CHAPTER
SEVEN

MULTIPROCESSOR ARCHITECTURE AND PROGRAMMING

This chapter covers multiprocessor system architectures and multiprocessing requirements. Hardware-software factors which limit the performance are presented. The interconnection topologies between the processors and the main memory are discussed. Models are shown to evaluate the performance of these interconnection structures. We present some memory configurations and a classification of operating systems for multiprocessors. Language and programming issues in using multiprocessor systems are discussed at the end.

7.1 FUNCTIONAL STRUCTURES

In this section, we introduce the functional structures of multiprocessor systems. Example systems discussed below include the Cm*, Cyber-170, Honeywell 60/66, and the PDP-10 multiprocessor configurations. More example systems will be given in Chapter 9. Functional characteristics of processor architecture for multiprocessors are presented.

Multiprocessors can be grossly characterized by two attributes: first, a multiprocessor is a single computer that includes multiple processors and second, processors may communicate and cooperate 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.

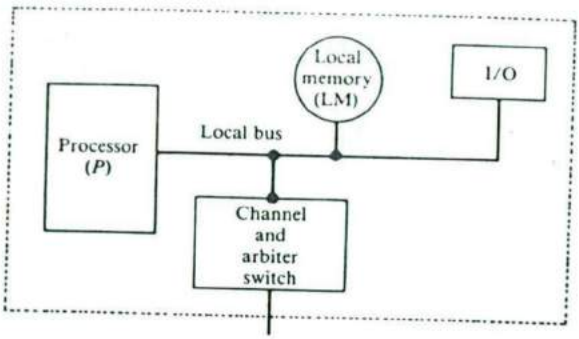
There are some similarities between multiprocessors and multicomputer systems since both are motivated by the same basic goal—the support of concurrent operations in the system. However, there exists an important distinction between multiple computer systems and multiprocessors, based on the extent of resource sharing and cooperation in the solution of a problem. A *multiple computer system* consists of several autonomous computers which may or may not communicate with each other. An example of a multiple computer system is the IBM Attached Support Processor System. A *multiprocessor system* is controlled by one operating system which provides interaction between processors and their programs at the process, data set, and data element levels. An example is the Denelcor's HEP system.

Two different sets of architectural models for a multiprocessor are described below. One is a *tightly coupled* multiprocessor and the other is a *loosely coupled* multiprocessor. Tightly coupled multiprocessors communicate through a shared main memory. Hence the rate at which data can communicate from one processor to the other is on the order of the bandwidth of the memory. A small local memory or high-speed buffer (cache) may exist in each processor. A complete connectivity exists between the processors and memory. This connectivity can be accomplished either by inserting an interconnection network between the processors and the memory 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 attempt to access the same memory unit concurrently. In Chapter 2, we have seen some configurations of interleaved main memory suitable for multiprocessors. The degree of conflicts can be reduced by increasing the degree of interleaving. However, this must be coupled with careful data assignments to the memory modules. Another limiting factor is the processor-memory interconnection network itself. This will be discussed in more detail later.

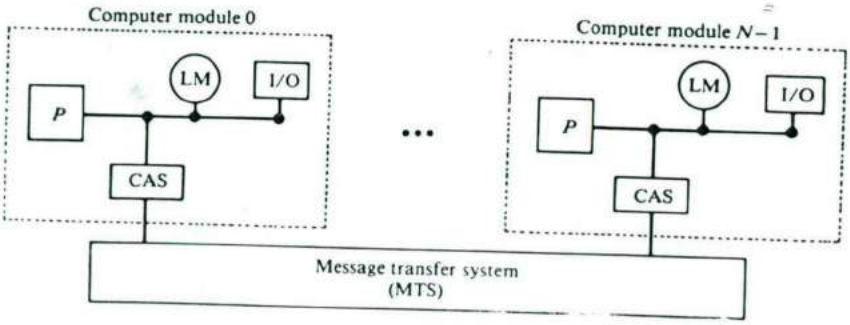
7.1.1. Loosely Coupled Multiprocessors

Loosely coupled multiprocessor systems do not generally encounter the degree of memory conflicts experienced by tightly coupled systems. In such systems, each processor has a set of input-output devices and a large local memory where it accesses most of the instructions and data. We refer to the processor, its local memory and I/O interfaces as a *computer module*. Processes which execute on different computer modules communicate by exchanging messages through a message-transfer system (MTS). The degree of coupling in such a system is very loose. Hence, it is often referred to as a distributed system. The determinant factor of the degree of coupling is the communication topology of the associated message-transfer system. Loosely coupled systems (LCS) are usually efficient when the interactions between tasks are minimal. Tightly coupled systems (TCS) can tolerate a higher degree of interactions between tasks without significant deterioration in performance.

Figure 7.1*a* shows an example of a computer module of a nonhierarchical loosely coupled multiprocessor system. It consists of a processor, a local memory, local input-output devices and an interface to other computer modules. The interface may contain a channel and arbiter switch (CAS). Figure 7.1*b* illustrates the connection between the computer modules and a message-transfer system. If requests from two or more different computer modules



(a) A computer module



(b) Loose coupling of computer modules

Figure 7.1 Nonhierarchical loosely coupled multiprocessor system.

collide in accessing a physical segment of the MTS, the arbiter is responsible for choosing one of the simultaneous requests according to a given service discipline. It is also responsible for delaying other requests until the servicing of the selected request is completed. The channel within the CAS may have a high-speed communication memory which is used for buffering block transfers of messages. The communication memory is accessible by all processors. With the advent of VLSI technology, the computer module can be fabricated on a single integrated circuit and be used as the building block of a multiprocessor system.

The message-transfer system for a nonhierarchical LCS could be a simple time shared bus, as in the PDP-11, or a shared memory system. The latter case can be implemented with a set of memory modules and a processor-memory interconnection network or a multiported main memory. In a multiported memory system, the arbitration and selection logic of the switch are distributed into the memory modules. The MTS is one of the most important factors that determine the performance of the multiprocessor system. For LCS configurations that use a single time shared bus, the performance is limited by the message arrival rate on the bus, the message length, and the bus capacity (in bits per second).

Contentions for the bus increase as the number of computer modules increases. For the LCS with a share memory MTS, the limiting factor is the memory conflict problem imposed by the processor-memory interconnection network.

The communication memory may also be centralized and connected to a time shared bus, or be part of the shared memory system. Conceptually, a distributed or centralized communication memory can be considered as consisting of logical ports which can be accessed by the processors. Processes (tasks) can communicate with other processes allocated to the same processor, or with tasks allocated to other processors. Associated with each task is an input port stored in the local memory of the processor to which the task is allocated. Every message issued by the task is directed to the input port of the destination task. Communication between tasks allocated to the same processor takes place through the local memory only. Communication between tasks allocated to different processors is through a communication port residing in the communication memory. One communication port is associated with each processor as its input port.

The logical structure of the communication between tasks is shown in Figure 7.2. A process allocated to processor P_1 puts a message into the input port of another task in P_i, as illustrated by the arrow marked with **a.** The **b** arrows show a two-step action in transferring messages between processors. Arrow $\mathbf{b}_1$ sends a message to the input port of processor P_2. Arrow $\mathbf{b}_2$ shows the moving of a message to the input port of the destination process.

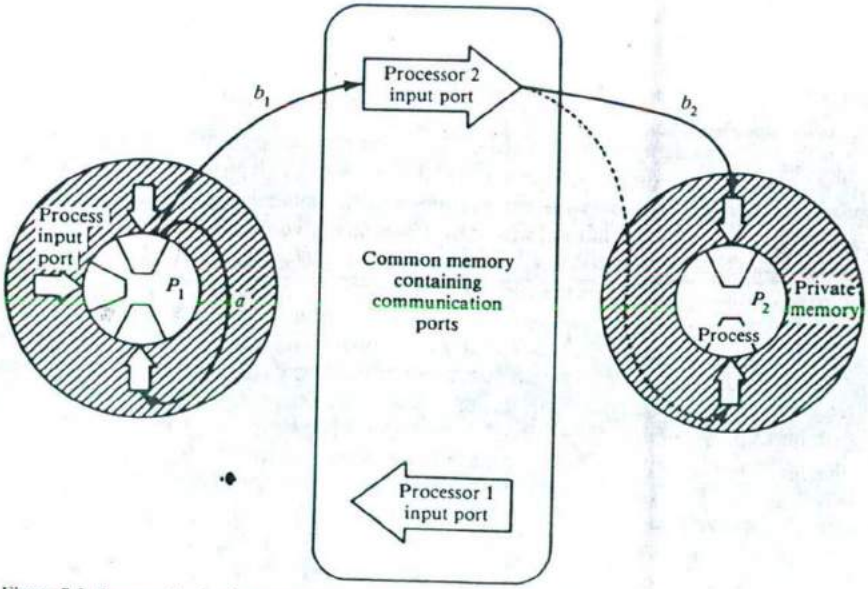
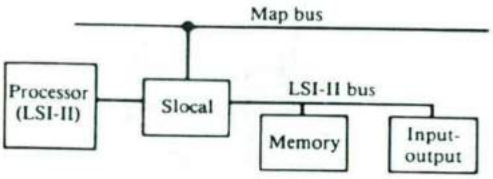
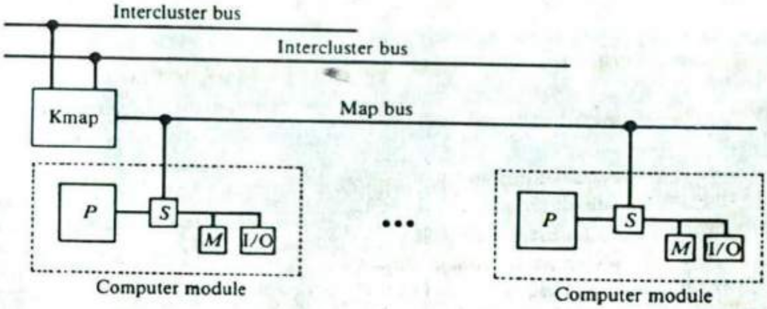


Figure 7.2 Communication between processes in a multiprocessor environment.

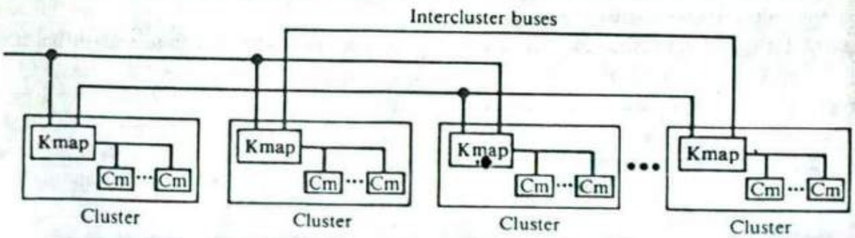
The Cm* Architecture For a hierarchical LCS, we consider the example of a computer system project at Carnegie Mellon University called the Cm*. Each computer module of the Cm* includes a local switch called the *Slocal*, as shown in Figure 7.3*a*. The switch is somewhat similar to the CAS in Figure 7.1*a*. The Slocal intercepts and routes the processor's requests to the memory and I/O devices outside the computer module via a map bus, shown in Figure 7.3*b*. It also accepts references from other computer modules to its local memory and I/O devices. The address translation is shown in Figure 7.4. It uses the four high-order bits of the processor's address along with the current address space, as indicated by the X-bit of the LSI-11 processor status word (PSW), to access a mapping table



(*a*) A computer module



(*b*) A cluster of computer modules



(c) A network of clusters

Figure 7.3 Hierarchically structured multiprocessor system.

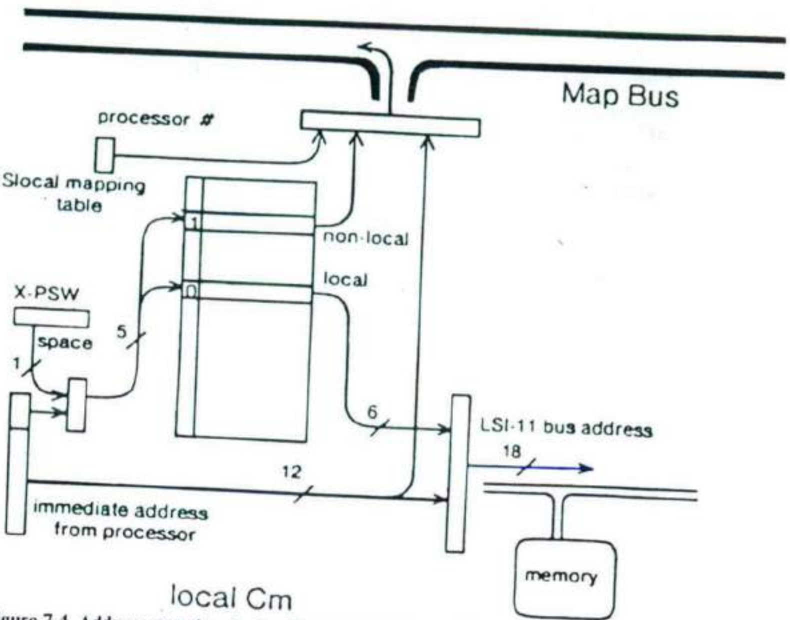


Figure 7.4 **Address mapping in the Slocal of the Cm*. (Courtesy of Cm* project at Carnegie-Mellon University, 1980.)**

which determines whether the reference is local or nonlocal. A virtual address of the nonlocal reference is formed by concatenating the nonlocal address field given by the mapping table and the source processor's identification. This virtual address is subsequently fetched by the Kmap via the map bus in response to a service request for nonlocal access. A number of computer modules may be connected to a map bus so that they share the use of a single Kmap. The *Kmap* is a processor that is responsible for mapping addresses and routing data between Slocals.

The computer modules are connected in hierarchical clusters by two-level buses, as shown in Figures 7.3*b* and 7.3*c*. A *cluster*, which is regarded as the lowest level, is made up of computer modules, a Kmap and the map bus. Clustering can enhance the cooperative ability of the processors in a cluster to operate on shared data with low communication overhead. It also provides hardware facilities to execute a group of tightly coupled cooperating processes. Any nonlocal reference to memory is handled by the Kmap in the cluster of the target memory module. The map bus may create a bottleneck since only one transaction can take place at a time. The performance of the system is facilitated by interconnecting the clusters of computer modules in a hierarchy. Clusters communicate via *intercluster buses* which are connected between Kmaps, as shown in Figure 7.3*c*. In general, a cluster need not have a direct intercluster bus connection to every other cluster in the configuration. Hence the complexity of the interconnection network can be simplified.

The Kmap is a microprogrammed, 150-ns cycle, three-processor complex with a common data memory. The Kmap provides the address mapping, communication, and synchronization functions within the system. Moreover, key operating system primitives can be moved into the Kmap, thereby relieving the computer modules from major supervisor functions. The data memory is used to cache address translation tables and mechanisms for synchronization and other resource management functions.

The three processors in the Kmap are the *Kbus*, the *Linc*, and the *Pmap*, as shown in Figure 7.5. The Kbus is the bus controller which arbitrates requests to the map bus. The Linc manages communication between the Kmap and other Kmap. The Pmap is the mapping processor which responds to requests from the Kbus and Linc. It also performs most of the request processing. The Pmap communicates with the computer modules in its cluster via the map bus which is a packet-switched bus. Three sets of queues provide interfaces between the Kbus, Linc, and the Pmap. Since the Kmap is much faster than the memory in the computer modules, it is multiprogrammed to handle up to eight concurrent requests. Each of the eight partitions is called a *context* and exists in the Pmap. Typically, each context processes one transaction. If one context needs to wait for a message packet to return with the reply to some request, the Pmap switches to another context that is ready to run so that some other transaction can proceed concurrently.

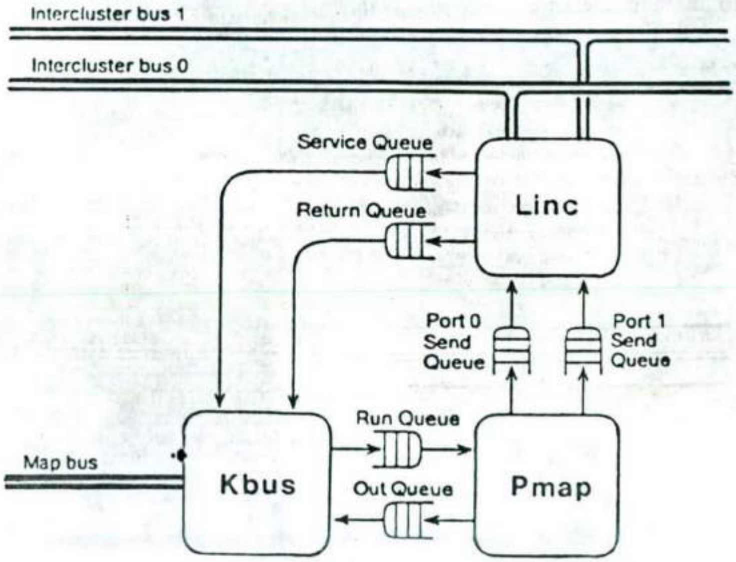
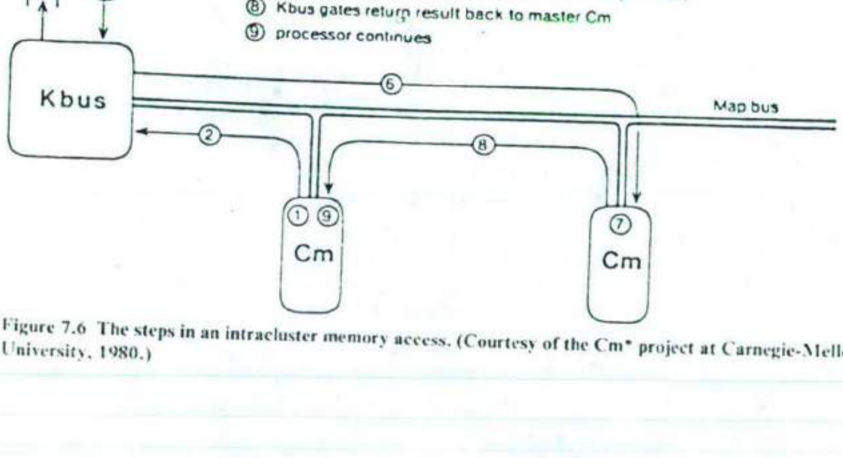


Figure 7.5 The components of the Kmap in Cm*. (Courtesy of the Cm* project at Carnegie-Mellon University, 1980.)

A service request is signaled to the Kbus whenever the processor of a computer module makes a nonlocal memory reference. Such a computer module (Cm) is called the *master* Cm. In response to the service requests, the Kbus allocates a Pmap context and fetches the virtual address of the memory reference via the map bus. Furthermore, the Kbus activates the new context by placing an entry, which contains the virtual address, in the Pmap run queue. The Pmap context performs the vritual to physical address translation via a microsubroutine. Using the physical address, it can initiate a physical memory access in any of the computer modules in its cluster.

The initiation is performed when the context invokes the appropriate Kbus operation by loading a request (with the physical address) into the Kbus *out queue*, shown in Figure 7.5. After loading the request into the out queue, the Pmap makes a context switch to another runnable context. The Kmap services the out requests by sending the physical address of the memory request via the map bus to the destination Cm. When the destination Cm completes the memory access, it signals a return request to the Kmap, which fetches the result of the memory access via the map bus and reactivates the request context. These steps, which are taken in an intracluster memory access, are depicted in Figure 7.6.

The intercluster bus is also a packet-switched bus which is jointly controlled by the Linc processors in each of the Kmaps directly connected to the bus. The Linc maintains queues of incoming and outgoing messages and interacts with the Kbus to activate and reactivate Pmap contexts. Each Linc interfaces to two independent intercluster buses, as shown in Figure 7.5. An intercluster message is sent from an immediate source Kmap to an immediate destination Kmap (in one



① processor initiates non-local memory access
② Kbus reads virtual address from master Cm
③ context activation waits in run queue
④ Pmap microsubroutine performs address translation
⑤ request for memory cycle waits in out queue
⑥ Kbus sends physical address to destination Cm
⑦ destination Cm steals memory cycle from its processor
⑧ Kbus gates return result back to master Cm
⑨ processor continues
Pmap
Kbus
Map bus
Cm
Cm

Figure 7.6 The steps in an intracluster memory access. (Courtesy of the Cm* project at Carnegie-Mellon University, 1980.)

hop). These intercluster messages were designed to be used as a mechanism for remote procedure calls between Kmaps. The identity of the destination is encoded in the message so that the Linc can determine which message is sent to its cluster. If the final destination is several hops away from the initial source, the message is handled by intermediate Kmaps until it reaches the final destination.

Intercluster messages are of two types: *forward* messages, which invoke a new context at the destination Kmap, and *return* messages, which return to a waiting context also at the destination Kmap. A return message contains the context number of the context which is to be reactivated in the Pmap. When a Pmap context desires to invoke some operation in another Kmap, it prepares a forward intercluster message, instructs the Linc to transmit it on a specified intercluster bus and then swaps context. The forward message includes the source Kmap number and the originating Pmap context number so that the remote Kmap will be able to send back a return message.

When the remote Linc receives a forward message, it causes the Kmap to activate a new Pmap context to decode the message and respond to the request. It is assumed that the message contains some operation code which the Pmap context can identify and execute. After performing the operation, the context prepares a return intercluster message, and instructs the Linc to transmit it on a specified intercluster bus. The Pmap also informs the Kbus that the context is now free and switches context. As an example of such a cross-cluster operation, we consider the mapping of a nonlocal memory reference to a location in the physical memory of another cluster. Some computer module initiates the nonlocal memory reference which activates a context in its cluster's Kmap. This context becomes the master context and the Kmap becomes the master Kmap. The destination Kmap is the slave Kmap and the new context activated at the slave Kmap is the slave context. Figure 7.7 depicts the steps involved in performing a multicluster operation.

Collectively, the Kmaps and the Slocals form the distributed memory switch. They mediate each nonlocal reference, thus sustaining the appearance of a single large, uniformly addressed memory. However, memory reference times may have large variations. Approximate interreference times for local, intracluster, and intercluster references are 3, 9, and 26 μs, respectively. The distributed memory switch makes the Cm* structure potentially more reliable than tightly coupled systems because the system can still be operational in a degraded mode when an intercluster bus fails. The LSI-11 processor used in the Cm is a relatively slow processor (approximately a 0.1 MIPS) in comparison to the Kmap. It is not quite obvious how the design of the Kmap would be affected if a faster processor is used in the Cm. The Kmap technology and design affect the service rate it provides. In large systems, it can be a limiting factor. The Cm* architecture seems well suited for parallel algorithms with high locality in an intracluster bus. If locality is poor and the processes requests frequently cross intercluster buses, the performance of the algorithm may become poor.

