the two contention points (the bus and the shared memory) limit the scalability of the system.

1.3.2 Crossbar-Connected System

Crossbar-connected System (illustrated in *Figure 5*) is a grid structure of processor and memory modules. The every cross point of grid structure is attached with switch. By looking at the *Figure* it seems to be a contention free architecture of multiprocessing system, it shows simultaneous access between processor and memory modules as *N* number of processors are provided with *N* number of memory modules. Thus each processor accesses a different memory module. Crossbar needs N^2 switches for fully connected network between processors and memory. Processors may or may not have their private memories. In the later case the system supports uniform-memory-access.

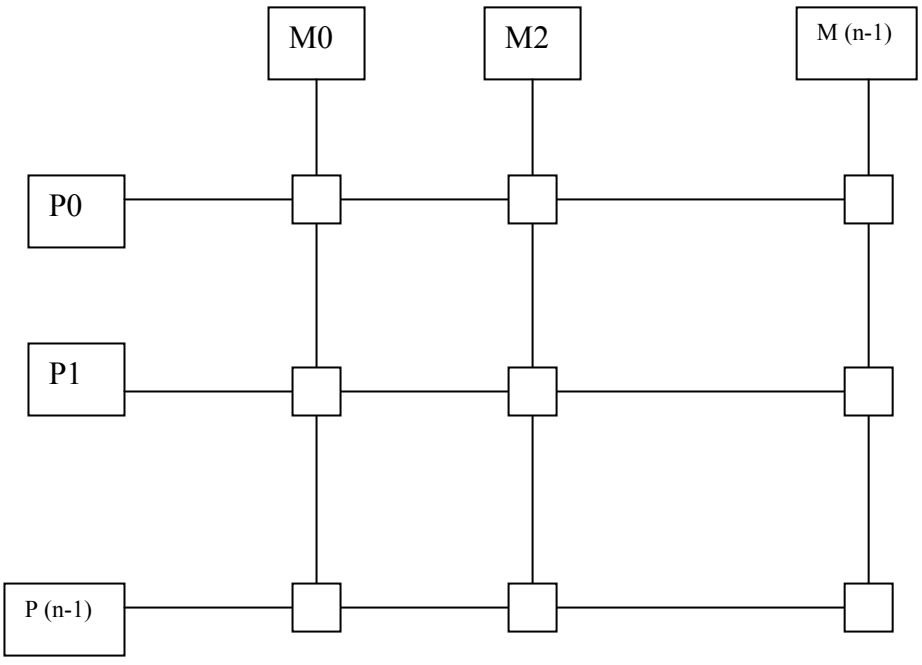

Figure 5: Crossbar-connected system

Where P= processor, M=Memory Module and =Switch

But Unfortunately this system also faces the problem of contention when,

1) More than two processors (P1 and P2) attempt to access the one memory module (M1) at the same time. In this condition, contention can be avoided by making any one processor (P1) deferred until the other one (P2) finishes this work or left the same memory module (M1) free for processor P1.

2) More than two processor attempts to access the same memory module. This problem cannot be solved by above-mentioned solution.

Thus in crossbar connection system the problem of contention occurs on at memory neither at processor nor at interconnection networks level. The solution of this problem includes quadratic growth of its switches as well as its complexity level. Not only this, it will make the whole system expensive and also limit the scalability of the system.

1.3.3 Hypercubes System

Hypercube base architecture is not an old concept of multiprocessor system. This architecture has some advantages over other architectures of multiprocessing system. As the *Figure 6* indicates, the system topology can support large number of processors by providing increasing interconnections with increasing complexity. In an *n-degree* hypercube architecture, we have:

1) 2^n nodes (Total number of processors)
2) Nodes are arranged in n-dimensional cube, i.e. each node is connected to *n* number of nodes.
3) Each node is assigned with a unique address which lies between 0 to2^n –1
4) The adjacent nodes (n−1) are differing in 1 bit and the *nth* node is having maximum '*n*' internode distance.

Let us take an example of **3-degree hypercube** to understand the above structure:

1) 3-degree hypercube will have 2^n nodes i.e., $2^3 = 8$ nodes
2) Nodes are arranged in 3-dimensional cube, that is, each node is connected to 3 number of nodes.
3) Each node is assigned with a unique address, which lies between 0 to 7 (2^n –1), i.e., 000, 001, 010, 011, 100, 101, 110, 111
4) Two adjacent nodes differing in 1 bit (001, 010) and the 3rd (n^{th}) node is having maximum '3'internode distance (100).