The two-level hierarchy can be extended to an n-level hierarchy. An example could consist of a binary tree structure with n levels. In such a structure, the tree has a root node and each node which represents a cluster has six branches, with

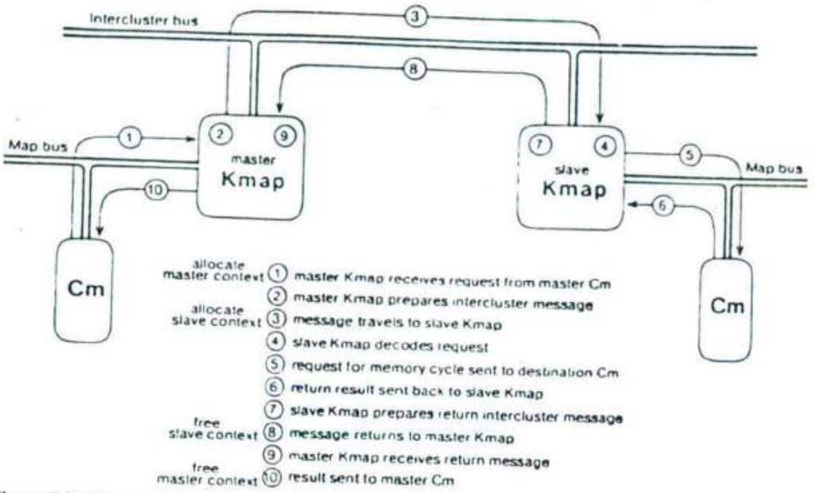

Intercluster bus
Map bus
master Kmap
slave Kmap
Map bus
Cm
Cm
allocate master context ① master Kmap receives request from master Cm
② master Kmap prepares intercluster message
allocate slave context ③ message travels to slave Kmap
④ slave Kmap decodes request
⑤ request for memory cycle sent to destination Cm
⑥ return result sent back to slave Kmap
⑦ slave Kmap prepares return intercluster message
free slave context ⑧ message returns to master Kmap
⑨ master Kmap receives return message
free master context ⑩ result sent to master Cm

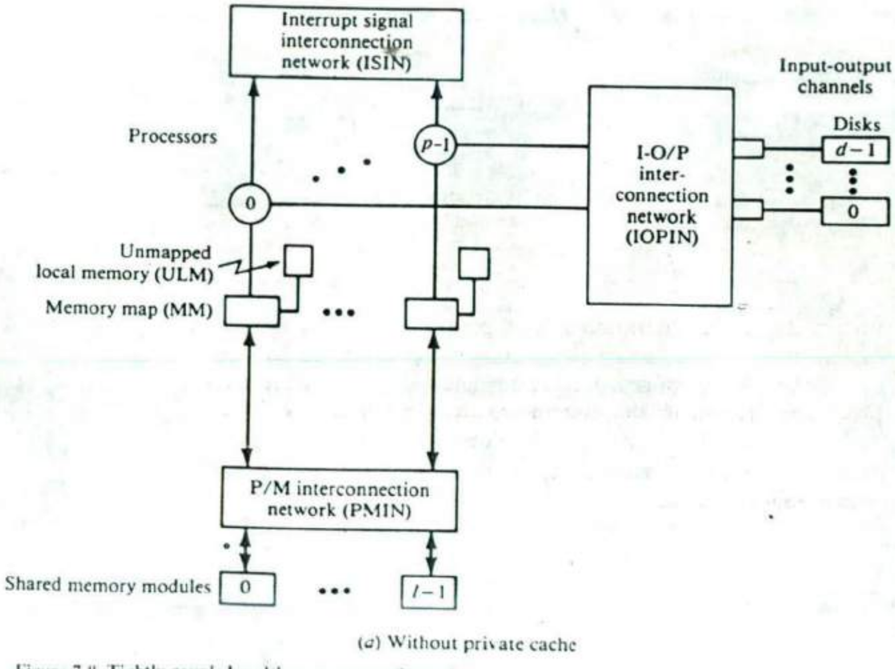
Figure 7.7 The steps in a cross-cluster memory access. (Courtesy of the Cm* project at Carnegie-Mellon University, 1980.)

the exception of the leaf nodes. The total number of nodes is $(b^n - 1)/(b - 1)$, and the number of links is $(b^n - b)/(b - 1)$. In a binary tree, each cluster is connected strictly to its two children and to a single parent. Communication between leaf nodes faces a bottleneck toward the top of the tree. Hence, such tree structures perform well on a large range of problems. Binary tree structures have been shown to be theoretically promising for sorting, matrix multiplication, and for solving some NP-hard problems. The basic scheme involves divide-and-conquer techniques.

7.1.2 Tightly Coupled Multiprocessors

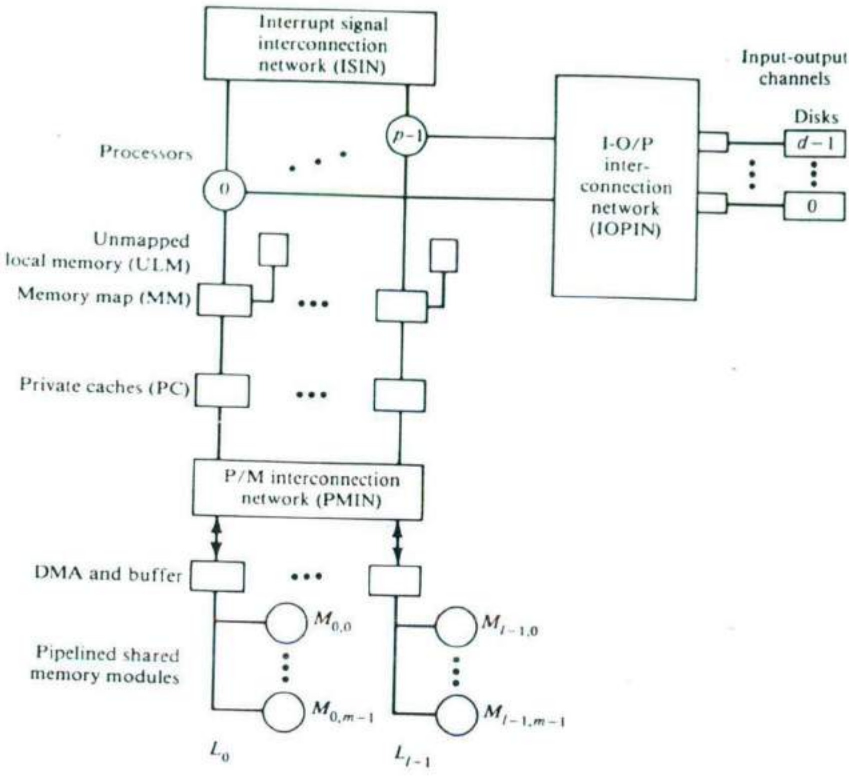
Because of the large variability of interreference times, the throughput of the hierarchical loosely coupled multiprocessor may be too low for some applications that require fast response times. If high-speed or real-time processing is desired, tightly coupled systems (TCS) may be used. Two typical models of a TCS are discussed. The first model is shown in Figure 7.8*a*. It consists of p processors, l memory modules, and d input-output channels. These units are connected through a set of three interconnection networks, namely, the processor-memory interconnection network (PMIN), the I/O-processor interconnection network (IOPIN), and the interrupt-signal interconnection network (ISIN). The PMIN is a switch which can connect every processor to every memory module. Typically, this switch is a p by l crossbar which has pl sets of cross points. A set of cross points for a particular processor-memory pair includes $(n + k)$ cross points, where n is the width of the address within a module and k is the width of the data path. Hence the crossbar switch for a p by l multiprocessor system has a complexity $O(pl(n + k))$. For large p and l, the crossbar may dominate the cost of the multiprocessor system. If the crossbar switch is distributed across the memory modules, a multiported memory results. The complexity of the multiported memory is similar to that of the crossbar. Alternately, the PMIN could be a multistage network, some examples of which were discussed in Chapter 5.

A memory module can satisfy only one processor's request in a given memory cycle. Hence, if two or more processors attempt to access the same memory module, a conflict occurs which is resolved or arbitrated by the PMIN. If necessary the PMIN may be designed to permit broadcasting of data from one processor to two or more memory modules. To avoid excessive conflicts, the number of memory modules l is usually as large as p. Another method used to reduce the degree of conflicts is to associate a reserved storage area with each processor. This is the unmapped local memory (ULM) in Figure 7.8*a*. It is used to store Kernel code and operating system tables often used by the processes running on that processor. For example, if each processor is multiprogrammed, each time a task switch is desired the state of the process to be blocked may be saved in the ULM. The ULM helps in reducing the traffic in the PMIN and hence the degree of conflicts,


Interrupt signal interconnection network (ISIN)
Input-output channels
Disks
Processors
p-1
0
d-1
0
I-O/P inter-connection network (IOPIN)
Unmapped local memory (ULM)
Memory map (MM)
P/M interconnection network (PMIN)
Shared memory modules
0
l-1

(*a*) Without private cache

Figure 7.8 Tightly coupled multiprocessor configurations.



Interrupt signal interconnection network (ISIN)
Input-output channels
Disks
$d-1$
0
Processors
$p-1$
0
I-O/P inter-connection network (IOPIN)
Unmapped local memory (ULM)
Memory map (MM)
Private caches (PC)
P/M interconnection network (PMIN)
DMA and buffer
$M_{0,0}$
$M_{l-1,0}$
Pipelined shared memory modules
$M_{0,m-1}$
$M_{l-1,m-1}$
L_0
L_{l-1}

(*b*) With private cache

Figure 7.8 (*continued*)

provided process migration is not permitted. An example of a multiprocessor system with similar processor memory configuration to that shown in Figure 7.8*a* is the C.mmp multiprocessor designed at Carnegie Mellon University with 16 processors. Details of this system are given in Chapter 9.

In the multiprocessor organization of Figure 7.8*a*, each processor may make memory references which are accessed in main memory. These memory references contribute to the memory conflicts at the memory modules. Since each memory reference goes through the PMIN, it encounters delay in the processor memory switch and, hence, the instruction cycle time increases. The increase in the instruction cycle time reduces the system throughput. 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 crossbar switch can be reduced, which subsequently reduces the contention at the cross points. A multiprocessor organization which uses a private cache with each processor is shown in Figure 7.8*b*. This multiprocessor organization encounters the cache coherence problem. More than one inconsistent copy of data may exist in the system. Various solutions to the cache coherence problem are given in Section 7.3. Examples of multiprocessors with private caches are the IBM 3084 and the S-1.

In Figure 7.8, there is a module attached to each processor that directs the memory references to either the ULM or the private cache of that processor. This module is called the *memory map* and is similar in operation to the Slocal discussed earlier. The general scheme for implementing memory maps was discussed in Chapter 2. The ISIN permits each processor to direct an interrupt to any other processor. Synchronization between processors is facilitated by the use of such an interprocessor network. The ISIN can also be used by a failing processor to broadcast a hardware-initiated alarm to the functioning processors. The IOPIN permits a processor to communicate with an I/O channel which is connected to peripheral devices.

The complexity of the ISIN may vary from a simple time shared bus to a complex crossbar switch. For example, in the Univac 1100/80 and Honeywell 60/66 multiprocessor systems, a connection is established between every pair of processors for the ISIN. The C.mmp system uses a time shared bus for interprocessor communication. A time shared bus is much cheaper than a crossbar switch but encounters more contentions and delays due to bus-arbitration logic. However, the interrupt request rate to the bus is usually low enough to make the shared bus an attractive solution to interprocessor communication.

The set of processors used in a multiprocessor system may be homogeneous or heterogeneous. It is homogeneous if the processors are functionally identical. For example, the multiprocessor system of the IBM 3081K has two identical processors. Even if the processors are homogeneous, they may be asymmetric. That is, two functionally identical components may differ along other dimensions, such as I/O accessibility, performance or reliability. Examples with symmetric multiprocessor configurations are the Honeywell 60/66 and the Univac 1100/80. Examples of the asymmetric multiprocessors are the attached processor systems such as the IBM 3084 AP and the C.mmp.

In most cases, the asymmetry or symmetry of the multiprocessor system is usually transparent to the user processes. It is only of interest to the operating system, especially with respect to load balancing and other scheduling considerations. In general, a homogeneous system is easier to program and eliminates the connector problem, which arises in getting two dissimilar processors to effectively communicate. The symmetric system usually can better facilitate error recovery in case of failure.

Input-output asymmetricity The asymmetricity of the processors can also be extended to the input-output devices with respect to the connectivity of these devices to the processors. An I/O interconnection network that has complete connectivity is symmetric. Because symmetric systems are usually expensive, some multiprocessors have a high degree of asymmetry in the I/O subsystem. Figure 7.9 is an example of an asymmetric I/O subsystem. In such systems, devices attached to one processor cannot be directly accessed by another processor. This is the example I/O subsystem used in the C.mmp. A fully symmetric subsystem is more flexible and provides more accessibility.

In a fully symmetric structure, the failure of a central processor does not preclude the accessibility of a given device by any other processor. In the asymmetric case, the failure of a CPU causes all devices attached to that processor to become inaccessible. Furthermore, a request for data transfer to an I/O device which is attached to a given processor causes undesirable task switching overhead if the request is made by another processor. The inaccessibility problem encountered by a set of devices attached to a faulty processor can be slightly overcome by having redundant connections, as exemplified in Figure 7.10. In this example,

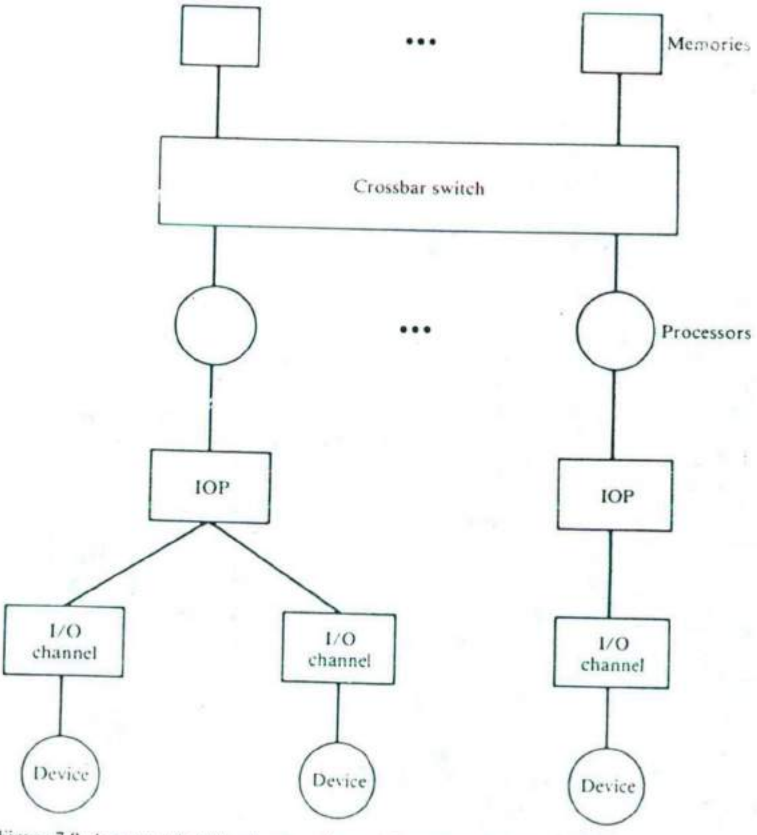
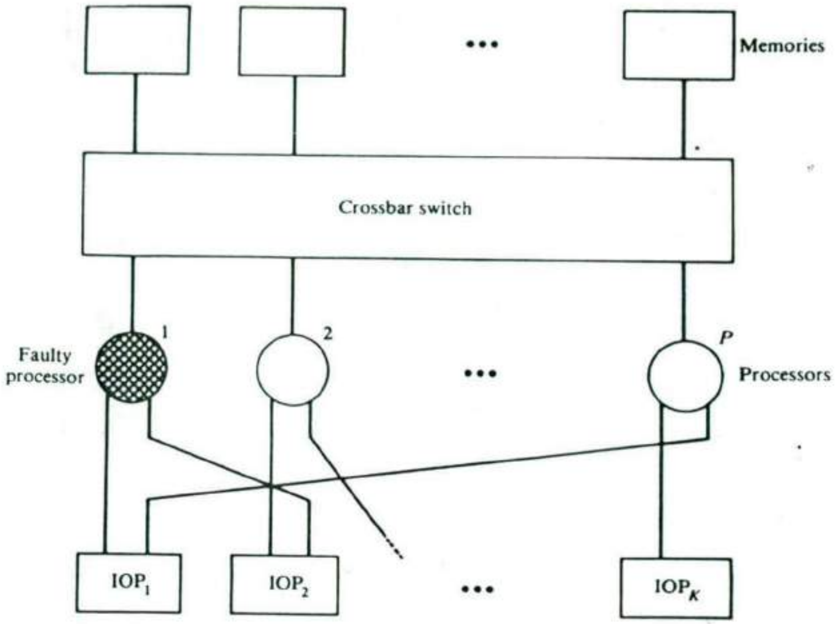


Figure 7.9 Asymmetric I/O subsystem in a multiprocessor system.

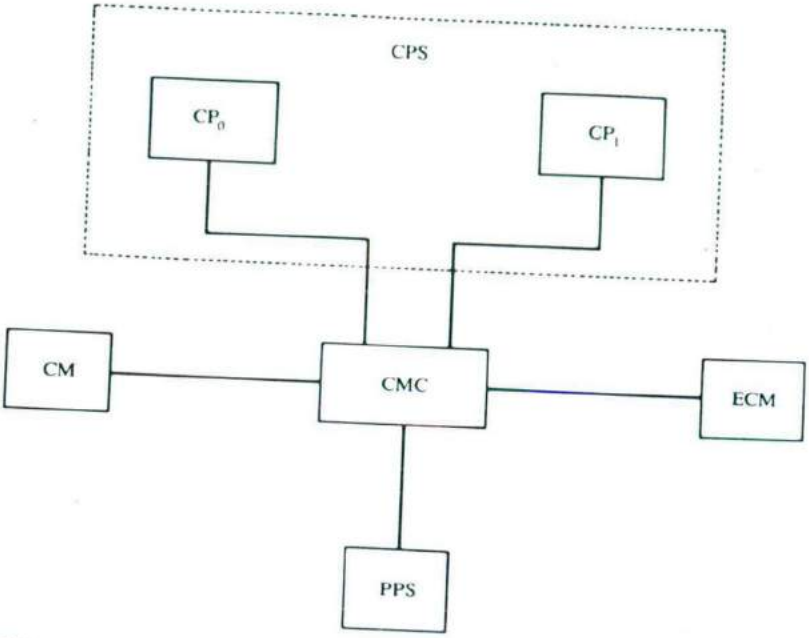


IOP_1 still accessible in the event of processor 1 failure

Figure 7.10 Increased availability in an asymmetric I/O subsystem through redundant connections.

IOP_1 is still accessible through processor P when processor 1 fails. The availability is provided at the cost of additional arbitration logic required for the multiple paths. Also, the extra logic must be sufficiently reliable that the degradation it introduces is more than compensated by the extra reliability of redundant paths. However, if the reliability of the extra logic is poor, then the reliability and availability of the system in Figure 7.10 will be poorer than that of the original system. The disadvantage of the fully symmetric case is the cost of the crossbar switch. This cost can be reduced without significant sacrifice in availability by using a multistage network such as the delta network discussed in the previous section, or a multiported system. Three examples of a tightly coupled multiprocessor system are the Cyber-170, the Honeywell 60/66, and the PDP-10. These examples are briefed below and details can be found in Satyanarayanan (1980).

The Cyber-170 architecture Figure 7.11 shows an example configuration of a Cyber-170 multiprocessor system. This configuration consists of two subsystems—the central processing subsystem and the peripheral processing subsystem. These subsystems have access to a common central memory (CM) through a central memory controller, which is essentially a high-speed crossbar switch. In addition to the central memory, there is an optional secondary memory called the extended



CM: central memory

CMC: central memory controller

CP_i: ith central processor

CPS: central processing subsystem

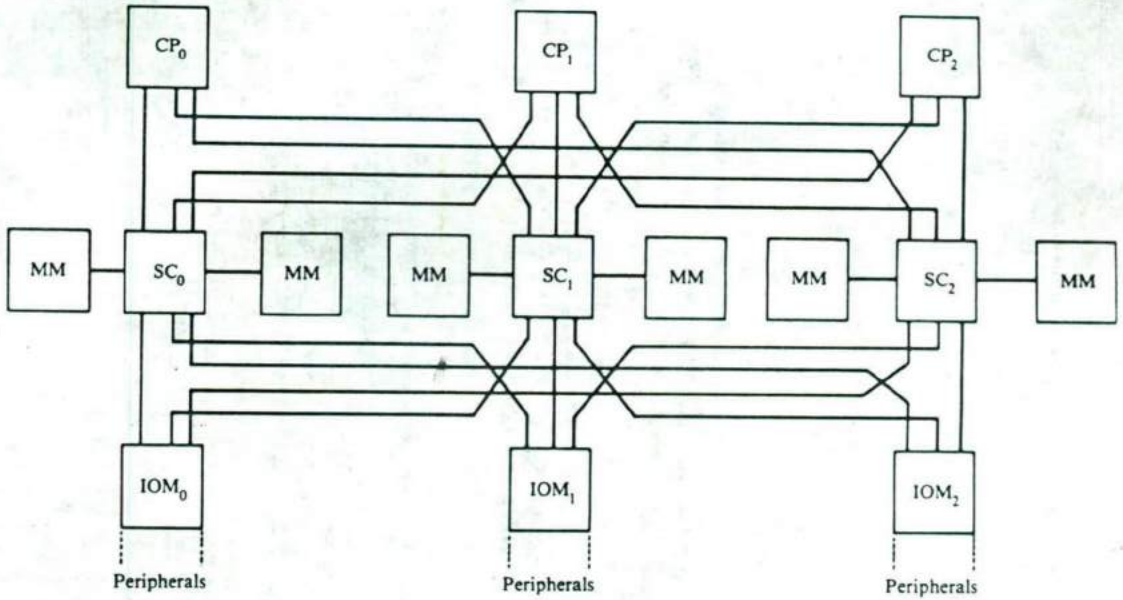
ECM: extended core memory

PPS: peripheral processing subsystem

Figure 7.11 A Cyber-170 multiprocessor configuration with two processors. (Courtesy of Control Data Corp.)

core memory (ECM), which is a low-speed random-access read-write memory. The ECM and the CM form a two-level memory hierarchy. In this configuration, the CMC becomes the switching center, which performs the combined functions of the ISIN, IOPIN, and PMIN described in Figure 7.8.

Honeywell 60/66 architecture A configuration of a Honeywell 60/66 multiprocessor system is shown in Figure 7.12. In this system, every central processor and every I/O multiplexer is connected to every controller (SC). This provides adequate redundancy in paths for high availability. In the event of a failure on the SC, all IOMs are still accessible by each processor. The system controller acts as a memory controller for its associated pair of memory modules. It also acts as an



CP_i: *i*th central processor

IOM: I/O multiplexer

MM: memory

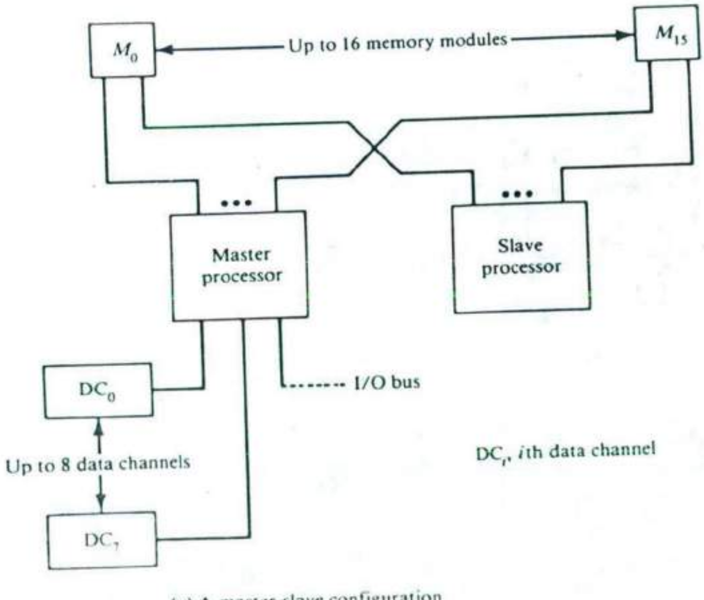
SC_i: *i*th system controller

Figure 7.12 A Honeywell 60/66 multiprocessor system. (Courtesy of Honeywell Corp.)

intelligent switch to route interrupts and other communications among the various system components. When more than one element attempts to access the same memory module, the corresponding system controller resolves the conflict. This triple redundancy organization is particularly designed to enhance availability and fault tolerance.

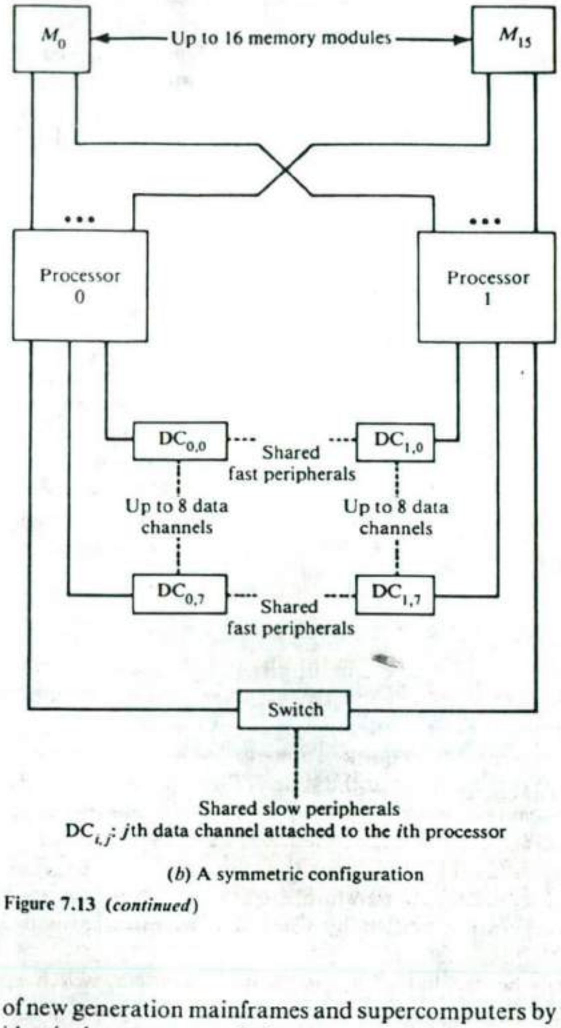
The PDP-10 multiprocessor Figure 7.13 shows two configurations of the PDP-10 multiprocessor with multiported memory modules. Each CPU has a cache of 2K words where each word is 36 bits. Figure 7.13*a* illustrates the asymmetric master-slave configuration. The two processors are identical, but the asymmetry is a result of the connection of the peripherals to the master only. Hence, the slave cannot initiate peripheral operations nor respond to an interrupt directly. The symmetrical configuration of the PDP-10 multiprocessor is shown in Figure 7.13*b*. Both processors are connected to a set of shared fast and slow peripherals. However, each data channel is attached to one processor, which is the only processor that can use it. Note that slow peripherals are connected to both processors via a switch. There is no cache invalidate interface between them. It is assumed that a software solution is used to enforce cache consistency.

The three tightly coupled multiprocessors discussed above are just a few of the commercial systems available. There is a trend to achieve improved performance



(*a*) A master-slave configuration

Figure 7.13 Two architectural configurations of the PDP-10 multiprocessor system. (Courtesy of Digital Equipment Corp.)



$DC_{i,j}$: jth data channel attached to the ith processor

(*b*) A symmetric configuration

Figure 7.13 (*continued*)

of new generation mainframes and supercomputers by tightly coupling a number of identical processors. Other commercial systems discussed in Chapter 9 are UNIVAC 1100/94, IBM 3081, HEP, and Cray X-MP. The relative merits of tight coupling depend on cost effectiveness, which is technology dependent.

One of the advantages of a multiprocessing system is its potential for effective error recoverable capability. Recoverability, however, is not synonymous with reliability. The number of interconnections and the multiplicity of processing units and other system modules may, in fact, cause a higher probability of failure on a multiprocessor than on a uniprocessor. However, the inherent redundancy in a multiprocessor system probably increases its ability to be fault tolerant.

7.1.3 Processor Characteristics for Multiprocessing

Most multiprocessors have been built using processors not originally designed for multiprocessor architecture. Two examples of these are the C.mmp system, which uses DEC's PDP-11 processors, and the Cm* multiprocessor, which uses LSI-11 microprocessors. One reason for using off-the-shelf components is to shorten development time. However, these off-the-shelf components can create undesirable features in the system. A number of desirable architectural features are described below for a processor to be effective in a multiprocessing environment.

Process recoverability The architecture of a processer used in a multiprocessor system should reflect the 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 the process can continue. Without this feature, the potential for reliability is substantially reduced. Most processors contain the process state of the current-running process in internal registers which are not accessible from outside the processor and are not written to memory in the event of a fault. With current technology, it should be possible to separate the general-purpose registers from the processor itself without much loss of speed. It is desirable to have a register file shared by all processors in the event of a gracefully degraded operation mode.

Efficient context switching Another reason for having a shared general-purpose register is that a large register file can be used in a multi-programmed processor. For effective utilization, it is necessary for the processor to support more than one addressing domain and, hence, to provide a domain-change or context-switching operation. Such switching operations require extensive queueing and stack operations. The context switch operation saves the state of the current process and switches to a selected ready-to-run process by restoring the state of the new process. The state of a running process is indicated by the contents of the processor registers. An example of a processor with multiple domains 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) provided in the Cyber-170 processor, which has 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 in the central memory. This area is called the exchange package. Such save are restore operations for all processor registers can, if not properly designed, significantly contribute to the overhead in establishing concurrency in the system.

By providing an ample number of register sets, a task switch can be accomplished efficiently by changing the contents of the *current process register* in the processor to point to the register set containing the state of the selected process, as shown in Figure 7.14. The current process register points to the register set currently in use. For example, each processor in the S-1 multiprocessor system has 16 sets of registers. Stack instructions which rapidly save and restore the processor status word tend to minimize switching overheads. The implementation of reentrant procedure calls is related to the stack manipulative structure of the processor.

Large virtual and physical address spaces A processor intended to be used in the construction of a general-purpose medium to large-scale multiprocessor must support a large physical address space. Even when an algorithm is decomposed so that it can be implemented using very small amounts of code, processes sometimes need to access large amounts of data objects. The 16-bit address space of the processor used in the 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, the virtual address space should be segmented to promote modular sharing and the checking of address bounds for memory protection and software reliability. For example, the processor used in the S-1 multiprocessor system has 2 gigabytes of virtual memory and 4 gigawords of physical memory, where each word is 36 bits wide.

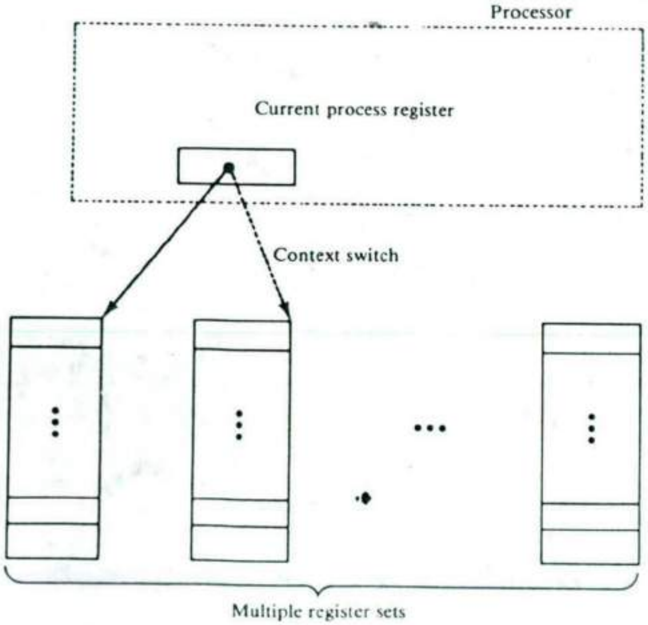


Figure 7.14 Context switching in a processor with multiple register sets.

Effective synchronization primitives The processor design must provide some implementation of indivisible actions which serve as the basis for *synchronization primitives*. These synchronization primitives require efficient mechanisms for establishing mutual exclusion. Mutual exclusion is required when two or more processes are in execution concurrently and must cooperate to exchange data during the computation. Mechanisms for establishing mutual exclusion involve some form 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 using the 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 are the *test-and-set* and *compare-and-swap*. These primitives will be discussed in Chapter 8.

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 an 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 interprocessor-communication mechanisms are the IBM 370/168 MP, Cray X-MP, and the C.mmp, which will be discussed in Chapter 9. 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 processor must have a wait state or some mechanism to suspend the processor in a queue.

Instruction set The instruction set of the processor should have adequate facilities for implementing high-level languages that permit effective concurrency at the procedure level and for efficiently manipulating data structures. Instructions 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 timers 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. There are different implementations of the error-detection technique, but the basic idea is that the timer will in some way raise an error-condition indicator if it is not reset within a specified time limit. Various techniques used to interrupt a processor will be discussed later.

7.2 INTERCONNECTION NETWORKS

The principal characteristic of a multiprocessor system is the ability of each processor to share a set of main memory modules and, possibly, I/O devices. This sharing capability is provided through a set of two interconnection networks. One is between the processors and memory modules and the other, between the processors and the I/O subsystem. There are several different physical forms available for the interconnection network (IN). Four basic organizations of the IN are discussed in this section. The classification scheme presented here is due to Enslow (1977). Techniques are presented to evaluate the effective bandwidth of some interconnection networks.

7.2.1 Time Shared or Common Buses

The simplest interconnection system for multiple processors is a common communication path connecting all of the functional units. An example of a multiprocessor system using the common communication path is shown in Figure 7.15. 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 components such as

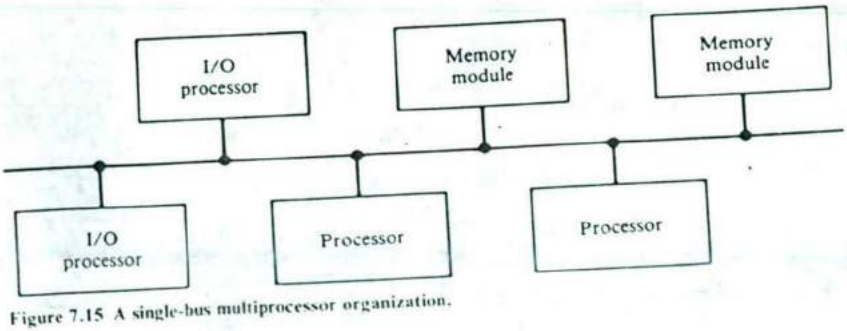


Figure 7.15 A single-bus multiprocessor organization.

switches. Transfer operations are controlled completely by the bus interfaces of the sending and receiving units. Since the bus is a shared resource, a mechanism must be provided to resolve contention.

The conflict-resolution methods include static or fixed priorities, first-in, first-out (FIFO) queues and daisy chaining. A centralized bus controller or arbiter, though simplifying the conflict resolution, may have negative effects on system reliability and flexibility. A unit (processor or I/O) that wishes to initiate a transfer must first determine the availability status of the bus, then address the destination unit to determine its availability and capability to receive the transfer. A command is also issued to inform the destination unit what operation it is to perform with the data being transferred, after which the data transfer is finally initiated. A receiving unit recognizes its address placed on the bus and responds to the control signals from the sender. These concepts, although basic, typify the operations on the bus.

An example of a time shared bus is the PDP-11 Unibus, which has 56 signal lines to provide the control, address lines, and data paths necessary to transfer 16-bit words. The function of each of the signals is shown in Table 7.1. The five bus-request signals are used for requesting bus control by a "master" at four different priority levels. They are also used for *attention signals* by a "slave." The master-slave assignment is dynamic. When a requester is granted control of the bus, it becomes a temporary master until it relinquishes control. The master can then select certain devices as slaves to control the transfer of data. Each of the request signals has a corresponding bus-grant signal. The functions of these signals are described in more details later.

Although the single-bus organization is quite reliable and relatively inexpensive, it does introduce a single critical component in the system that can cause complete system failure as a result of a malfunction in any of the bus interface circuits. Moreover, system expansion, by adding more processors or memory,

Table 7.1 PDP-11 unibus signals

Signal	Number of lines	Function
Address	18	Identifies destination of information (memory location or device address)
Data	16+ 2 parity	Information value
Control	2+	
	1 master sync 1 slave sync	Data transfer control
	1	Initialization
Bus request	5	Priority interrupt request
Bus grant	5 (unidirectional)	Bus assignment (made by CPU)
Bus busy	1	Bus status
Interrupt	1	Interrupt request
Selection acknowledge	1	Acknowledgment signal
Power fail	2	Power failure detection

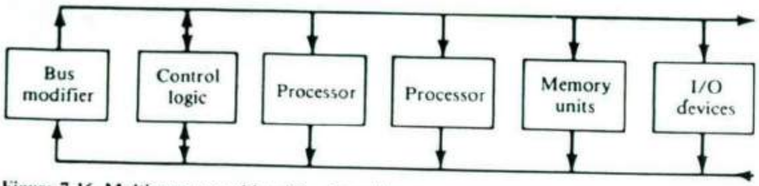


Figure 7.16 Multiprocessor with unidirectional buses.

increases the bus contention, which degrades system throughput and increases arbitration logic. The total overall transfer rate within the system is limited by the bandwidth and speed of this single path. For this reason, private memories and private I/Os are highly advantageous. Interconnection techniques that overcome these weaknesses add to the complexity of the system.

An extension of the single path organization to two unidirectional paths, as shown in Figure 7.16, alleviates some the problems mentioned above without an appreciable increase in system complexity or decrease in reliability. However, a single transfer operation in such a system usually requires the use of both buses, hence not much is actually gained.

The next step in alleviating the limitations of the time shared bus is to provide multiple bidirectional buses, as shown in Figure 7.17, to permit multiple simultaneous bus transfers; however, this increases the system complexity significantly. In this case, the interconnection subsystem becomes an active device. A number of computer systems, such as the Tandem-16 and Pluribus, employ variations of the time shared system of buses discussed above. In general, the above organizations are usually appropriate for small systems.

In view of the increasing numbers and speeds of devices attached to a central bus as a result of changing technology and applications, the bus can become heavily loaded. Therefore, the bus impairs the performance of the devices and, thus, of the overall system. There are several factors that affect the characteristics and performance of a bus. These include the number of active devices on the bus, the bus-arbitration algorithm, centralization (or distribution) of control, data width, synchronization of data transmission, and error detection. We will examine several

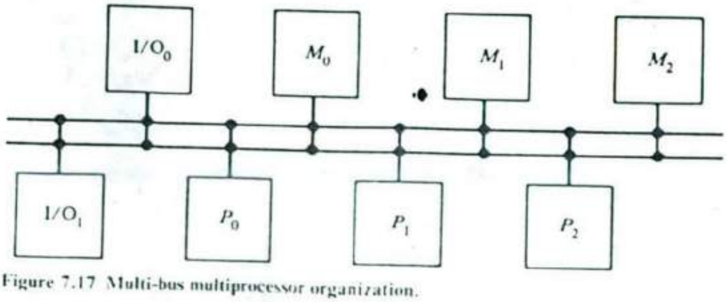


Figure 7.17 Multi-bus multiprocessor organization.

bus-arbitration algorithms which control access to the bus by the competing devices.

Current technology (processing speed) requires relatively simple algorithms for bus arbitration. These algorithms are usually implemented in the hardware and allow the arbitration for a bus cycle to be overlapped with the previous transfer.

(A) The static priority algorithm Many digital buses used today assign unique static priorities to the requesting devices. When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it. This approach is usually implemented using a scheme called *daisy chaining*, in which all services are effectively assigned static priorities according to their locations along a bus grant control line. The device closest to a central bus controller is assigned the highest priority (Figure 7.18). Requests are made on a common request line, BRQ. The central bus control unit propagates a bus grant signal (BGT) if the acknowledge signal (SACK) indicates that the bus is idle.

The first device which has issued a bus request that receives the BGT signal stops the latter's propagation. This sets the bus-busy flag in the controller and the device assumes bus control. On completion, it resets the bus-busy flag in the controller and a new BGT signal is generated if other requests are outstanding. The DEC PDP-11 Unibus uses this approach. The Motorola MC68000 processor incorporates such a bus control unit.

Another DEC bus called the synchronous backplane interconnect (SBI) and used in the VAX 11/780 computer implements static priorities using a distributed scheme called *parallel priority resolution*, in which the time required to determine which requesting device has the highest priority is fixed (unlike daisy chaining). Using static priorities clearly gives preferences and, thus, lower wait times to devices with higher priorities.

(B) The fixed time slice algorithm Another common bus-arbitration algorithm divides the available bus bandwidth into fixed-length time slices that are then sequentially offered to each device in a round-robin fashion. Should the selected

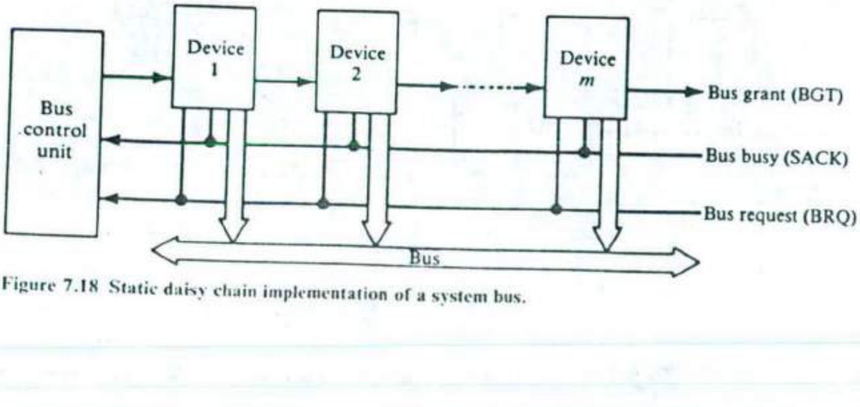


Figure 7.18 Static daisy chain implementation of a system bus.

device elect not to use its time slice, the time slice remains unused by any device. This technique, called *fixed time slicing* (FTS) or *time division multiplexing* (TMD), is used by Digital's Parallel Communications Link, which also allows a flexible assignment of available time slices to the devices. This scheme is usually used with synchronous buses in which all devices are synchronized to a common clock.

The service given to each device in the FTS scheme for access to the bus is independent of that device's position or identity on the bus; schemes with this characteristic are said to be symmetric. In particular, all *m* devices are given one out of every *m* time slices at fixed intervals in this scheme. Symmetric bus-arbitration algorithms optimally load-balance all bus requests because no preference is given to any device. It further delivers a bounded maximum wait time to the devices. However, it suffers a high average wait time (and, thus, a lower bus utilization).

When the bus is not heavily loaded, FTS incurs a substantially higher standard deviation from all wait times than does the static priority scheme, although the variability of service is lower and remains constant regardless of the bus load. Both algorithms offer good performance under light bus loading; these characteristics and their relative simplicity explain their widespread popularity.

(C) Dynamic priority algorithms The following dynamic priority algorithms allow the load-balancing characteristics of symmetric algorithms such as fixed time slicing to be achieved without incurring the penalty of high wait times. The devices are assigned unique priorities and compete to access the bus, but the priorities are dynamically changed to give every device an opportunity to access the bus. If the algorithm used to permute the priorities favors no individual device (is symmetric), then the system load balances the bus requests. Further, using priorities overcomes the inefficiency inherent in the fixed time slice scheme of allocating full time slices to the devices before requests are placed. Two algorithms for dynamically permuting priorities are the *least recently used* (LRU) and the *rotating daisy chain* (RDC).

The LRU algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. This is accomplished by reassigning priorities after each bus cycle. The second dynamic priority algorithm generalizes the daisy chain implementation of static priorities. Recall that in the daisy chain scheme all devices are given static and unique priorities according to their priorities on a bus-grant line emanating from a central controller.

In the RDC scheme, no central controller exists, and the bus-grant line is connected from the last device back to the first in a closed loop (Figure 7.19). Whichever device is granted access to the bus serves as bus controller for the following arbitration (an arbitrary device is selected to have initial access to the bus). Each device's priority for a given arbitration is determined by that device's distance along the bus-grant line from the device currently serving as bus controller; the latter device has the lowest priority. Hence, the priorities change dynamically with each bus cycle.

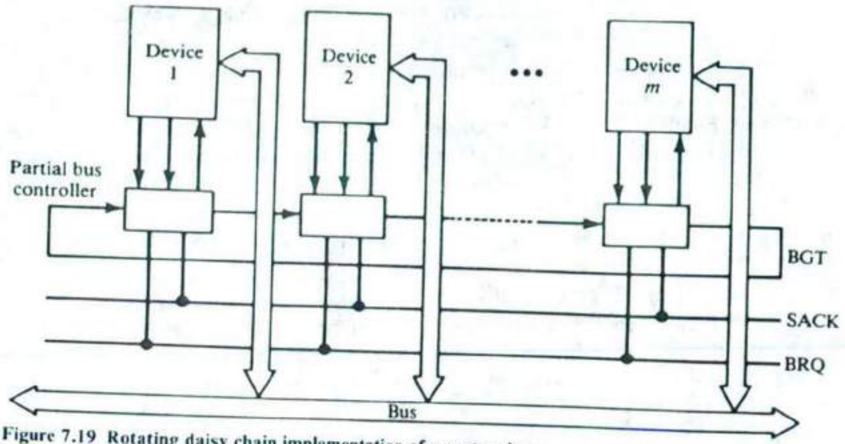


Figure 7.19 Rotating daisy chain implementation of a system bus.