Hypercube provide a good basis for scalable system because its communication length grows logarithmically with the number of nodes. It provides a bi-directional communication between two processors. It is usually used in loosely coupled multiprocessor system because the transfer of data between two processors goes through several intermediate processors. The longest internodes delay is *n-degree*. To increase the input/output bandwidth the input/output devices can be attached with every node (processor).

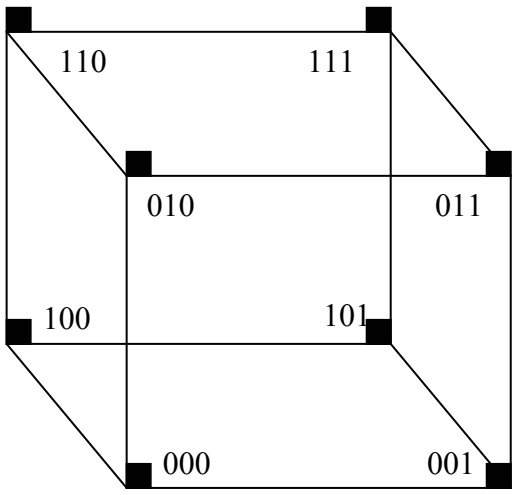

Figure 6: Hypercubes System

1.3.4 Multistage Switch Based system

Multistage Switch Based System permits simultaneous connection between several input-output pairs. It consists of several stages of switches which provide multistage interconnection network. A *N* input-output connections contains K= $\log_2$N stages of $^{N}/_{2}$ switches at each stage. In simple words, *N*N* processor-memory interconnection network requires ($^{N}/_{2}$) x = $\log_2$N switches.

For example, in a 8×8 process-memory interconnection network requires ($8/_2$* $\log_2 8$) = 4*3= 12 switches. Each switch acts as 2×2 crossbar.

This network can connect any processor to any memory module by making appropriate connection of each of the '*K*' stages. The binary address of processor and memory gives binary string routing path between module pairs of input and output. the routing path *id* decided by on the basis of destination binary address, that the sender includes with each request for connection. Various combinations of paths between sources to destination are possible by using different switch function (straight, swap, copy, etc.)

In multistage switch based system all inputs are connected to all outputs in such a way that no two-processor attempt to access the same memory at the same time. But the problem of contention, at a switch, arises when some memory modules are contested by some fixed processor. In this situation only one request is allowed to access and rest of the requests are dropped. The processor whose requests were dropped can retry the request or if buffers are attached with each switch the rejected request is forwarded by buffer automatically for transmission. This Multistage interconnection networks also called store-and-forward networks.

1.4 TYPES OF MULTIPROCESSOR OPERATING SYSTEMS

The multiprocessor operating systems are complex in comparison to multiprograms on an uniprocessor operating system because multiprocessor executes tasks concurrently.

Therefore, it must be able to support the concurrent execution of multiple tasks to increase processors performance.

Depending upon the control structure and its organisation the three basic types of multiprocessor operating system are:

1) Separate supervisor
2) Master-slave
3) Symmetric Supervision

1.4.1 Separate Supervisors

In separate supervisor system each process behaves independently. Each system has its own operating system which manages local input/output devices, file system and memory well as keeps its own copy of kernel, supervisor and data structures, whereas some common data structures also exist for communication between processors. The access protection is maintained, between processor, by using some synchronization mechanism like semaphores. Such architecture will face the following problems:

1) Little coupling among processors.
2) Parallel execution of single task.
3) During process failure it degrades.
4) Inefficient configuration as the problem of replication arises between supervisor/kernel/data structure code and each processor.

1.4.2 Master-Slave

In master-slave, out of many processors one processor behaves as a master whereas others behave as slaves. The master processor is dedicated to executing the operating system. It works as scheduler and controller over slave processors. It schedules the work and also controls the activity of the slaves. Therefore, usually data structures are stored in its private memory. Slave processors are often identified and work only as a schedulable pool of resources, in other words, the slave processors execute application programmes.

This arrangement allows the parallel execution of a single task by allocating several subtasks to multiple processors concurrently. Since the operating system is executed by only master processors this system is relatively simple to develop and efficient to use. Limited scalability is the main limitation of this system, because the master processor become a bottleneck and will consequently fail to fully utilise slave processors.

1.4.3 Symmetric

In symmetric organisation all processors configuration are identical. All processors are autonomous and are treated equally. To make all the processors functionally identical, all the resources are pooled and are available to them. This operating system is also symmetric as any processor may execute it. In other words there is one copy of kernel that can be executed by all processors concurrently. To that end, the whole process is needed to be controlled for proper interlocks for accessing scarce data structure and pooled resources.