(D) The first-come first-served algorithm In the *first-come, first-served* (FCFS) scheme, requests are simply honored in the order received. This scheme is symmetric because it favors no particular processor or device on the bus; thus, it load balances the bus requests. It has been shown that, under the condition of fixed service times by the central resource (fixed bus-transfer times in this case), FCFS yields the smallest possible average wait time and standard deviation of all wait times. In essence, FCFS is the optimal bus-arbitration algorithm with respect to these performance measures.

Unfortunately, FCFS is difficult to implement for at least two reasons. Any implementation of FCFS must provide a mechanism to record the arrival order of all pending bus requests, unlike the previous algorithms. More important, it is always possible for two bus requests to arrive within a sufficiently small interval so their relative ordering cannot be correctly distinguished. Hence, any implementation can only approximate the behavior of FCFS. Despite the above difficulties in realizing an implementation, it is important to measure the performance of FCFS as an indicator of the best possible performance that a bus-arbitration algorithm can achieve with respect to the above criteria.

Two other techniques used in bus-control algorithms are *polling* and *independent requesting*. In a bus-controller that uses polling, the bus grant signal (BGT) of the static daisy chain implementation is replaced by a set of $\lceil \log_2 m \rceil$ polling lines, as shown in Figure 7.20. The set of poll lines is connected to each of the devices. On a bus request, the controller sequences through the device addresses by using the poll lines. When a device D_i which requested access recognizes its address, it raises the SACK line (to indicate bus busy).

The bus-control unit acknowledges by terminating the polling process and D_i gains access to the bus. The access is maintained until the device lowers the SACK line. Note that the priority of a device is determined by its position in the polling sequence. In the independent requesting technique, a separate bus request (BRQ_i)

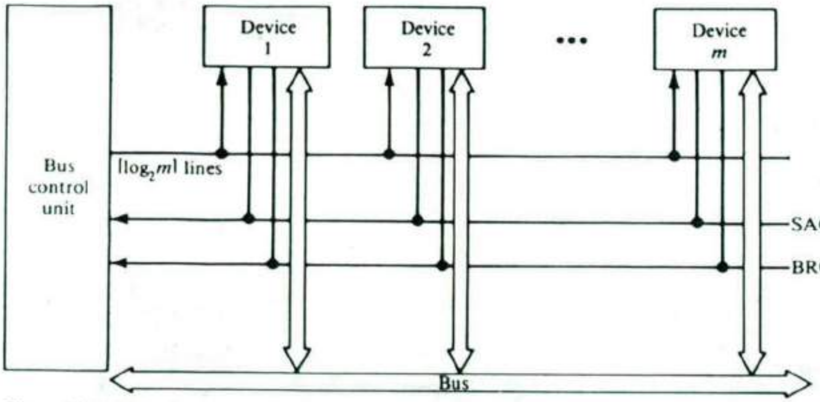


Figure 7.20 Polling implementation of a system bus.

and bus grant (BGT_i) line are connected to each device i sharing the bus, as shown in Figure 7.21. This requesting technique can permit the implementation of LRU, FCFS, and a variety of other allocation algorithms.

7.2.2 Crossbar Switch and Multiport Memories

If the number of buses in a time-shared bus system is increased, a point is reached at which there is a separate path available for each memory unit, as shown in Figure 7.22. The interconnection network is then called a *nonblocking* crossbar.

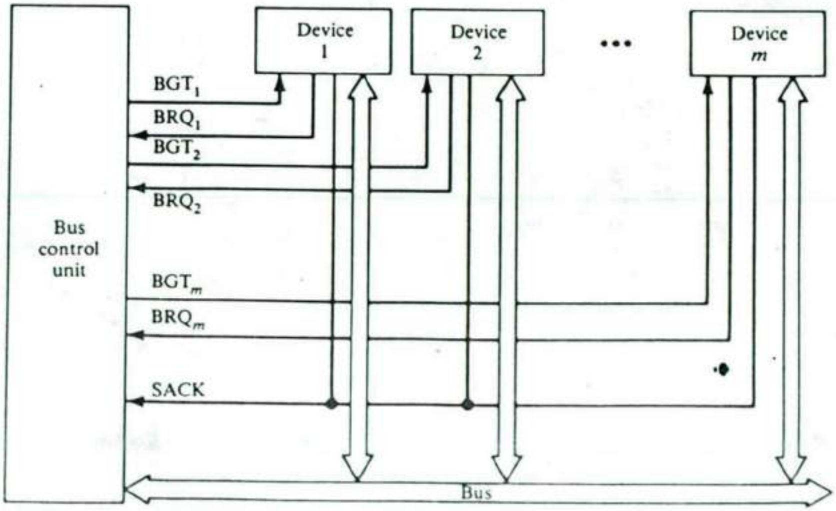


Figure 7.21 Independent request implementation of a system bus.

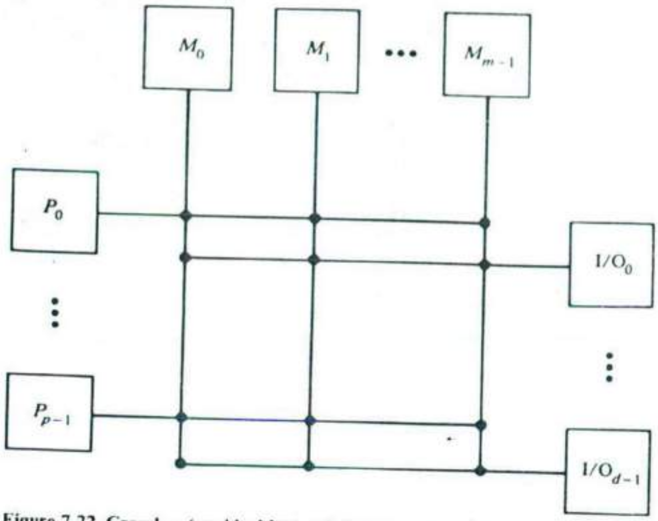


Figure 7.22 Crossbar (nonblocking) switch system organization for multiprocessors. (Courtesy of *ACM Computing Surveys*, Enslow 1977.)

The crossbar switch possesses complete connectivity with respect to the memory modules because there is a separate bus associated with each memory module. Therefore, the maximum number of transfers that can take place simultaneously is limited by the number of memory modules and the bandwidth-speed product of the buses rather than by the number of paths available.

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 transfers for all memory units. 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 the 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.

In a crossbar switch or multiported device, conflicts occur when two or more concurrent requests are made to the same destination device. In the following discussion, we assume that there are 16 destination devices (memory modules) and 16 requestors (processors). The implementation to be described can also be used for a processor to device connection. Figure 7.23 shows an example functional design of a crossbar switch element or multiported memory for one module. The switch consists of arbitration and multiplexer modules. Each processor generates a memory module request signal (REQ) to the arbitration unit, which selects the

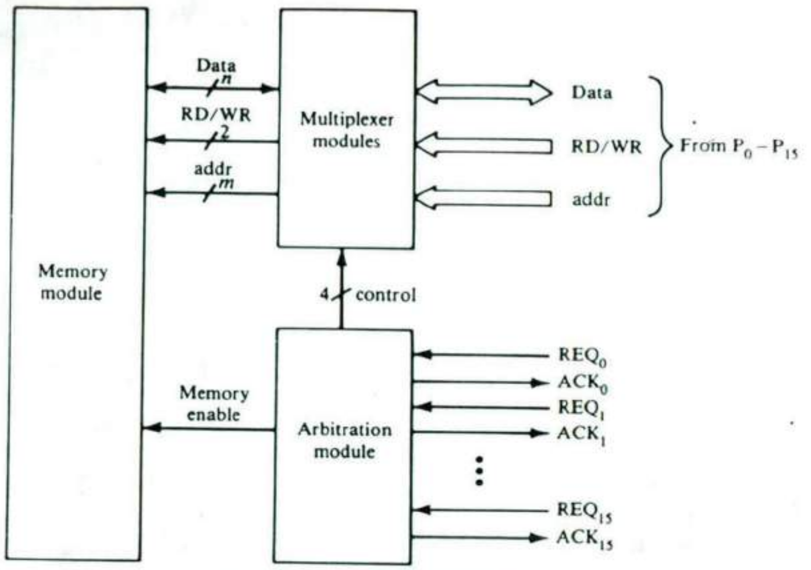


Figure 7.23 Functional structure of a crosspoint in a crossbar network.

processor with the highest priority. The selection is accomplished with a priority encoder. The arbitration module returns an acknowledge signal (ACK) to the selected processor. After the processor receives the ACK, it initiates its memory operation.

The multiplexer module multiplexes data, addresses of words within the module, and control signals from the processor to the memory module using a 16-to-1 multiplexer. The multiplexer is controlled by the encoded number of the selected processor. This code was generated by the priority encoder within the arbitration module.

Such a scheme was used to implement the processor-memory switch for the C.mmp, which has 16 processors and 16 memory modules. The switch consists of 16 sets of cross points from one processor port to the 16 memory ports, and another 16 sets of cross points from one memory port to the 16 processor ports. Theoretically, expansion of the system is limited only by the size of the switch matrix, which can often be modularly expanded within initial design or other engineering limitations. One effect of VLSI on the crossbar interconnection system is the feasibility of designing crossbar matrices for a larger capacity than initially required and equipping them only for the present requirements. Expansion would then be facilitated, since all that is required is the addition of the missing cross points.

In order to provide the flexibility required in access to the input-output devices, a natural extension of the crossbar switch concept is to use a similar switch on the device side of the I/O processor or channel, as shown in Figure 7.24. The

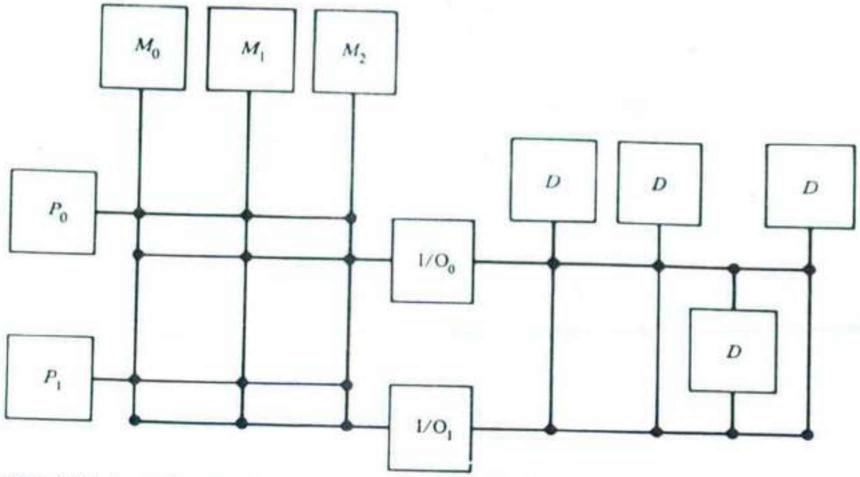


Figure 7.24 A crossbar organization for inter-processor-memory-I/O connections.

hardware required for the implementation is quite different and not nearly so complex because controllers and devices are normally designed to recognize their own unique addresses. The effect is the same as if there were a primary bus associated with each I/O channel and crossbuses for each controller or device.

The crossbar switch has the potential for the highest bandwidth and system efficiency. However, because of its complexity and cost, it may not be cost-effective for a large multiprocessor system. The reliability of the switch is problematic; however, it can be improved by segmentation and redundancy within the switch. In general, it is normally quite easy to partition the system to logically isolate malfunctioning units. There are a number of examples of systems utilizing the crossbar interconnection systems. Some of these are the C.mmp and the S-1 multiprocessor systems, which are to be discussed in Chapter 9.

If the control, switching, and priority arbitration logic that is distributed throughout the crossbar switch matrix is distributed at the interfaces to the memory modules, a multiport memory system is the result, as the example shows in Figure 7.25. This system organization is well suited to both uni- and multiprocessor system organizations and is used in both. The method often utilized to resolve memory-access conflicts is to assign permanently designated priorities at each memory port. The system can then be configured as necessary at each installation to provide the appropriate priority access to various memory modules for each functional unit, as shown in Figure 7.26. Except for the priority associated with each, all of the ports are usually electrically and operationally identical. In fact, the ports are often merely a row of identical cable connectors, and electrically it makes no difference whether an I/O or central processor is attached.

The flexibility possible in configuring the system also makes it possible to designate portions of memory as private to certain processors, I/O units, or com-

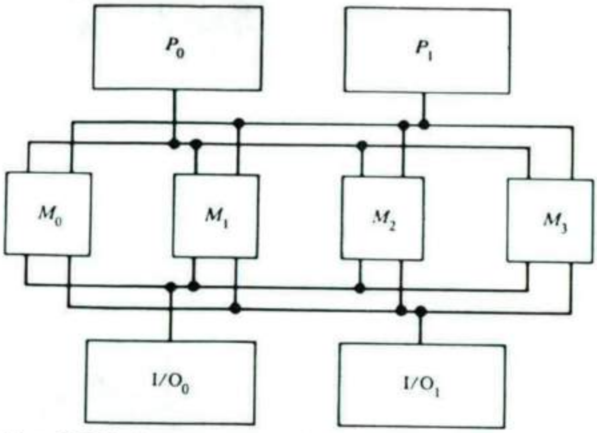


Figure 7.25 Multiport memory organization without fixed priority assignment. (Courtesy of *ACM Computing Surveys*, Enslow 1977.)

binations thereof, as shown in Figure 7.27. In this figure, memory modules M_0 and M_3 are private to processors P_0 and P_1, respectively. This type of system organization can have definite advantages in increasing protection against unauthorized access and may also permit the storage of recovery routines in memory areas that are not susceptible to modification by other processors; however, there are also serious disadvantages in system recovery if the other processors are not able to access control and status information in a memory block associated with a faulty processor.

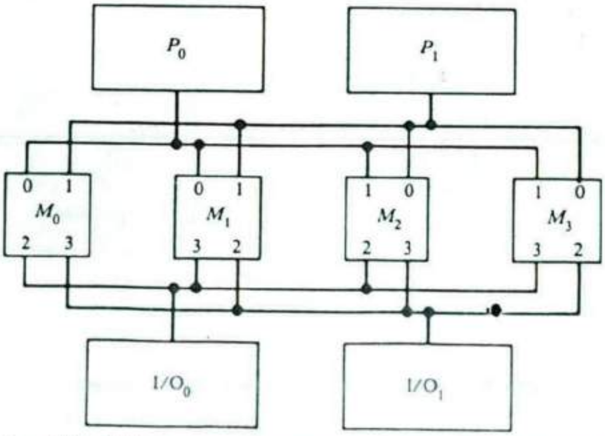


Figure 7.26 Multiport-memory system with assignment of port priorities. (Courtesy of *ACM Computing Surveys*, Enslow 1977.)

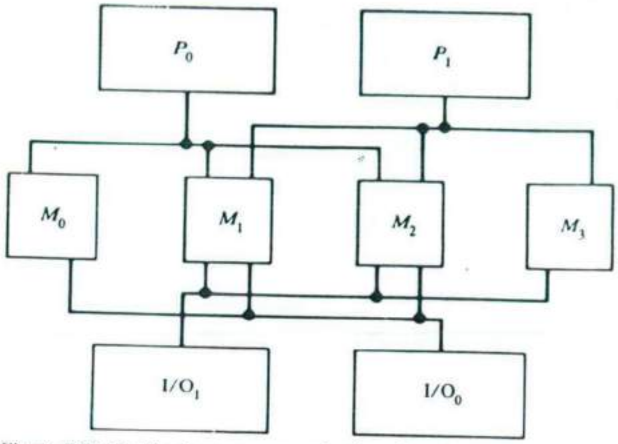


Figure 7.27 Multiport memory organization with private memories. (Courtesy of *ACM Computing Surveys*, Enslow 1977.)

The multiport memory system organization also can support nonblocking access to the memory if a *full-connected* topology is utilized. Since each word access is a separate operation, it also permits the exploitation of interleaved memory addresses for access by a single processor. However, for multiple processors, interleaving may actually degrade memory performance by increasing the number of memory-access conflicts that occur as all processors cycle through all memory following a sequence of consecutive addresses. Interleaving also results in the effective loss of more than one module of memory when there is a failure. There are a number of examples of multiport memory systems such as the Univac 1100/90 and IBM System 370/168 to be discussed in Chapter 9.

It is very difficult to justify the use of a crossbar switch or multiport memories for large multiprocessing systems. The absence of a switch with reasonable cost and performance is one of the reasons that has prevented the growth of large multimicroprocessor systems. The high cost of the switch may be circumvented by using a switch with a slightly restricted number of possible permutations. In the next subsection, we discuss some multistage networks for multiprocessors. These networks are far less expensive than full crossbars or multiport memories for large multiprocessing systems. Moreover, the multistage networks are modular and easy to control. The modularity permits incremental expansion and repairability. A comparison of the three multiprocessor interconnection structures is given in Table 7.2.

7.2.3 Multistage Networks for Multiprocessors

In order to design multistage networks, we need to understand the basic principles involved in the construction and control of simple crossbar switches. Consider the 2×2 crossbar switch shown in Figure 7.28. This 2×2 switch has the capability

Table 7.2 Comparison of three multiprocessor hardware organizations

Multiprocessors with time shared bus:

1. Lowest overall system cost for hardware and least complex.
2. Very easy to physically modify the hardware system configuration by adding or removing functional units.
3. Overall system capacity limited by the bus transfer rate. Failure of the bus is a catastrophic system failure.
4. Expanding the system by the addition of functional units may degrade overall system performance (throughput).
5. The system efficiency attainable is the lowest of all three basic interconnection systems.
6. This organization is usually appropriate for smaller systems only.

Multiprocessors with crossbar switch:

1. This is the most complex interconnection system. There is a potential for the highest total transfer rate.
2. The functional units are the simplest and cheapest since the control and switching logic is in the switch.
3. Because a basic switching matrix is required to assemble any functional units into a working configuration, this organization is usually cost-effective for multiprocessors only.
4. Systems expansion (addition of functional units) usually improves overall performance. There is the highest potential for system efficiency such as for system expansion without reprogramming of the operating system.
5. Theoretically, expansion of the system is limited only by the size of the switch matrix, which can often be modularly expanded within initial design or other engineering limitations.
6. The reliability of the switch, and therefore the system, can be improved by segmentation and/or redundancy within the switch.

Multiprocessors with multiport memory:

1. Requires the most expensive memory units since most of the control and switching circuitry is included in the memory unit.
2. The characteristics of the functional units permit a relatively low cost uniprocessor to be assembled from them.
3. There is a potential for a very high total transfer rate in the overall system.
4. The size and configuration options possible are determined (limited) by the number and type of memory ports available; this design decision is made quite early in the overall design process and is difficult to modify.
5. A large number of cables and connectors are required.

of connecting the input A to either the output labeled 0 or the output labeled 1, depending on the value of some control bit c_A of the input A. If $c_A = 0$, the input is connected to the upper output, and if $c_A = 1$, 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.

The 2×2 switch shown in Figure 7.28 is not buffered. In such a switch, the performance may be limited by the switch setup time which is experienced each time a rejected request is resubmitted. To improve the performance, buffers can be inserted within the switch, as shown in Figure 7.29. Such a switch has also been shown to be effective for packet switching when used in a multistage network.

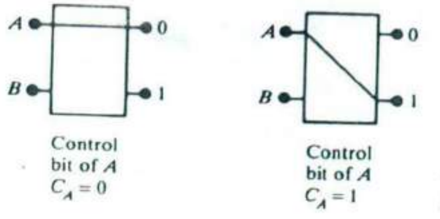


Figure 7.28 A 2 × 2 crossbar switch.

It is straightforward to construct a 1×2^n demultiplexer using the above described 2×2 module. This is accomplished by constructing a binary tree of these modules, as shown in Figure 7.30, for a 1×8 demultiplexer tree. The destinations are marked in binary. If the source A is required to connect to destination $(d_2 d_1 d_0)_2$, then the root node is controlled by bit d_2, the modules in the second stage are controlled by bit d_1, and the modules in the last stage are controlled by bit d_0. It is clear that A can be connected to any one of the eight output terminals. It is also obvious that input B can be switched to any one of the eight outputs. The method of constructing the 1×2^n demultiplexer tree can be extended to build a $2^n \times 2^n$ multistage network. Below we extend the tree network to devise a general multistage network called a banyan network.

A *banyan network* can be roughly described as a partially ordered graph divided into distinct levels. Nodes with no arcs fanning out of them are called *base* nodes and those with no arcs fanning into them are called *apex* nodes. The *fanout* f of a node is the number of arcs fanning out from the node. The *spread*

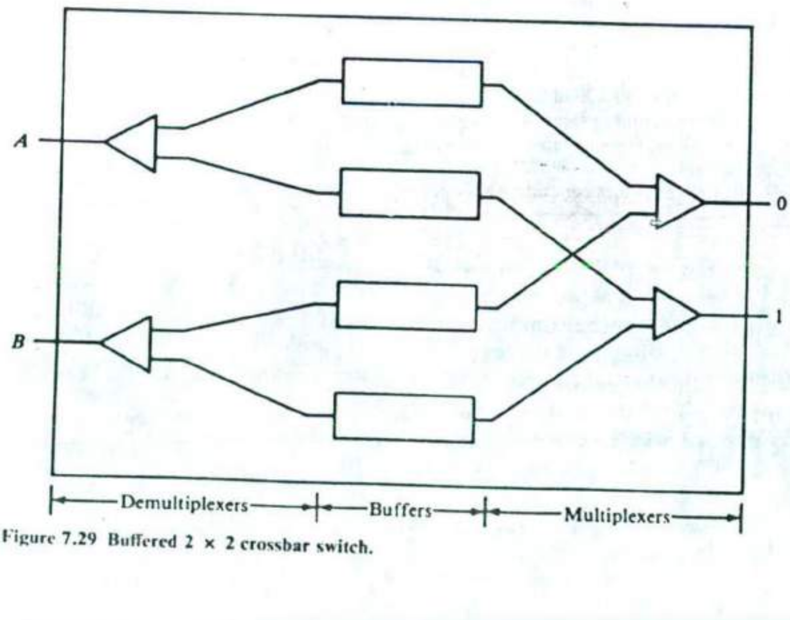


Figure 7.29 Buffered 2 × 2 crossbar switch.

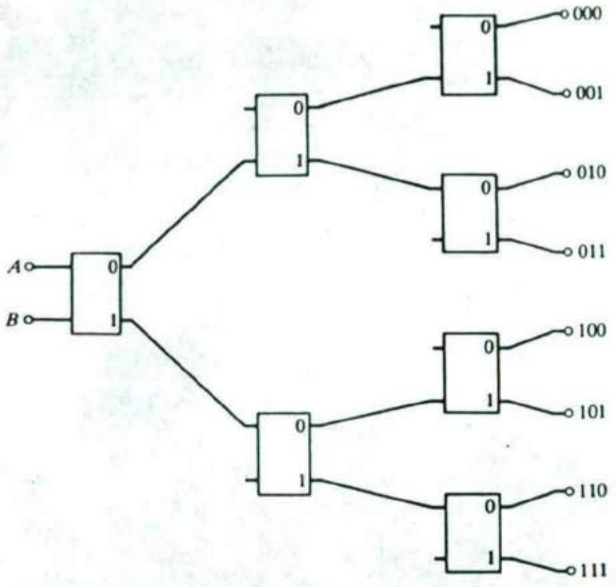
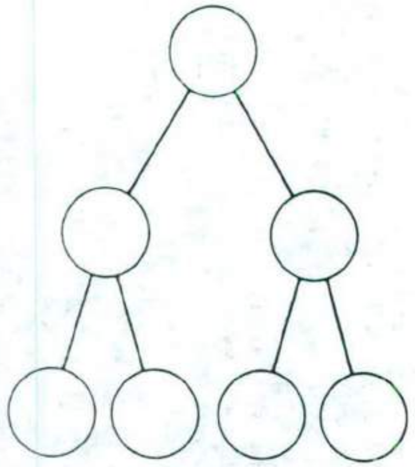


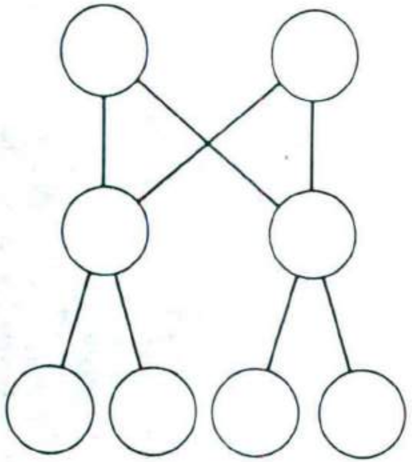
Figure 7.30 1-by-8 demultiplexer implemented with 2 × 2 switch boxes.

s of a node is the number of arcs fanning into it. An (f, s, l) banyan network can thus be described as a partially ordered graph with l levels in which there is exactly one path from every base to every apex node. The fanout of each nonbase node is f and the spread of each nonapex node is s. Each node of the graph is an $s \times f$ crossbar switch.

The banyan network can be derived from a uniform tree with fanout f. We illustrate the derivation of a (2, 2, 2) banyan network from the two-level binary tree shown in Figure 7.31*a*. Since the spread is 2, it means that two arcs should be fanning into each nonroot node. Therefore, we replicate the rest to have s copies of the root and attach the root to the next level nodes, as shown in Figure 7.31*b*. To make the spread of the leaf nodes equal to two, replicate the top two levels (interleaving the second level nodes). Join the second level nodes to the leaf nodes to make the fanout of the second level nodes equal to 2 and the spread of the leaf nodes equal to 2. This completes the derivation of the (2, 2, 2) banyan network and is shown in Figure 7.31*c*.

A banyan network has the advantage of providing a complete interconnection of one set of n devices to another set of n devices at a cost in switching circuitry that grows as $n \log n$. A crossbar switch, by contrast, grows as n^2. In general, an (f, s, l) banyan network can be defined as l recursions on an $s \times f$ crossbar switch. A cross point is thus a banyan of height 1. Hence different topologies of banyan networks can be implemented for multiprocessors. However, more studies need


(a) A tree, fanout 2 (2 level)


(b) Replication of the root – spread 2

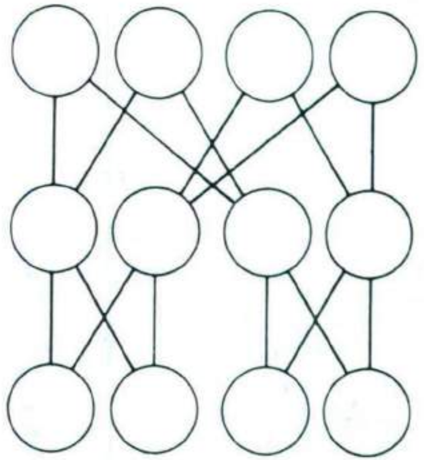

(c) Replication of the next level

Figure 7.31 Formation of a (2, 2, 2) Banyan graph.

to be performed to evaluate the cost effectiveness of these topologies under different multiprocessor work loads. Below, we discuss an implementation of a special type of banyan network called the delta network, which has been proposed for multiprocessors by Patel (1981).

Let an $a \times b$ crossbar module have the capability to connect any one of its a inputs to any one of the b outputs, where the outputs are labeled $0, 1, \ldots, b - 1$. An input terminal is connected to the output labeled d if the control digit supplied by the input is d, where d is a base-b digit. An $a \times b$ module also arbitrates between conflicting requests by accepting one of the requests and blocking or rejecting others.

A *delta network* is defined as an $a^n \times b^n$ switching network with n stages consisting of $a \times b$ crossbar modules. The interconnection or link patterns between stages is such that there exists a unique path of constant length from any source to any destination. Moreover, the path is digit controlled such that a crossbar module connects an input to one of its b outputs depending on a single base-b digit taken from the destination address. In a delta network, no input or output terminal of any crossbar module is left unconnected.

To systematize the link patterns between stages of the delta network, we define a q-shuffle of qc objects, denoted by S_{q*c} where q and c are some positive integers. S_{q*c} is a permutation of qc indices $\langle 0, 1, 2, \ldots, (qc - 1)\rangle$ and is defined as

$$S_{q*c}(i) = \left(qi + \left\lfloor \frac{i}{c} \right\rfloor\right) \text{mod } qc \tag{7.1}$$

for $0 \le i \le qc - 1$. An alternative $S_{q*c}(i)$ can be expressed as

$$S_{q*c}(i) = \begin{cases} qi \bmod (qc - 1) & \text{for } 0 \le i < qc - 1 \\ i & \text{for } i = qc - 1 \end{cases} \tag{7.2}$$

A q-shuffle of qc playing cards can be viewed as follows. Divide the deck of qc cards into q piles of c cards each: top c cards $\langle 0, 1, \ldots, (c - 1)\rangle$ in the first pile, next c cards $\langle c, c + 1, \ldots, (2c - 1)\rangle$ in the second pile, and so on. Now pick the cards one at a time from the top of each pile, the first card from the top of pile one, the second card from the top of pile two, and so on in a circular fashion until all the piles of cards are exhausted. This new order of cards represents an S_{q*c} permutation of the previous order. Figure 7.32 illustrates an example of a 4 shuffle of 12 indices, namely S_{4*3}. From the above description, it is clear that a 2 shuffle is the well-known perfect shuffle discussed in Chapter 5.

In order to construct an $a^n \times b^n$ delta network, we use the a shuffle as the link pattern between every two consecutive stages of the network. If the destination D is expressed in a base-b system as $(d_{n-1}d_{n-2}\cdots d_1d_0)_b$, where $D = \sum_{0 \le j < n} d_j b^j$ and $0 \le d_i < b$, then the base-b digit d_i controls the crossbar modules of stage $(n - i)$. The a-shuffle function is used to convert the outputs of a stage to the inputs of the next stage, where the inputs and outputs are numbered $0, 1, 2, \ldots,$ starting at the

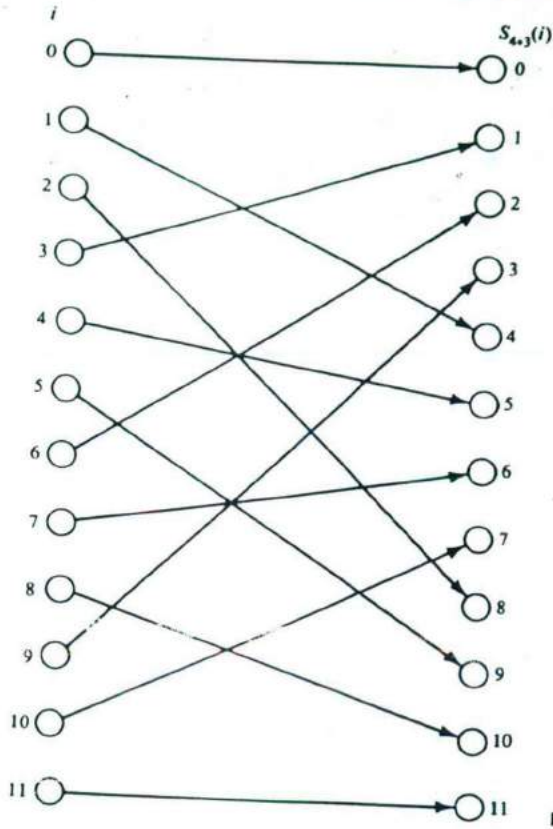


Figure 7.32 The 4 shuffle of 12 objects.

top. Figure 7.33 shows a general $a^n \times b^n$ delta network which has a^n sources and b^n destinations. Numbering the stages of the network as 1, 2, . . . , n, starting at the source side of the network requires that there be a^{n-1} crossbar modules in the first stage. The first stage then has $a^{n-1}b$ output terminals. This implies that stage two must have $a^{n-1}b$ input terminals, which requires $a^{n-2}b$ crossbar modules in the second stage. In general, the ith stage has $a^{n-i}b^{i-1}$ crossbar modules of size $a \times b$. Thus, the total number of $a \times b$ crossbar modules required in an $a^n \times b^n$ delta network can be found as $(a^n - n^n)/(a - b)$, for $a \neq b$ and nb^{n-1} for $a = b$. Two delta networks, one $4^2 \times 3^2$ and the other $2^3 \times 2^3$, derived from Figure 7.33 are shown in Figures 7.34 and 7.35, respectively; the interstage link patterns are 4 shuffle and 2 shuffle, respectively. Note that the destinations in Figures 7.34 and 7.35 are labeled in bases 3 and 2, respectively. It has been shown that the a-shuffle link pattern used between adjacent stages allows a source to connect to any destination by using the destination-digit control of each $a \times b$ crossbar module.

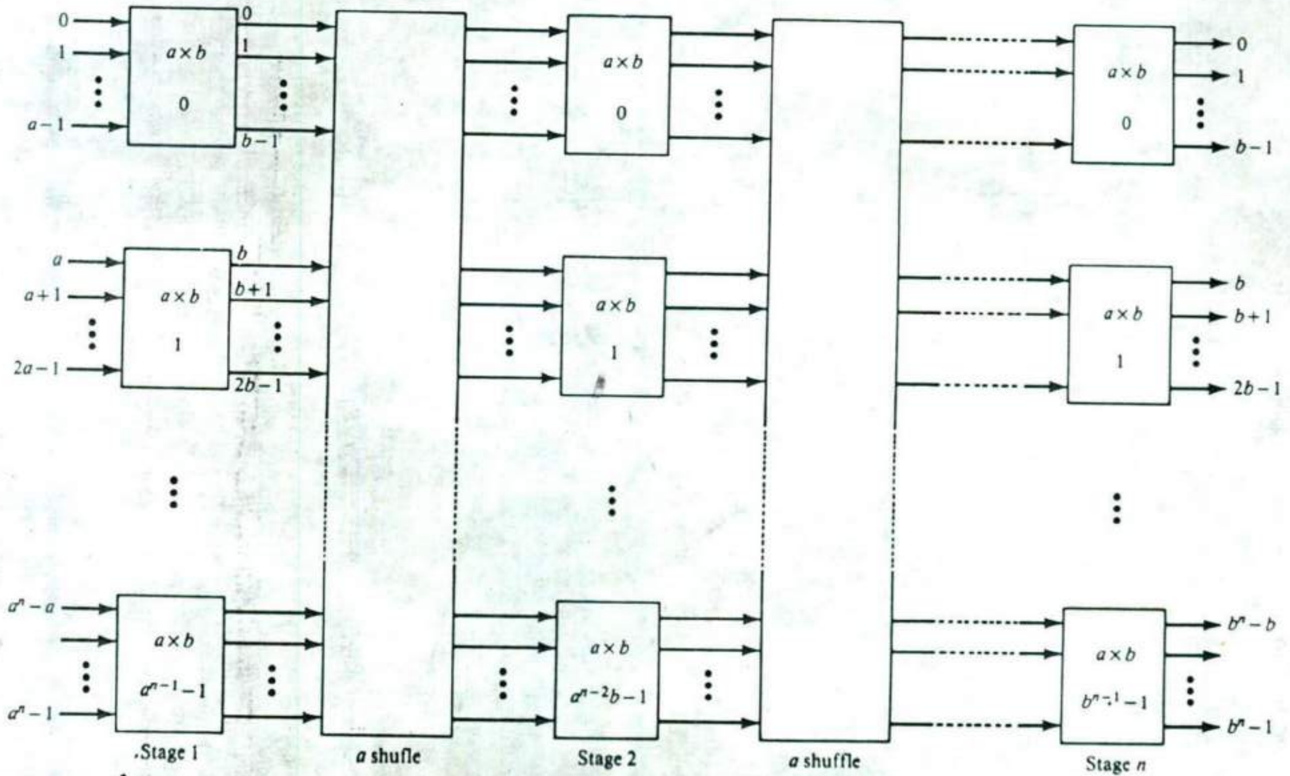


Figure 7.33 An $a^n \times b^n$ Delta network. (Courtesy of *IEEE Trans. Computers*, Patel 1981.)

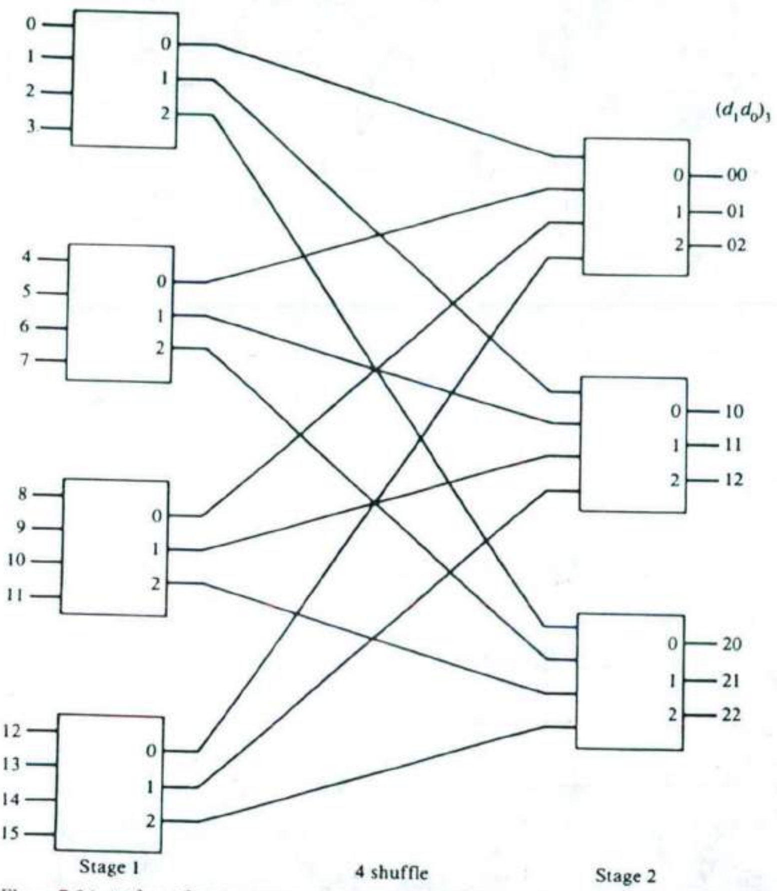


Figure 7.34 A $4^2 \times 3^2$ delta network. (Courtesy of *IEEE Trans. Computers*, Patel 1981.)

Note that the network of Figure 7.35 does not allow an identity permutation, which is useful if, say, memory module *i* is a "favorite" or home module of processor *i*. Therefore, identity permutation allows most of the memory references to be made without conflict. A simple renaming of the inputs of Figure 7.35 will permit an identity permutation. This is shown in Figure 7.36. In this case, if all 2×2 switches were in the straight position, then an identity permutation is generated.

In general, *b* is a power of 2 and *a* is very small, usually between 1 and 4. Figure 7.37 illustrates the functional block diagram of a 2×2 crossbar module. All single lines in the figure are 1-bit lines. The double lines on the INFO box represent address lines, incoming and outgoing data lines, and a read-write control line. The data lines may or may not be bidirectional. The function of the INFO box

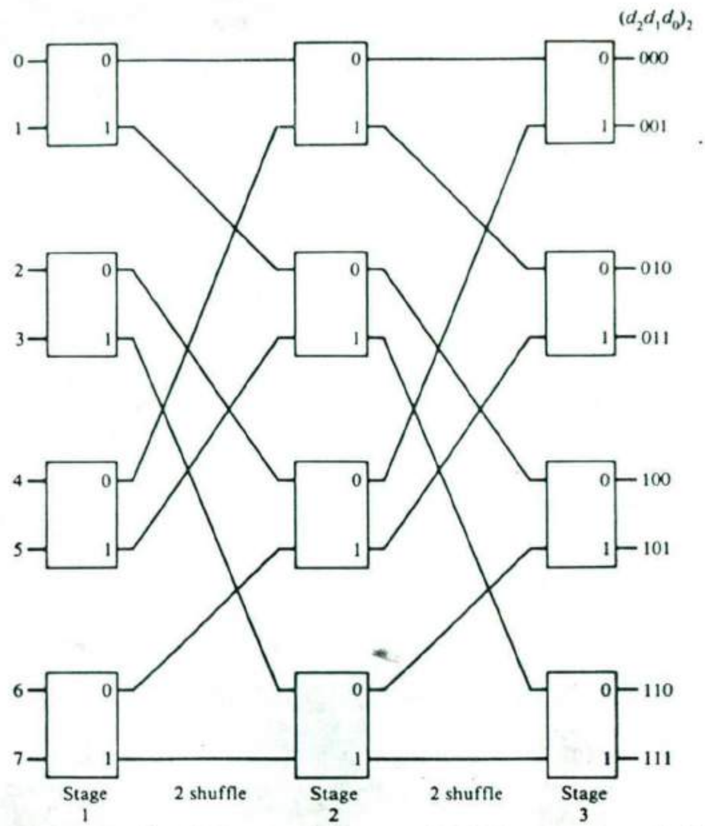


Figure 7.35 A $2^3 \times 2^3$ delta network. (Courtesy of *IEEE Trans. Computers*, Patel 1981.)

is that of a simple 2×2 crossbar; if the input X is 1, then a cross connection exists, and if X is 0, then a straight connection exists.

The function of the control box is to generate the signal X and provide arbitration. A request exists at an input port if the corresponding request line is 1. The destination digit provides the nature of the request; a 0 for the connection to the upper output port and a 1 for the lower port. In case of conflict, the request r_0 is given the priority and a busy signal $b_1 = 1$ is supplied to the lower input port. A busy signal is eventually transmitted to the source which originated the blocked request. The logic equations for all the labeled signals are given with the block diagram. For the INFO box, the equations are given for left to right direction. The parallel generation of X and $\overline{X}$ reduces one gate level.

The operation of a $2^n \times 2^n$ delta network using the above described 2×2 modules is as follows: Recall that there are n stages in this network. All processors requiring memory access must submit their requests at the same time by placing a

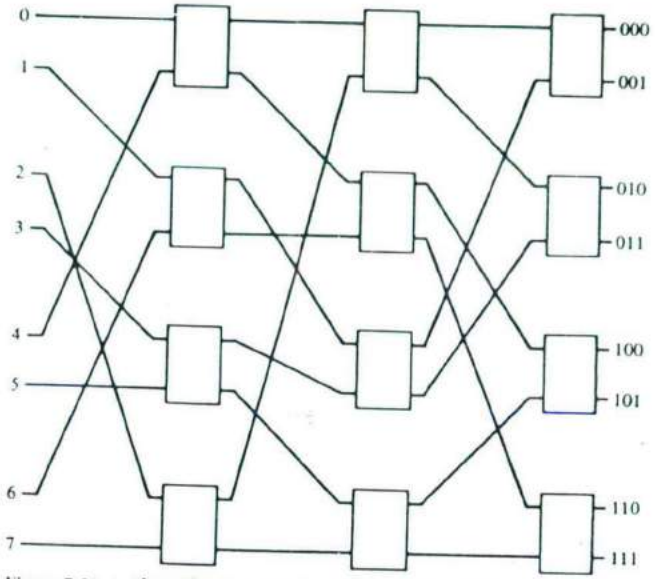


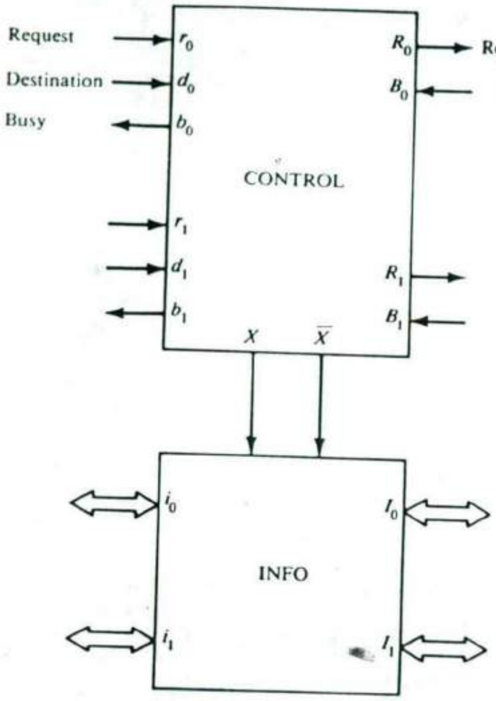
Figure 7.36 A $2^3 \times 2^3$ delta network to allow identity permutation.

1 on the respective request lines. If the busy line is 1, then the processor must resubmit its request. This can be accomplished simply by doing nothing; i.e., continue to hold the request line high. Thus the operation of the implementation described here is synchronous; that is, the requests are issued at fixed intervals at the same time. An asynchronous implementation is preferable if the network has many stages. However, such an implementation would require storage buffers for addresses, data and control in every module and also a complex control module. Thus, the cost of such an implementation might well be excessive.