The simplest way to achieve this is to treat the entire operating system as a critical section and allow only one processor to execute the operating system at one time. This method is called ‘floating master’ method because in spite of the presence of many processors only one operating system exists. The processor that executes the operating system has a special role and acts as a master. As the operating system is not bound to any specific processor, therefore, it floats from one processor to another.
Parallel execution of different applications is achieved by maintaining a queue of ready processors in shared memory. Processor allocation is then reduced to assigning the first ready process to first available processor until either all processors are busy or the queue is emptied. Therefore, each idled processor fetches the next work item from the queue.

1.5 MULTIPROCESSOR OPERATING SYSTEM FUNCTIONS AND REQUIREMENTS

A multiprocessor operating system manages all the available resources and schedule functionality to form an abstraction it will facilitates programme execution and interaction with users.

A processor is one of the important and basic types of resources that need to be manage. For effective use of multiprocessors the processor scheduling is necessary. Processors scheduling undertakes the following tasks:

(i) Allocation of processors among applications in such a manner that will be consistent with system design objectives. It affects the system throughput. Throughput can be improved by co-scheduling several applications together, thus availing fewer processors to each.

(ii) Ensure efficient use of processors allocation to an application. This primarily affects the speedup of the system.

The above two tasks are somehow conflicting each other because maximum speedup needs dedication of a large proportion of a systems processors to a single application

which will decrease throughput of the system. Due to the difficulties of automating the process, the need for explicit programmer direction arises to some degree. Generally the language translators and preprocessors provide support for explicit and automatic parallelism. The two primary facets of OS support for multiprocessing are:

(i) Flexible and efficient interprocess and interprocessor synchronization mechanism, and

(ii) Efficient creation and management of a large number of threads of activity, such as processes or threads.

The latter aspect is important because parallelism is often accomplished by splitting an application into separate, individually executable subtasks that may be allocated to different processors.

The **Memory management** is the second basic type of resource that needs to be managed. In multiprocessors system memory management is highly dependent on the architecture and inter-connection scheme.

- In loosely coupled systems memory is usually handled independently on a pre-processor basis whereas in multiprocessor system shared memory may be simulated by means of a message passing mechanism.
- In shared memory systems the operating system should provide a flexible memory model that facilitates safe and efficient access to share data structures and synchronization variables.

A multiprocessor operating system should provide a hardware independent, unified model of shared memory to facilitate porting of applications between different multiprocessor environments. The designers of the mach operating system exploited the duality of memory management and inter-process communication.

The third basic resource is **Device Management** but it has received little attention in multiprocessor systems to date, because earlier the main focus point is speedup of compute intensive application, which generally do not generate much input/output after the initial loading. Now, multiprocessors are applied for more balanced general-purpose applications, therefore, the input/output requirement increases in proportion with the realised throughput and speed.

1.6 MULTIPROCESSOR SYNCHRONIZATION

Multiprocessor system facilitates parallel program execution and read/write sharing of data and thus may cause the processors to concurrently access location in the shared memory. Therefore, a correct and reliable mechanism is needed to serialize this access. This is called synchronization mechanism. The mechanism should make access to a shared data structure appear atomic with respect to each other. In this section, we introduce some new mechanism and techniques suitable for synchronization in multiprocessors.

1.6.1 Test-and-Set

The test-and-set instruction automatically reads and modifies the contents of a memory location in one memory cycle. It is as follows:

```
function test-and-set (var m: Boolean); Boolean;
begin
        test-and set:=m;
        m:=rue
end;
```

The test-and-set instruction returns the current value of variable m (memory location) and sets it to true. This instruction can be used to implement *P* and *V* operations (Primitives) on a binary semaphore, *S*, in the following way (*S* is implemented as a memory location):

P(S): while Test-and-Set (S) do nothing;
V(S): S:=false;
Initially, *S* is set to false. When a *P(S)* operation is executed for the first time, test-and-set*(S)* returns a false value (and sets *S* to true) and the "while" loop of the *P(S)* operation terminates. All subsequent executions of *P(S)* keep looping because *S* is true until a *V(S)* operation is executed.

1.6.2 Compare-and-Swap

The compare and swap instruction is used in the optimistic synchronization of concurrent updates to a memory location. This instruction is defined as follows (*r1and r2* are to registers of a processor and *m* is a memory location):

function test-and-set *(var m: Boolean); Boolean;*
var temp: integer;
begin
    temp:=m;
    if temp = r1 then {m:= r2;z:=1}
    else [r1:= temp; z:=0}
end;

If the contents of *r1* and *m* are identical, this instruction assigns the contents of *r2* to *m* and sets *z* to *1*. Otherwise, it assigns the contents of *m* to *r1* and set *z* to *0*. Variable *z* is a flag that indicates the success of the execution. This instruction can be used to synchronize concurrent access to a shared variable.