7.2.4 Performance of Interconnection Networks

In this section, we analyze $p \times m$ crossbar networks and delta networks for processor-memory interconnections. Both networks are analyzed under identical assumptions for the purpose of comparison. We analyze the networks for finding the expected bandwidth given the rate of memory requests. Bandwidth is expressed in the average number of memory requests accepted per cycle. A cycle is defined as the time it takes for a request to propagate through the logic of the network plus the time needed to access a memory word plus the time used to return through the network to the source. We shall not distinguish the read or write cycles in this analysis. The analysis is based on the following assumptions:

1. Each processor generates random and independent requests for a word in memory. The requests are uniformly distributed over all memory modules.



$X = r_0d_0 + r_0d_1 \qquad \overline{X} = r_0\overline{d_0} + \overline{r_0}d_1$

$R_0 = r_0\overline{d_0} + r_1\overline{d_1} \qquad R_1 = r_0d_0 + r_1d_1$

$b_0 = \overline{X}B_0 + XB_1 \qquad b_1 = XB_0 + \overline{X}B_1 + r_0d_0d_1 + r_0d_0d_1$

$I_0 = i_0\overline{X} + i_1X \qquad I_1 = i_0X + i_1\overline{X}$

Figure 7.37 Details of 2 × 2 delta networks. (Courtesy of *IEEE Trans. Computers*, Patel 1981.)

2. At the beginning of every cycle, each processor generates a new request with a probability r. Thus, r is also the average number of requests generated per cycle by each processor.
3. The requests which are blocked (not accepted) are ignored; that is, the requests issued at the next cycle are independent of the requests blocked.

The last assumption is there to simplify the analysis. Although the model does not make proper account of rejections, it still serves a useful purpose. It can be solved exactly and it gives a lower bound on the expected bandwidth. In practice, of course, the rejected requests must be resubmitted during the next cycle or buffered in the module where the conflict occurs; thus the independent request assumption will not hold. Later, the last assumption will be relaxed to improve the model. Moreover, simulation studies performed by many authors for similar problems have shown that the probability of acceptance is only slightly lowered if the third assumption above is omitted. Thus the results of the analysis are fairly reliable and they provide a good measure for comparing different networks.

Analysis of crossbars Assume a crossbar of size $p \times m$ that is, p processors (sources) and m memory modules (destinations). In a full crossbar, two requests are in conflict if, and only if, the requests are to the same memory module. Therefore, in essence we are analyzing memory conflicts rather than network conflicts. Recall that r is the probability that a processor generates a request during a cycle. Let $q(i)$ be the probability that i requests arrive during one cycle. Then

$$q(i) = \binom{p}{i} r^i (1 - r)^{p-i} \tag{7.3}$$

where $\binom{p}{i}$ is the binomial coefficient.

Let $E(i)$ be the expected number of requests accepted by the $p \times m$ crossbar during a cycle, given that i requests arrived in the cycle. To evaluate $E(i)$, we know that from combinations the number of ways that i random requests can map to m distinct memory modules is m^i. Suppose now that a particular memory module is not requested. Then the number of ways to map i requests to the remaining $(m - 1)$ modules is $(m - 1)^i$. Thus, $m^i - (m - 1)^i$ is the number of maps in which a particular module is always requested. Thus the probability that a particular module is requested is $[m^i - (m - 1)^i]/m^i$. For every memory module, if it is requested, it means one request is accepted by the network for that module. Therefore, the expected number of acceptances, given i requests, is

$$E(i) = \frac{m^i - (m - 1)^i}{m^i} \cdot m = \left[1 - \left(\frac{m - 1}{m}\right)^i\right] m$$

Thus the expected bandwidth $B(p, m)$, that is, the average number of requests accepted per cycle, is

$$B(p, m) = \sum_{0 \le i \le p} E(i) \cdot q(i)$$

which simplifies to:

$$B(p, m) = m - m\left(1 - \frac{r}{m}\right)^p \tag{7.4}$$

Let us define the ratio of expected bandwidth to the expected number of requests generated per cycle as the probability of acceptance P_A. P_A is the probability that an arbitrary request will be accepted. Therefore

$$P_A = \frac{B(p, m)}{rp} = \frac{m}{rp} - \frac{m}{rp}\left(1 - \frac{r}{m}\right)^p \tag{7.5}$$

It is interesting to note the limiting values of $B(p, m)$ and P_A as p and m grow very large. Let $k = p/m$; then

$$\lim_{m \to \infty} \left(1 - \frac{r}{m}\right)^{km} = e^{-rk}$$

Thus for very large values of p and m

$$B(p, m) \simeq m(1 - e^{-rp/m}) \tag{7.6}$$

$$P_A \simeq \frac{m}{rp}(1 - e^{-rp/m}) \tag{7.7}$$

The above approximations are good within 1 percent of actual value when p and m are greater than 30 and within 5 percent when p and $m \geq 8$. Note that for a fixed ratio p/m, the bandwidth of Eq. 7.6 increases linearly with m.

Equation 7.5 was derived under the hypothesis of independent requests. In reality, however, a rejected request is not simply discarded but resubmitted during the next cycle, thereby increasing the request rate. We will not derive a detailed model that takes into account the exact behavior of a system with rejected requests. Instead, we will approximate the behavior by means of a simplifying assumption that makes the improved model more tractable and is still a bound on the exact model. We assume that the resubmitted request addresses the modules uniformly. A processor can be in one of two states, A or W. W is the state corresponding to a *wasted* cycle due to a rejected request. A is an *active* cycle during which a processor may issue a new request. The behavior of any one processor is described by the Markov graph of Figure 7.38.

Let q_A and q_W be the steady state probabilities that the processor is in state A and W, respectively. Solving for q_A and q_W, we obtain

$$q_A = \frac{P_A}{P_A + r(1 - P_A)} \tag{7.8}$$

and

$$q_W = 1 - q_A$$

The request rate r should be defined more precisely as the rate assuming conflict-free accesses. We refer to r as the static request rate. However, memory

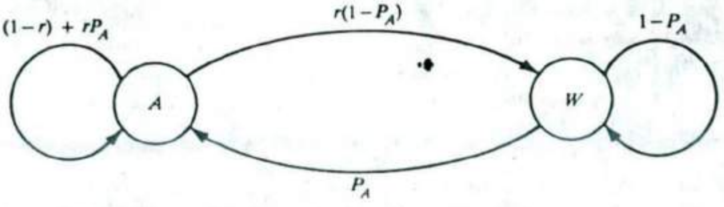


Figure 7.38 Markov graph for computing dynamic request rate r'.

requests are also made during each wasted cycle. Therefore, the memory modules encounter a dynamic request rate r' which is actually higher than the static request rate because of memory conflicts. r' can be obtained from the Markov graph as

$$r' = rq_A + q_W = \frac{r}{r + P_A(1 - r)} \tag{7.9}$$

Therefore Eq. 7.5 becomes

$$P_A = \frac{m}{r'p}\left[1 - \left(1 - \frac{r'}{m}\right)^p\right] \tag{7.10}$$

Equations 7.9 and 7.10 define an iterative process by which we can compute P_A for a given m, p, and r. r' can be initialized to r for the iterative process.

Thus, P_A is a measure of the wasted cycles of blocked requests. A higher P_A indicates a lower number of wasted cycles and a lower P_A indicates higher number of wasted cycles. The average number of wasted cycles $\overline{w}$ per request can be computed if we note that a request that is rejected i times consecutively before it is accepted waits for i cycles:

$$\begin{aligned}\overline{w} &= \sum_{i=1}^{\infty} i(1 - P_A)^i P_A \\ &= \frac{1 - P_A}{P_A}\end{aligned} \tag{7.11}$$

Note that the reassignment of a rejected request (according to a uniform distribution among the memory modules) causes the model to overestimate the bandwidth. This is assumed to simplify the model. In practice, the requests are queued at the memory module and serviced by the module in a first-come-first-served fashion.

Analysis of delta networks Assume a delta network of size $a^n \times b^n$ constructed from $a \times b$ crossbar modules. Thus, there are a^n processors connected to b^n memory modules. We apply the result of Eq. 7.4 for a $p \times m$ crossbar to an $a \times b$ crossbar and then extend the analysis for the complete delta network. However, to apply Eq. 7.4 to any $a \times b$ crossbar module, we must first satisfy the assumptions of the analysis. We show below that the independent request assumption holds for every $a \times b$ module in a delta network.

Each stage of the delta network is controlled by a distinct destination digit (in base b) for the setting of individual $a \times b$ switches. Since the destinations are independent and uniformly distributed, so are the destination digits. Thus, for example, in some arbitrary stage i, an $a \times b$ crossbar uses digit d_{n-i} of each request; this digit is not used by any other stage in the network. Moreover, no digit other than d_{n-i} is used by stage i. Therefore, the requests at any $a \times b$ module are independent and uniformly distributed over b different destinations. Thus we can apply the result of Eq. 7.4 to any $a \times b$ module in the delta network.

Given the request rate r at each of the a inputs of an $a \times b$ crossbar module, the expected number of requests that it passes per time unit is obtained by setting $p = a$ and $m = b$ in Eq. 7.4, which is

$$b - b\left(1 - \frac{r}{b}\right)^a$$

Dividing the above expression by the number of output lines of the $a \times b$ module gives us the rate of requests on any one of b output lines:

$$1 - \left(1 - \frac{r}{b}\right)^a \tag{7.12}$$

Thus for any stage of a delta network, the output rate of requests, r_{out}, is a function of its input rate, r_{in}, and is given by

$$r_{\text{out}} = 1 - \left(1 - \frac{r_{\text{in}}}{b}\right)^a \tag{7.13}$$

Since the output rate of a stage is the input rate of the next stage, one can recursively evaluate the output rate of any stage starting at stage 1. In particular, the output rate of the final stage n determines the bandwidth of a delta network; that is, the number of requests accepted per cycle.

Let us define r_i to be the rate of requests on an output line of stage i. Then the following equations determine the bandwidth $B(a^n, b^n)$ of an $a^n \times b^n$ delta network, given r, the rate of requests generated by each processor:

$$B(a^n, b^n) = b^n r_n \tag{7.14}$$

where $$r_i = 1 - \left(1 - \frac{r_{i-1}}{b}\right)^a \quad \text{and} \quad r_0 = r$$

The probability that a request will be accepted is

$$P_A = \frac{b^n r_n}{a^n r} \tag{7.15}$$

Since we do not have a closed-form solution for the bandwidth of delta networks (Eq. 7.14), we cannot directly compare the bandwidths of crossbar (Eq. 7.4) and delta networks. However, we present plots that compare the performance of crossbar and delta networks using Eqs. 7.5 and 7.15. Figure 7.39 shows the probability of acceptance, P_A, for $2^n \times 2^n$ and $4^n \times 4^n$ delta networks and $p \times p$ crossbar, when the request rate for each processor is $r = 1$. The curve marked delta -2 is for delta networks using 2×2 switches and delta -4 for delta networks using 4×4 switches. Notice that P_A for crossbar approaches a constant value as was predicted by Eq. 7.7. P_A for delta networks continues to fall as p grows. The model refinement developed for the crossbar switch can also be applied to delta networks iteratively and is left as an exercise for the reader.

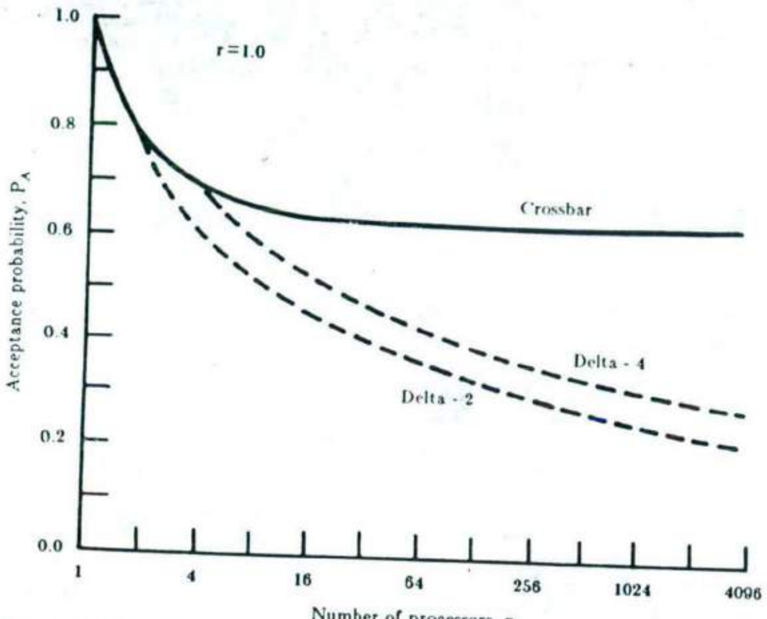


Figure 7.39 Probability of acceptance of $p \times p$ networks. (Courtesy of *IEEE Trans. Computers*, Patel 1981.)

7.3 PARALLEL MEMORY ORGANIZATIONS

This section addresses techniques for designing parallel memories for loosely and tightly coupled multiprocessors. The interleaving method presented is an extension of techniques applied to memory configurations for pipeline and vector processors. Many commercial multiprocessor systems are tightly coupled, where each processor has a private cache. The presence of multiple private caches introduces the problem of cache coherence or multicache consistency. Various solutions to cache-coherence problems are presented. Finally, we describe some simple models to evaluate the effectiveness of the various memory configurations.

7.3.1 Interleaved Memory Configurations

Low-order interleaving of memory modules is advantageous in multiprocessing systems when the address spaces 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 M_i 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, $\{M_i\} \cap \{M_j\} = \phi$, for $i \neq j$. The home-memory organization for multiprocessors has an additional architectural advantage beyond the reduction in memory interference.

The processor-memory interconnection network (PMIN) of a multiprocessor system may be expensive, slow, and complicated. Figure 7.40 is 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

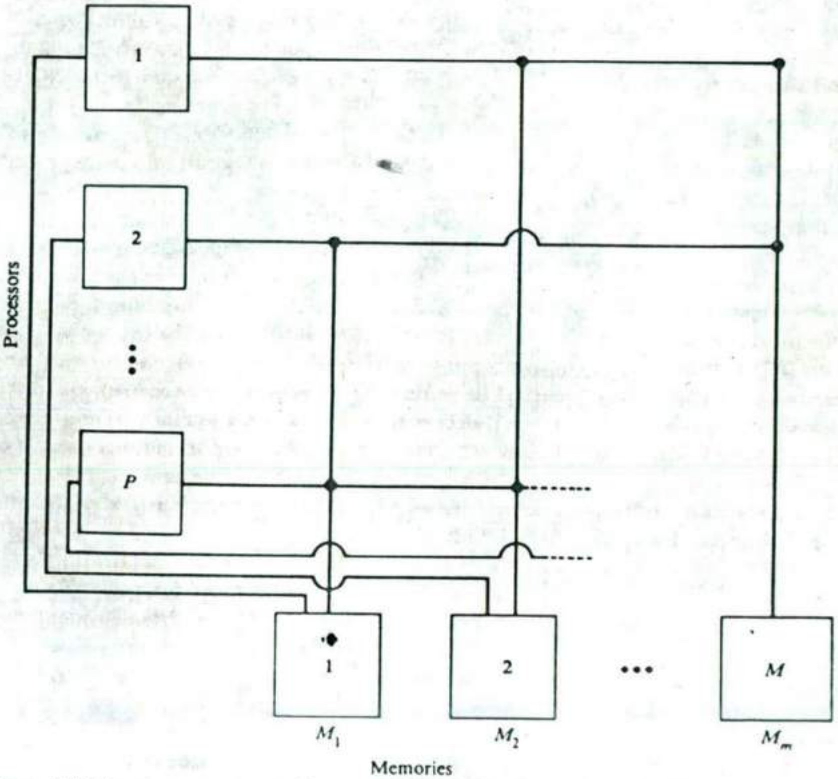


Figure 7.40 Home memory concept. (Courtesy of *IEEE Trans. Computers*, Smith 1978.)

permits enhanced access by each processor to its home memory by frequently avoiding switching time through the 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 becomes less critical and substantial cost savings may also be possible. This concept was applied in the design of Cm*. Home-memory organizations also permit significant gains in the reliability of system operation. A single memory failure when using a home-memory organization generally disables only a small subset of the processors currently running, that is, those with information in the failing memory.

The concurrent (C) access memory configuration described for pipeline processors can also be used for 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 processors encounter memory conflicts as well as transmission delays. To reduce these effects, a private cache is usually associated with each processor in a multiprocessor so that most of the referenced data and instructions can be found in the cache. However, the data bus width may affect the cost and transfer time of a block of data. For example, if each module has a data-transfer path of one 8-byte wide word and there are four memory modules on a bank of the C access configuration, then 32 bytes may be transferred in little more than the time required for one main memory cycle. Therefore, in a computer with a cache block size of 32 bytes only a little over one main memory cycle would be required to fetch all 32 bytes.

It should be noted that the cache is not usually interleaved, hence the arrival of the four words of information from main memory must be staggered slightly so as to allow the cache to accept each 8 bytes separately. A block of cache locations consists of contiguous memory locations. If the memory modules are interleaved on the low-order bits of the addresses, the block transfer will be inefficient. This occurs because consecutive memory locations of the block are in consecutive modules, and the delay incurred in setting up a path in the processor memory interconnection network for each access to a consecutive location of the block becomes very significant. Below we describe a more general parallel memory organization which can be used with a wide variety of multiple processor systems and in which memory module interleaving can be taken a step further to permit block transfer at even the cache bandwidth.

A two-dimensional memory organization called the L-M organization and arranged as l C-access configurations, each dimension with m modules, provides more flexibility. The L-M memory organization consists of $N(=2^n)$ identical memory modules arranged such that there are l lines or banks and m modules per line where $l = 2^\beta$ for integers β and n such that $0 \leq \beta \leq n$ $m = 2^{n-\beta}$ so that $lm = 2^n$. Again, a line refers to the address bus common to a set of m modules, as shown in Figure 7.41.

However, as a consequence of line and module sharing, the performance may be degraded. Furthermore, additional memory interference is introduced by this

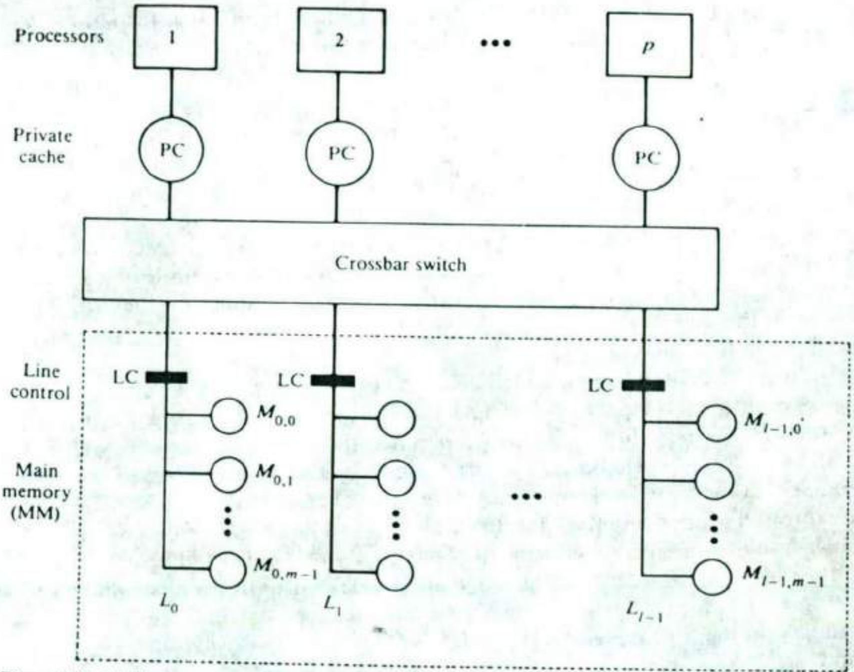


Figure 7.41 Multiprocessor system with private caches. (Courtesy of *IEEE Trans. Computers*, Briggs and Dubois 1983.)

organization. For a degenerate case in which there is one module per line, the memory conflict problem arises when two or more simultaneous memory requests reference the same module, hence the same line. For the two-dimensional memory, a conflict may also occur when a memory request references a busy line or a busy module on a line. A memory configuration characterized by (l, m) is a particular realization of the L-M memory organization.

Let us assume that a *write-back-write-allocate* cache replacement policy is adopted in the following discussion. Let w_b be the probability that the block frame to be replaced was modified. If a cache block frame which has not been modified is to be replaced, it is overwritten with the new block of data. However, a modified block frame that is to be replaced must be written to *main memory* (MM) before a block-read from MM is initiated. In this case, two consecutive transfers are made between the cache and MM. Hence, we assume that each time a cache miss occurs with probability w_b, a block-write to *MM* is required with probability w_b, followed by a block-read from MM.

One method of organizing the cache for block-reads and -writes is to assume that the two consecutive block transfers (one block-write followed by one block-read) are made between a processor and the same line. This assumption will be satisfied if a set-associative cache is used in which all the blocks that map to the same set are stored on the same line. This assumption implies that the number of sets is a multiple of the number of lines. Hence, in this method a cache miss requires the transfer of a $2b$-word block with a probability w_b and the transfer of a b-word with a probability $1 - w_b$.

The L-M memory organization is very cost-effective in matching the bandwidth of a cache memory which initiates block-transfer operations as a result of cache misses. The information is distributed in memory so that each block of a program resides on a line of memory. Consecutive words of a block are stored in consecutive modules on the same line. In this case, a line controller (LC) is associated with each line. The controller typically receives a cache request for a block transfer of size b and thereby issues b internal requests (IR) to consecutive modules on the line. The blocks in the memory are interleaved on the lines so that block i is assigned to modules on line i mod l.

When the main memory is used in the block-transfer mode, the address hold time or cycle of the memory module can be chosen to be equal to the cache cycle time in order to effectively utilize the line. In practice, the address cycle can be made as small as the cache cycle time by incorporating an address latch in each memory module. Let the cache cycle time be the unit time. Therefore, the memory cycle can be expressed as c time units. Also, the modules on a line are interleaved in a particular fashion so that the servicing of two memory requests could be overlapped on the same line. The modules on a line are interleaved so that a block of data of size $b(=2^r)$ is interleaved on consecutive modules on that line. Let line i and module j on that line be referred to as L_i and $M_{i,j}$, respectively, for $0 \leq i \leq l - 1$ and $0 \leq j \leq m - 1$. Then the kth word of the block of data which exists on line i is in module k mod m on that line, for $0 \leq k \leq b - 1$. It is important to note that the first word of a block which exists on line i is in the first module $M_{i,0}$ of that line. If $b < m$, memory modules $M_{i,b}, M_{i,b+1}, \ldots, M_{i,m-1}$, will not be utilized since a block starts in module $M_{i,0}$. Hence, for effective utilization of memory modules, it is assumed that $b \geq m$.

When a block request is accepted by a line i, the line controller at that line issues b successive internal requests to consecutive modules on line i, starting from module $M_{i,0}$. It is assumed that these internal requests are issued at the beginning of every time unit. Therefore, the internal request for the kth word of the block will be issued to module $M_{i,j}$, where $j = k$ mod m, for $0 \leq k \leq b - 1$. It is obvious that this set of b internal requests is not preemptible. Note that if $b > m$ or if the cache is set associative, the $(m + 1)$st internal request is for module $M_{i,0}$. Consequently, the first internal request must be completed by the time the $(m + 1)$st IR is issued. This constraint is satisfied if $c \leq m$.

In order to visualize the concurrent servicing of two memory requests on the same line, we define a time unit, $\langle t, t + 1 \rangle$, as beginning at time t^+ and ending at time $(t + 1)^-$. Therefore, the successive IRs which are generated to modules on a line, in the servicing of a memory request, do not encounter any conflicts. If a memory request is accepted on line i at time t, then the IR for the kth word of a block of size b is initiated at time $t + k$ to module $M_{i,j}$ for $j = k$ mod m and

$0 \leq k \leq b - 1$. Since the memory module cycle is c, module $M_{i,j}$ will be busy in the intervals $\langle t + k, t + k + c \rangle$ for the values of j.

Since b and m are powers of 2 and $b \geq m$, then b/m is an integer ≥ 1. Therefore, each module on a line, i, which accepts a memory request for block transfer at time t receives b/m internal memory requests. In particular, the last IR to module $M_{i,0}$ is made at time $t + (b/m - 1)m = t + b - m$. Thus, the last interval in which module $M_{i,0}$ is busy (during the current block transfer) is

$$\langle t + b - m, t + b - m + c \rangle.$$

After this period, a new block-transfer request which addresses the line can be accepted. Because the current block transfer was initiated at time t, all block-transfer requests arriving at $t + 1, t + 2, \ldots, t + b - m + c - 1$ will find line i busy. Hence, to a memory request, the line is busy for $b - m + c$ time units. We refer to this as the *line service time*, S_i, of line i. However, the *actual service time*, A_i, of a memory request to line i is $b + c - 1$. We refer to the difference $A_i - S_i$ as the *drain time* $D_i = m - 1$. Since D_i is independent of i, we denote D_i by D for all i's. Hence, during S_i, any memory request made to line i will not be accepted but queued. At the end of S_i, the current request proceeds to the next stage where the data transfer is completed in time D. At the same time, a new request made to line i can be accepted. Therefore, the servicing of a memory request can be considered as proceeding in two stages of a pipeline.

7.3.2 Performance Trade-offs in Memory Organizations

In order to evaluate different multiprocessor memory configurations, we introduce a versatile but approximate model. This model is used to illustrate the performance trade-offs in a memory configuration. Each processor has a cache. We assume the interconnection network between the processor and memory to be a crossbar switch. The models developed for the crossbar in Section 7.2 are not applicable since they assume a unit time for the memory cycle. In practice, the main memory cycle c may vary for different memories and should be considered as an attribute of the memory configuration.

General model In each case, the multiprocessor system consists of p homogeneous processors and l banks or lines of interleaved memory. For generality, assume that the first stage of memory service time for modules of bank k is S_k. The drain (second stage service) time of each bank is D_k. Let q_k be the fraction of all references made to bank k. We denote by T the average "think time" spent within the processor nodes before a reference is made to a memory module. Figure 7.42 shows a representation of the model by a closed queueing network. We further represent the model by a state graph shown in Figure 7.43. In this graph, node A denotes an active state of the processor and node W, a waiting state. Node LT represents the state for the first part of the transfer during which the line is kept busy (line service time). The node DT represents the state in which a transfer is completed without holding the line. The state graph does not constitute a Markov chain since each

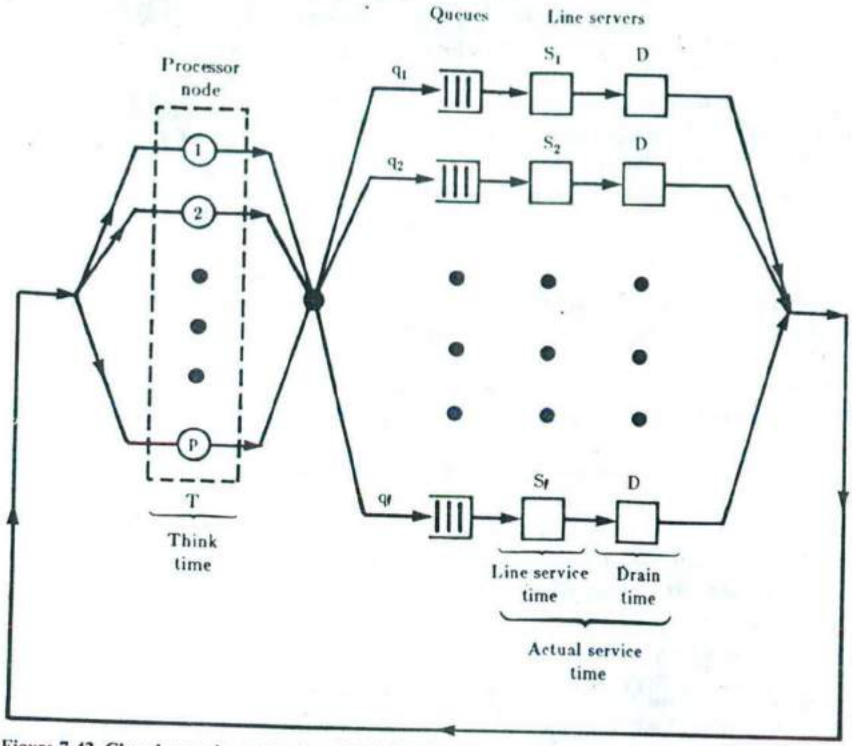


Figure 7.42 Closed queueing network model for a multiprocessor system.

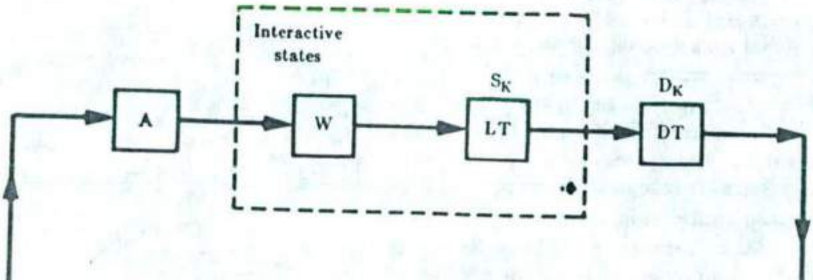


Figure 7.43 State graph model for memory requests in a tightly coupled multiprocessor.

state has a different average duration. Each processor goes through an independent state (state A) followed by interactive states (states W and LT) and another independent state (DT).

In an independent state, a processor executes on its own node without conflict. Interactive states are characterized by a potential for conflicts with other processors. Hence, during any LT state, the memory line is busy and no other processor can access the line. Let C be the average memory-request cycle time. The performance index is the average processor utilization U, defined as the average fraction of time spent by each processor in processing instructions. This performance index reflects the degree of matching between the processors and the memory organization.

We number the processors from 1 to p and the memory lines from 1 to l. Let

$$I_k(t) = [i_{k,1}(t), i_{k,2}(t), \ldots, i_{k,p}(t)] \tag{7.16}$$

for $k = 1, \ldots, l$, with $i_{k,j}(t) = 1$ if processor j is not waiting for or using line k, and $i_{k,j}(t) = 0$ if processor j is waiting for or using line k at time t.

$I_k(t)$ is called the *indicator vector* for line k at time t. Each component $i_{k,j}(t)$ indicates whether or not processor j is waiting for or holding line k. Note that a processor waits for or holds a line whenever it is in state W or LT (interactive states), respectively. Let X_k be the probability that a line k is busy and S_k is the average line service time of a request. Then

$$\begin{aligned} X_k &= \text{Prob[at least one processor is waiting for or holding line } k] \\ &= 1 - \text{Prob[no processor is waiting for or holding line } k] \\ &= 1 = \text{Prob}[i_{k,1} \cdot i_{k,2} \cdots i_{k,p} = 1] = 1 - E[i_{k,1} \cdot i_{k,2} \cdots i_{k,p}] \end{aligned}$$

This last equality results from the fact that the expectation of a random variable taking only the values 0 and 1 is equal to the probability of the variable being 1. The rate of completed requests by line k is X_k/S_k.

In equilibrium, this rate can be equated to the rate of submitted requests to a line. To compute this second member of the equation, we note that a processor submits a request whenever it departs from state A. This occurs for each processor whenever a cycle in the network of Figure 7.43 is completed. Recall that C is the average time taken by such a cycle. The rate of submitted requests for the memory by any one processor is $1/C$. Since there are p requesting processors and each request is submitted with probability q_k to line k, the average rate of submitted requests to line k is pq_k/C.

Let Y be the average fraction of time a given processor is in an independent state. Hence, Y is also the probability of being in such a state. The symmetry of the system implies the same value of Y for all the processors. Since T is the average time in state A, $Y = (T + D)/C$. Substituting for $1/C$ in the equation for the average rate of submitted requests to a given line and equating this rate to the rate of completed request, we obtain $X_k = S_k p Y q_k/(T + D)$. Substituting for X_k, we have

$$E[i_{k,1} \cdot i_{k,2} \cdots i_{k,p}] + \rho_k Y = 1 \tag{7.17}$$

where $\rho_k = S_k p q_k/(T + D)$.

This equation is exact. However, the first term of the left hand side of the equation is very complex to estimate in general. The approximation consists in neglecting the interactions between processors. As a result of the approximation, the components of $I_k(t)$ are not correlated. This approximation performs best for a short and deterministic line-service time. Indeed, large instances of the line-service time are most likely to result in instantaneous longer queues and more interactions between the processors. Under the noncorrelation conditions

$$E[i_{k,1} \cdot i_{k,2} \cdots i_{k,p}] = E[i_{k,1}] \cdot E[i_{k,2}] \cdots E[i_{k,p}] \tag{7.18}$$

If we denote by Z_k the fraction of time spent by each processor waiting for or holding line k, Eq. 7.17 becomes $(1 - Z_k)^p + \rho_k Y = 1$ because of the symmetry of the system.

On the other hand, since a processor is either in an independent state or in an interactive state (waiting for or holding one of the lines), then, by the law of total probability, in a system with l lines we have

$$Y + \sum_{k=1}^{l} Z_k = 1 \tag{7.19}$$

We use Eq. 7.18 with the condition that $1 - \rho_k Y > 0$, $Z_k = 1 - (1 - \rho_k Y)^{1/p}$. Consequently, by the substitution for Z_k in Eq. 7.19 and rearranging, we obtain

$$Y = 1 - l + \sum_{k=1}^{l} (1 - \rho_k Y)^{1/p} \tag{7.20}$$

S_k is the mean line-service time and $T + D$ is the mean time between an exit from an interactive state and a visit to the next interactive state. Note that S_k can be found as the mean time that a processor spends holding a memory line k. Similarly, T is found as the mean time spent outside of an interactive state. Y can be solved by Newton's iterative method given that a unique solution exists for Y between 0 and $\min(1, 1/\rho_k)$. Let us illustrate the application of this model to the system mentioned earlier. For simplicity, assume that for these examples $q_k = 1/l$ and $S_k = S$ for all values of k. Then Eq. 7.20 becomes:

$$Y = \frac{1}{\rho}\left[1 - \left(\frac{Y + l - 1}{l}\right)^p\right] \tag{7.21}$$

where $\rho = pS/l(T + D)$.

Assume that in each of these processors, a machine cycle consists of an integer number, d, of cache cycles. Let θ be the probability that a memory request is issued by a processor to the cache controller in a machine cycle. Thus, the fraction of references made by the processor to the cache controller in each cache cycle is, $x = \theta/d$.

For the set-associative cache, if a block-write is not required (with a probability $1 - w_b$) on a cache miss, then the line which accepts the memory request is busy for $b - m + c$ time units. However, if a block-write is required (with probability w_b) in addition to the block-read, then two consecutive block transfers (each of size b)

are made uninterruptedly on the same memory line. In this case, the line that accepts the memory request is busy for $2b - m + c$ time units. Hence the mean line-service time is:

$$S = b(1 + w_b) - m + c \tag{7.22}$$

Since $1 - h$ is the cache-miss ratio, the probability that a given cache cycle requires an access to memory is $x(1 - h)$. We account for the crossbar switch setup and traversal times by t_x. Hence the average time spent in state A is:

$$T = \frac{1}{x(1 - h)} + t_x \tag{7.23}$$

Since the drain time, $D = m - 1$, we can determine Y from Eq. 7.21 for the multiprocessor system with set-associative caches. The processor utilization, U, which is the fraction of time the processor is busy processing instructions is given by

$$U = \frac{1/[x(1 - h)]}{C} = \frac{1}{x(1 - h)C}$$

Since $C = (T + D)/Y$, the utilization can be rewritten as

$$U = \frac{Y}{x(1 - h)(T + D)} \tag{7.24}$$

We illustrate the set-associative cache example with a multiprocessor system with $p = 16$ processors, $x = 0.4$, and $t_x = 0$ (infinitely fast crossbar). The cache hit ratio, $h = 0.95$, and the block size b is allowed to vary. The memory organization has a fixed number of modules per line ($m = 4$), but the number of lines vary. Also the memory module cycle time, c, varies. Figure 7.44 shows the application of Eqs. 7.21 through 7.24 for the given set of parameters.

This result assumes that the cache size is adjusted to give the same hit ratio when the block size is varied. An increase in the block size deteriorates the utilization. An operating region should be chosen where the utilization is acceptable. Note that for certain values of b and c, small values of l will give high utilization. This possible reduction in l gives the designer a choice. If for a small number of lines, $l < 16$, the utilization is acceptable, the designer can consider trade-offs with low-cost multiport memory.

7.3.3 Multicache Problems and Solutions

The presence of private caches in a multiprocessor necessarily introduces problems of *cache coherence*, which may result in data inconsistency. That is, several copies of the same data may exist in different caches at any given time. This is a potential problem especially in asynchronous parallel algorithms which do not possess explicit synchronization stages of the computation. For example, process A, which runs on processor i, produces data x, which is to be consumed by process B, which runs on processor $j \neq i$ asynchronously. Process A writes a new x into its cache while process B uses the old value of x in its cache because it is not aware of

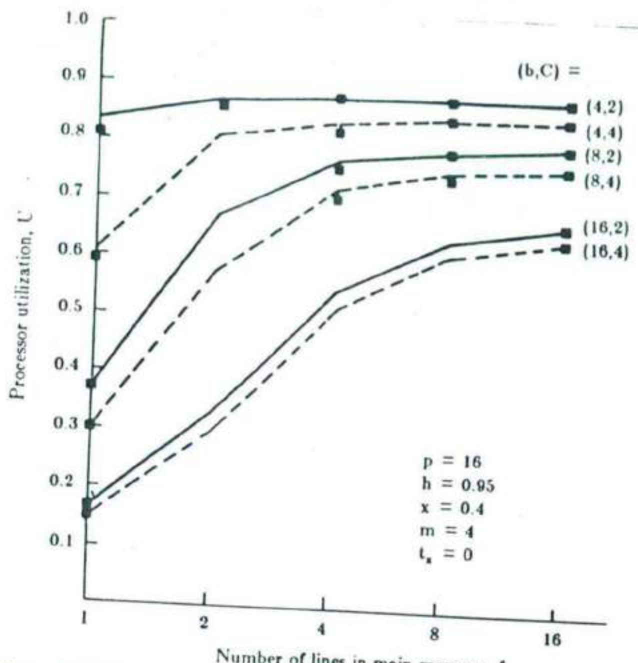


Figure 7.44 Effect of block size and main memory speed on processor utilization for the set-associative cache. (Courtesy of *IEEE Trans. Computers*, Briggs and Dubois, Jan. 1983.)

the new x. Process B may continue to use the old value of x in its cache unless it is informed of the presence of the new x in process A's cache so that a copy of it may be made in its cache. The possibility of having several processors using different copies of the same data must be avoided if the system is to perform correctly. Hence, data consistency must be enforced in the caches.

Another form of the data consistency problem occurs in a multiprogrammed multiprocessor system. In this case, a processor usually switches to other processes at the time of the arrival of external interrupt signals or page fault operations. If the suspended process migrates to another processor, the most recently updated data of this process might still be in the original processor's cache. Hence a process running on a new processor could use stale data in main memory. The new processor cannot recognize the data as stale, and thus would not be working with the process's proper context. Such an operation is incorrect and can result in subtle errors that are difficult to trace. In many multiprocessor systems such as the S-1, privileged instructions are provided to *sweep* the cache. The cache sweep operation is used to deliberately update main memory to reflect any changes in cache contents.

A system of caches is *coherent* if and only if a READ performed by any processor i of a main memory location x (which may be cached by other processors) always delivers the most recent value with the same address x. "Most recent" in this context has a special meaning in terms of a partial ordering of the READs and WRITEs of memory throughout the multiprocessor. However, for an intuitive understanding of the problem, it is sufficient to think of recency in terms of absolute time. In these terms, whenever a WRITE is done by one processor i to a memory location x, completion of the WRITE must guarantee that all subsequent READs of location x by any processor will deliver the new contents of x until another WRITE to x is completed.

The cache coherence problem exists only when the caches are associated with the processors. Designs have been proposed in which the caches are associated with the shared memory as shown in Figure 7.45. This avoids the cache coherence problem. This architecture is good for systems with a small number of processors. However, the potential gain in speed is then limited by the transmission delays through the interconnection network and by the conflicts at the caches. This technique has been shown to be adequate for multiprocessors where each processor is pipelined and executes multiple independent instruction streams.

Clearly, the cache coherence problem cannot be solved by a mere write-through policy. If a write-through policy is used, the main memory location is updated, but the possible copies of the variable in other caches are not automatically updated by the write-through mechanism. When a processor modifies a data in its cache, all the potential copies in other caches must be invalidated. "Write-through" is neither necessary nor sufficient for coherence.

Static coherence check Two different methods have been proposed to solve the cache coherence problem. The first method, called *static coherence check*, avoids

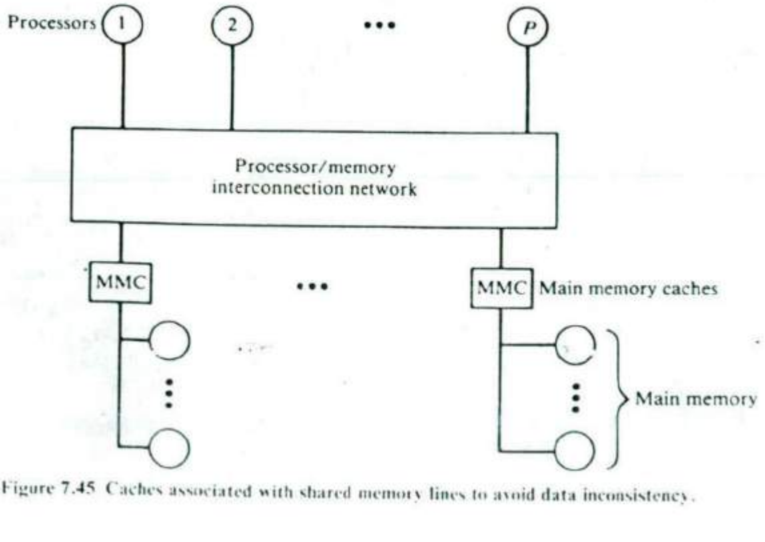


Figure 7.45 Caches associated with shared memory lines to avoid data inconsistency.

multiple copies by implementing different paths for shared writeable (*non-cacheable*) and private (*cacheable*) data. By data, we mean both code and operand. The shared data structures which are modifiable reside in main memory. They are never placed in the cache; that is, they are noncacheable. A reference to this shared data is made directly to main memory. Conversely, a read only segment of data which is shared by several processors need not be noncacheable. The cacheability of read only data reduces conflicts in main memory.

If t_m is the time to reference a datum in main memory, t_c the cache cycle time, and s the probability of referencing a shared modifiable datum, a lower bound on a datum reference is

$$(1 - s)t_c + st_m = t_c\left[(1 - s) + s\frac{t_m}{t_c}\right] \tag{7.25}$$

If the cycle ratio, t_m/t_c, is large (a typical value is 5), the performance of this scheme may be quite poor for algorithms with intense sharing, regardless of the cache size. Moreover, the requests for shared data increase contentions in the interconnection network and at the memory. The performance of this scheme can be improved by associating a high-speed memory or cache module with each memory line. This cache module is used to buffer the noncacheable data, thereby reducing the effective t_m and hence, the cycle ratio, t_m/t_c.

In a similar scheme which avoids these problems, the shared data is accessed through a shared cache while instruction fetches and private data references are made in private caches. Figure 7.46 illustrates this shared cache concept. Notice that the shared cache may consist of interleaved cache modules which may be connected to the processors and shared memory through an interconnection network. However, the complexity of this network is expected to be less than a full crossbar. All data references proceed at the cache speed except when conflicts occur at the shared cache or a miss occurs in either a private cache or the shared cache.

If the hit ratio is high enough in all caches, this scheme alleviates the contention problem at the main memory. The success of the shared cache concept relies on the relatively low rate of shared data references. Of course, the shared data may exhibit less locality than private data. However, the hit ratio in the shared cache improves as the degree of sharing of the shared variables increases. Indeed, a processor may find a shared variable in the shared cache even if it never referenced it before. Moreover, increasing the size of the shared cache is an effective method to improve the hit ratio.