1.6.3 Fetch-and-Add

The fetch and add instruction is a multiple operation memory access instruction that automatically adds a constant to a memory location and returns the previous contents of the memory location. This instruction is defined as follows:

Function Fetch-and-add (m: integer; c: integer);
Var temp: integer;
Begin
    Temp:= m;
    M:= m + c;
    Return (temp)
end;

This instruction is executed by the hardware placed in the interconnection network not by the hardware present in the memory modules. When several processors concurrently execute a fetch-and-add instruction on the same memory location, these instructions are combined in the network and are executed by the network in the following way:

- A single increment, which is the sum of the increments of all these instructions, is added to the memory location.
- A single value is returned by the network to each of the processors, which is an arbitrary serialisation of the execution of the individual instructions.
- If a number of processors simultaneously perform fetch-and-add instructions on the same memory location, the net result is as if these instructions were executed serially in some unpredictable order.

The fetch-and-add instruction is powerful and it allows the implementation of P and V operations on a general semaphore, S, in the following manner:

P(S): *while (Fetch-add-Add(S, -1)< 0) do*
begin
    Fetch-and-Add (S, 1);
    while (S<0) do nothing;
end;

The outer "while-do" statement ensures that only one processor succeeds in decrementing S to 0 when multiple processors try to decrement variable S. All the unsuccessful processors add 1 back to S and try again to decrement it. The "while-do" statement forces an unsuccessful processor to wait (before retrying) until S is greater then 0.

V(S): *Fetch-and-Add (S, 1).*

☞ Check Your Progress 1

1) What is the difference between a loosely coupled system and a tightly coupled system? Give examples.

……

……

……

……

2) What is the difference between symmetric and asymmetric multiprocessing?

……

……

……

……

1.7 SUMMARY

Multiprocessors systems architecture provides higher computing power and speed. It consists of multiple processors that putting power and speed to the system. This system can execute multiple tasks on different processors concurrently. Similarly, it can also execute a single task in parallel on different processors. The design of interconnection networks includes the bus, the cross-bar switch and the multistage interconnection networks. To support parallel execution this system must effectively schedule tasks to various processors. And also it must support primitives for process synchronization and virtual memory management. The three basic configurations of multiprocessor operating systems are: Separate supervisors, Master/slave and Symmetric. A multiprocessor operating system manages all the available resources and schedule functionality to form an abstraction. It includes Process scheduling, Memory management and Device management. Different mechanism and techniques are used for synchronization in multiprocessors to serialize the access of pooled resources. In this section we have discussed Test-and-Set, Compare-and-Swap and Fetch-and-Add techniques of synchronization.

1.8 SOLUTIONS/ANSWERS

1) One feature that is commonly characterizing tightly coupled systems is that they share the clock. Therefore, multiprocessors are typically tightly coupled but distributed workstations on a network are not.

   Another difference is that: in a tightly-coupled system, the delay experienced when a message is sent from one computer to another is short, and data rate is high; that is, the number of bits per second that can be transferred is large. In a loosely-coupled system, the opposite is true: the intermachine message delay is large and the data rate is low.

   For example, two CPU chips on the same printed circuit board and connected by wires etched onto the board are likely to be tightly coupled, whereas two

computers connected by a 2400 bit/sec modem over the telephone system are certain to be loosely coupled.

2) The difference between symmetric and asymmetric multiprocessing: all processors of symmetric multiprocessing are peers; the relationship between processors of asymmetric multiprocessing is a master-slave relationship. More specifically, each CPU in symmetric multiprocessing runs the same copy of the OS, while in asymmetric multiprocessing, they split responsibilities typically, therefore, each may have specialised (different) software and roles.

1.9 FURTHER READINGS

1) Singhal Mukesh and Shivaratri G. Niranjan, *Advanced Concepts in Operating Systems*, TMGH, 2003, New Delhi.

2) Hwang, K. and F Briggs, *Multiprocessors Systems Architectures*, McGraw-Hill, New York.

3) Milenkovic, M., *Operating Systems: Concepts and Design*, TMGH, New Delhi.

4) Silberschatz, A., J. Peterson, and D. Gavin, *Operating Systems Concepts*, 3rd ed., Addison Wesley, New Delhi.