The shared cache concept requires that data be tagged as private or shared. The tagging is basically static. *Static tags* are made during compile time and remain the same throughout the lifetime of the process. *Dynamic tags* are made during the execution of cooperating processes. A lookahead mechanism monitors the history of sharing of the data space in one phase and predicts the probability of sharing of the subspaces in the next phase. With this scheme, a data subspace could be in the shared cache for effective sharing in one phase and in the private caches in another phase, for efficient access, or vice-versa. The overhead may be unacceptable, as

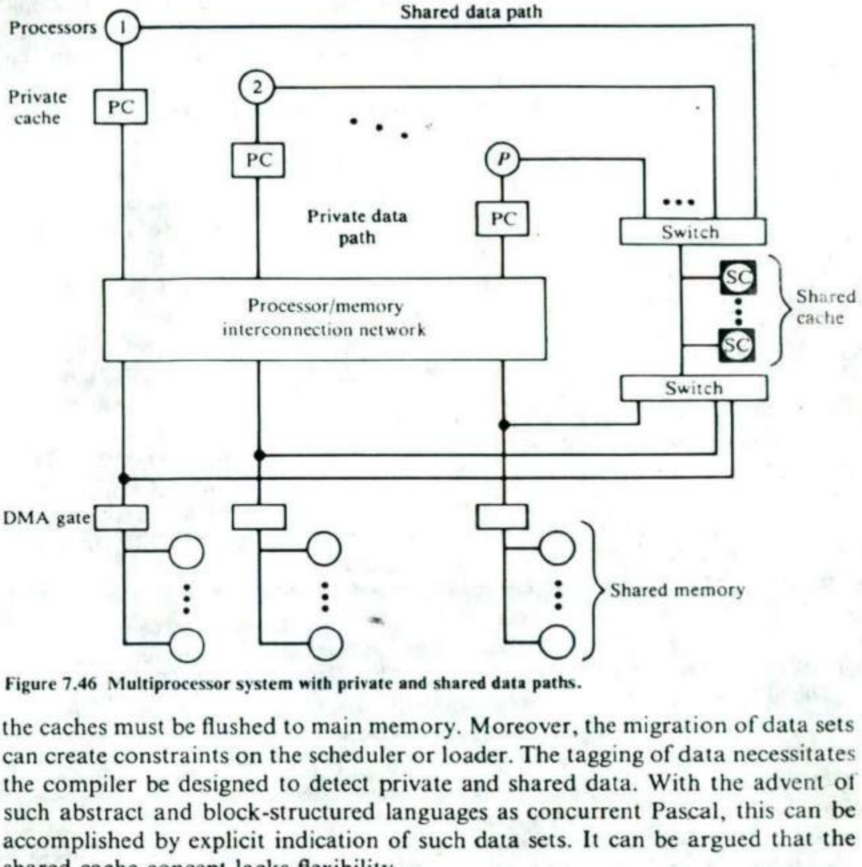


Figure 7.46 Multiprocessor system with private and shared data paths.

the caches must be flushed to main memory. Moreover, the migration of data sets can create constraints on the scheduler or loader. The tagging of data necessitates the compiler be designed to detect private and shared data. With the advent of such abstract and block-structured languages as concurrent Pascal, this can be accomplished by explicit indication of such data sets. It can be argued that the shared cache concept lacks flexibility.

Dynamic coherence check The second method for solving cache coherence is more flexible than the static coherence check, but also more complex and possibly more costly. In this scheme, called *dynamic coherence check*, multiple copies are allowed. However, whenever a processor modifies a location x in a cache block, it must check the other caches to invalidate possible copies. This operation is referred to as a *cross-interrogate* (XI). In the most rudimentary implementation of this method, the caches are tied on a high speed bus. When a local processor writes into a shared block in its cache, the processor sends a signal to all the remote caches to indicate that the "data at memory address x has been modified." At the same time, it writes through memory. Note that, to ensure correctness of execution, a processor which requests an XI must wait for an acknowledge signal from all other processors before it can complete the write operation. The XI invalidates the remote cache location corresponding to x if it exists in that cache.

When the remote processor references this invalid cache location, it results in a cache miss, which is serviced to retrieve the block containing the updated information.

For each write operation, $(n - 1)$ XIs result, where n is the number of processors. When n increases, the traffic on the high-speed bus becomes a bottleneck. Moreover, there is a potential for races if the XI requests are queued to accommodate the peak traffic on the bus. Some commercial multiprocessors with caches use this technique for a small number of processors. For example, the Honeywell 60/66 and Univac 1100/80 multiprocessors have cache-invalidate interfaces between every pair of caches. Note that the two sources of inefficiency for this technique are the necessity of a write-through policy, which increases the network traffic, and the redundant cache XIs which are performed. In the latter case, a cache is purged blindly whether or not it contains the data item x.

A more refined technique filters the XI requests before they are initiated. In a proposed design, the memory control element (MSC) maintains a central copy of the directories of all the caches. We will elaborate on a similar scheme called the *presence flag technique*, which assumes a write-back update policy. There are two central tables associated with the blocks of main memory (MM) (Figure 7.47). The first table is a two-dimensional table called the Present table. In this table, each entry $P[i, c]$ contains a *present* flag for the ith block in MM and the cth cache. If $P[i, c] = 1$, then the cth cache has a copy of the ith block of MM, otherwise it is zero. The second table is the *Modified* table and is one-dimensional. In this table, each entry $M[i]$ contains a *modified* flag for the ith block of MM. If $M[i] = 1$, it means that there exists a cache with a copy of the ith block more recent than the corresponding copy in MM. The Present and Modified tables can be implemented in a fast random-access memory.

The philosophy behind the cache coherence check is that an arbitrary number of caches can have a copy of a block, provided that all the copies are identical. They are identical if the processor associated with each of the caches has not attempted to modify its copy since the copy was loaded in its cache. We refer to such a copy as *read only* (RO) copy. In order to modify a block copy in its cache, a processor must own the block copy with *exclusive read only* (EX) or *exclusive read-write* (RW) access rights. A copy is held EX in a cache if the cache is the only one with the block copy and the copy has not been modified. Similarly, a copy is held RW in a cache if the cache is the only one with the block copy and the copy has been modified. Therefore, for consistency, only one processor can at any time own an EX or RW copy of a block.

To enforce the cache consistency rule, local flags are provided within each cache in addition to the global tables. A local flag $L[k, c]$ is provided for each block k in cache c. This flag indicates the state of each block in the cache. A block in a cache can be in one of three states: RO, EX, or RW. When a processor c fetches a block i on a read miss, the processor obtains an EX copy of the block, provided no other cache has a copy of block i and the fetch was for data. In other fetch misses, the block is assigned RO, as shown in Figure 7.48. The status information is recorded in the cache directory and global tables. The status is indicated

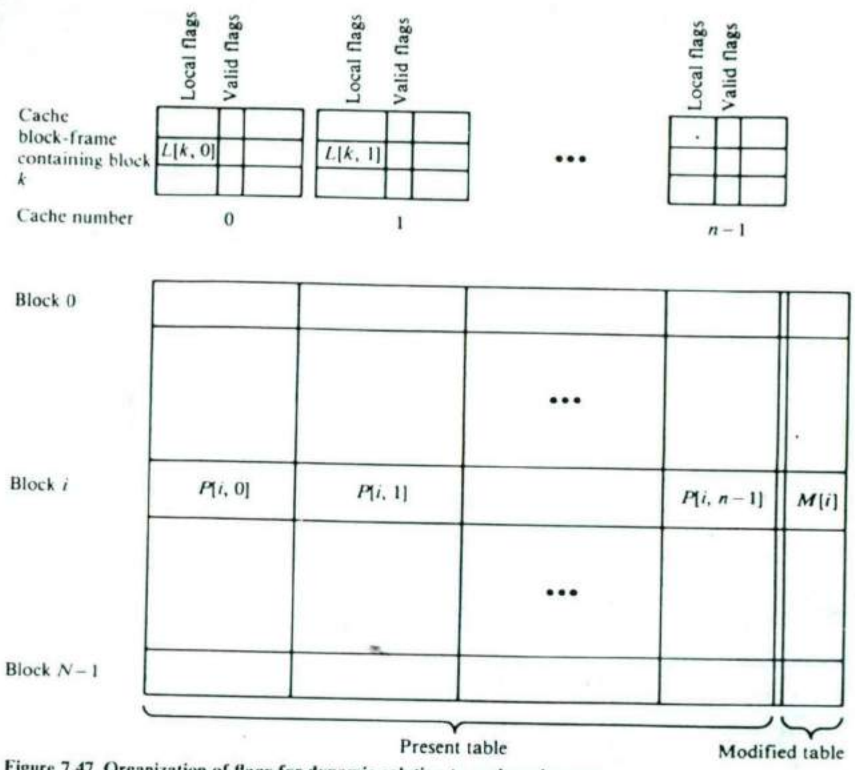
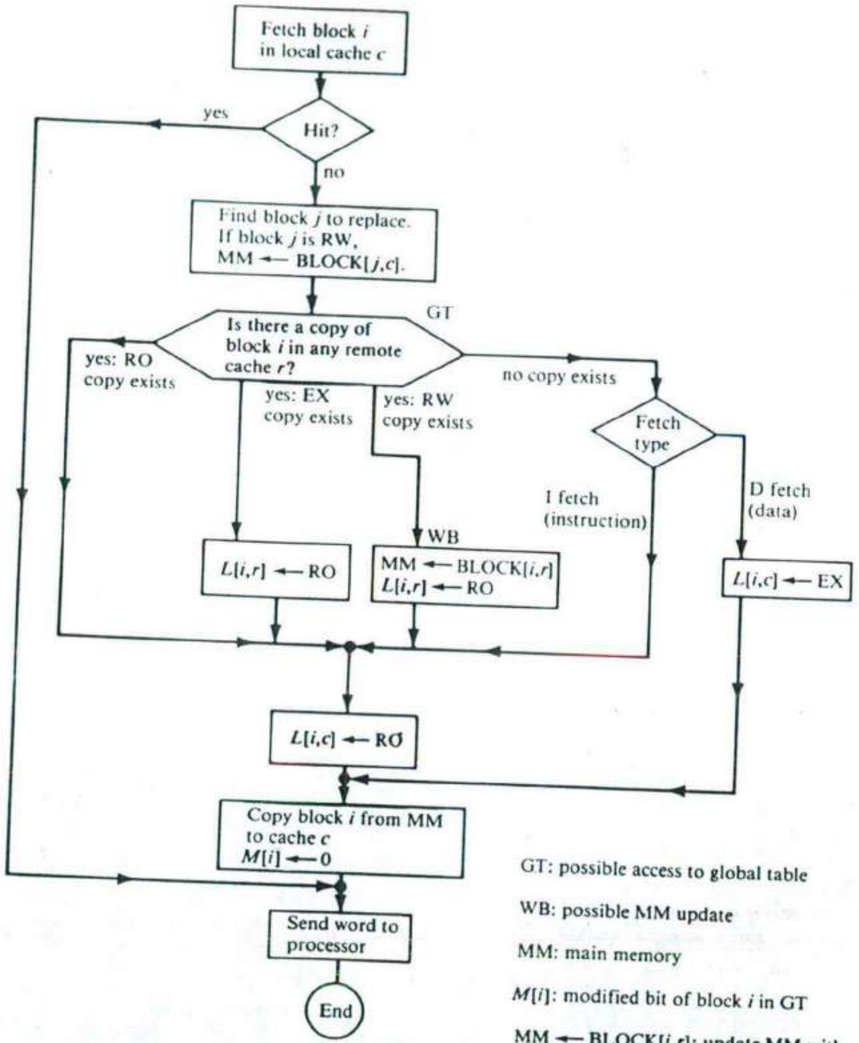


Figure 7.47 Organization of flags for dynamic solution to cache coherence.

in the global table by setting the appropriate present flag and clearing the corresponding modified bit.

As long as the copy of block *i* remains present in the cache, processor *c* can fetch it without any consistency check. If processor *c* attempts to store into its copy of block *i*, it must ensure that all other copies (if any) of block *i* are invalidated. To do this, the global table is consulted. It should indicate the processor caches that own a copy of block *i*. The modified bit for block *i* is updated in the global table to record the fact that processor *c* owns block *i* with RW access rights. Finally, the local $L[i, c]$ flag is set to RW to indicate that the block is modified. The flowchart for a store is given in Figure 7.49.

In this implementation, a block copy in a cache is invalidated whenever the cache receives a signal from some other processor attempting to store into it. Moreover, a cache which owns an RW copy may receive a signal from a remote cache requesting to own an RO copy. In this case, the RW copy's state is changed to RO.


Fetch block *i* in local cache *c*
Hit?
yes
no
Find block *j* to replace. If block *j* is RW, MM ← BLOCK[*j*,*c*].
Is there a copy of block *i* in any remote cache *r*?
GT
yes: RO copy exists
yes: EX copy exists
yes: RW copy exists
no copy exists
Fetch type
I fetch (instruction)
D fetch (data)
WB
L[*i*,*r*] ← RO
MM ← BLOCK[*i*,*r*] *L*[*i*,*r*] ← RO
L[*i*,*c*] ← EX
L[*i*,*c*] ← RO
Copy block *i* from MM to cache *c* *M*[*i*] ← 0
Send word to processor
End

GT: possible access to global table

WB: possible MM update

MM: main memory

M[*i*]: modified bit of block *i* in GT

MM ← BLOCK[*i*,*r*]: update MM with modified copy of *i* in cache *r*

L[*i*,*r*]: state flag of block *i* in cache *r*

Figure 7.48 Coherence check for fetch operation.

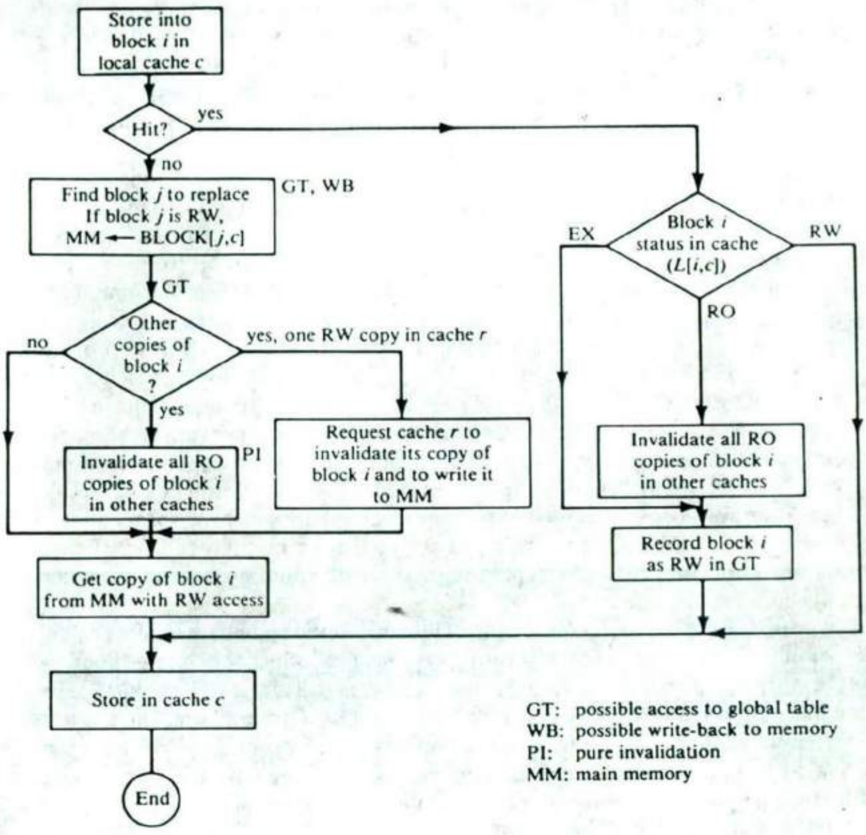

Store into block i in local cache c
Hit?
yes
no
Find block j to replace If block j is RW, MM ← BLOCK[j,c]
GT, WB
GT
Other copies of block i ?
no
yes
yes, one RW copy in cache r
Invalidate all RO copies of block i in other caches
PI
Request cache r to invalidate its copy of block i and to write it to MM
Get copy of block i from MM with RW access
Block i status in cache (L[i,c])
EX
RW
RO
Invalidate all RO copies of block i in other caches
Record block i as RW in GT
Store in cache c
End
GT: possible access to global table
WB: possible write-back to memory
PI: pure invalidation
MM: main memory

Figure 7.49 Coherence check for store operation.

There are four main sources of performance degradation in the dynamic coherence check algorithms shown in Figures 7.48 and 7.49. These sources are:

- Degradation of the average hit ratio due to block invalidation
- Traffic between caches to enforce consistency
- Concurrent access to the global tables resulting in conflicts
- Writeback due to invalidation of RW data

7.4 MULTIPROCESSOR OPERATING SYSTEMS

In this section, we discuss the operating system requirements for multiprocessors. First, a classification of multiprocessor operating systems is presented. We then discuss other system software supports needed for multiprocessing.

7.4.1 Classification of Multiprocessor Operating Systems

There is conceptually little difference between the 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 processors 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 schemes, memory and dataset protection, prevention of system deadlocks and abnormal process termination or exception handling. In addition to these capabilities, multiprocessor systems also need techniques for efficient utilization of 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. An operating system which performs 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 a large number of processors on the design of 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 the operating system. We introduce below only the basic configurations that have appeared in existing multiprocessor systems.

There are basically three organizations that have been utilized in the design of operating systems for multiprocessors, namely, *master-slave configuration, separate supervisor* for each processor, and *floating supervisor control.* 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 a uniprocessor operating system that includes full multiprogramming capabilities. Although the master-slave type of system is simple, it is normally quite inefficient in its control and utilization of the total system resources. The other two operating modes are superior to the master-slave in performance.

In a *master-slave* mode, one processor, called the master, maintains the status of all processors in the system and apportions the work to all the slave processors. An example of the master-slave mode is in the Cyber-170, where the operating system is executed by one peripheral processor P_0. All the other processors (central or peripheral) are treated as slaves to P_0. Another example is found in the DEC System 10, in which there are two identical processors. One of the processors is designated as master and the other as slave. The operating system runs only on the master, with the slave treated as a schedulable resource.

Since the supervisor routine is always executed in the same processor, a slave request via a trap or supervisor call instruction for an executive service must be sent to the master, which acknowledges the request and performs the appropriate service. The supervisor and its associated procedures need not be reentrant since there is only one processor that uses them. There are other characteristics of the master-slave operating system. Table conflicts and lock-out problems for system control tables are simplified by forcing a single processor to run the executive. However, this operating system mode causes the entire system to be very susceptible to catastrophic failures which require operation intervention to restart the master processor when an irrecoverable error occurs. In addition to the inflexibility of the overall system, the utilization of the slave processors may become appreciably low if the master cannot dispatch processes fast enough to keep the slaves busy. The master-slave mode is most effective for special applications where the work load is well defined or for asymmetrical systems in which the slaves have less capability than the master processor. It is the mode sometimes used if there are very few processors involved.

When there is a *separate supervisor system* (kernel) running in each processor, the operating system characteristics are very different from the master-slave systems. This is similar to the approach taken by computer networks, where each processor contains a copy of a basic kernel. Resource sharing occurs at a higher level, for example, via a shared file structure. Each processor services its own needs. However, since there is some interaction between the processors, it is necessary for some of the supervisory code to be reentrant or replicated to provide separate copies for each processor. Although each supervisor has its own set of private tables, some tables are common and shared by the whole system. This creates table access problems. The method used in accessing the shared resources depends on the degree of coupling among the processors. The separate supervisor operating system is not as sensitive to a catastrophic failure as a master-slave system. Also, each processor has its own set of input-output devices and files, and any reconfiguration of I/O usually requires manual intervention and possibly manual switching.

Unfortunately, the replication of the kernel in the processors would demand a lot of memory which may be underutilized, especially when compared with the utilization of the shared data structures. A static form of caching could be used to buffer frequently used portions of the operating system code, while the infrequently used code could be centralized in a shared memory. Unfortunately, the determination of which portions of operating system are frequently executed is relatively difficult to make and is likely to be dependent of the application workload.

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 the most flexible. In this mode, the supervisor routine floats from one processor to another, although several of the processors may be executing supervisory service routines simultaneously. This type of system can attain better load balancing over all types of resources. Conflicts in service requests are resolved by priorities that are either set statistically or under dynamic control. Since there is a considerable degree of sharing, most of the supervisory code must be reentrant. In this system, table-access conflicts and table lock-out delays cannot be avoided. It is important to control these accesses in such a way that system integrity is protected. This mode of operation has the advantages of providing graceful degradation and better availability of a reduced capacity system. Furthermore, it provides true redundancy and makes the most efficient use of available resources. Examples of operating systems that execute in this mode are the MVS and VM in the IBM 3081 and the Hydra on the C.mmp.

Most operating systems, however, are not pure examples of any of the three classes discussed above. The only generalization that is possible is that the first system produced is usually of the master-slave type and the ultimate being sought is the floating supervisor control. In Table 7.3, we summarize the major characteristics, advantages, and shortcomings of the above three types of operating systems for multiprocessor computers.

7.4.2 Software Requirements for Multiprocessors

One of the issues often raised in a discussion on multiprocessor software is the question of how it differs from uniprocessor software. In particular, how does software written to execute on multiple processors differ from that written to execute on the more familiar multiprogrammed uniprocessor environment? There are basically two sources of differences. These are the architectural attributes that are unique to the multiprocessor, and a new programming style peculiar to parallel applications. Such differences would warrant that the hardware and software of the system should provide facilities that are different from those found in conventional multiprogrammed uniprocessor environments. A multiprogrammed uniprocessor can simulate the multiple processor environment by creating multiple "virtual processors" for the users. For example, a Unix user routinely requests the concurrent execution of multiple programs with the output of one program "piped" as the input to the other. In this case, each program may be thought of as executing on a virtual processor. At this level of program execution there are few differences between a multiprogrammed uniprocessor system and a multiprocessor system. However, the presence of multiple processors and other replicated components usually increases the amount of management software that must be provided.

An architectural attribute that may affect programming in a multiprocessor system is nonhomogeneity. If the central processors are nonhomogeneous, that is, functionally different, they must be treated differently by software. For example, if one processor possesses emulation capability not possessed by another, some programs can only run to completion on the processor with the emulation capa-

Table 7.3 Operating system configurations for a multiprocessor computer

Master-slave operating system:

1. The executive routine is always executed in the same processor. If the slave needs service that must be provided by the supervisor, then it must request that service and wait until the current program on the master processor is interrupted and the supervisor is dispatched. The supervisor and the routines that it uses do not have to be reentrant since there is only the one processor using them.
2. Having a single processor executing the supervisor simplifies the table conflict and lock-out problem for control tables. The overall system is comparatively inflexible. This type of system requires comparatively simple software and hardware.
3. The entire system is subject to catastrophic failures that require operator intervention to restart when the processor designated as the master has a failure or irrecoverable error.
4. Idle time on the slave system can build up and become quite appreciable if the master cannot execute the dispatching routines fast enough to keep the slave(s) busy.
5. This type of operating system is most effective for special applications where the work load is well defined or for asymmetrical systems in which the slaves have less capability than the master processor.

Separate supervisor in each processor:

1. Each processor services it own needs. In effect, each processor (supervisor) has its own set of I/O equipment, files, etc.
2. It is necessary for some of the supervisory code to be reentrant or replicated to provide separate copies for each processor.
3. Each processor (actually each supervisor) has its own set of private tables, although some tables must be common to the entire system, and that creates table-access control problems.
4. The separate supervisor operating system is as sensitive as is the master-slave system; however, the restart of an individual processor that has failed will probably be quite difficult.
5. Because of the point immediately above, the reconfiguration of I/O usually requires manual intervention and possibly manual switching.

Floating-supervisor operating system:

1. The "master" floats from one processor to another, although several of the processors may be executing supervisor service routines at the same time.
2. This type of system can attain better load balancing over all types of resources.
3. Conflicts in service requests are resolved by priorities that can be set statically or under dynamic control.
4. Most of the supervisory code must be reentrant since several processors can execute the same service routine at the same time.
5. Table-access conflicts and table lock-out delays can occur, but there is no way to avoid this with multiple supervisors; the important point is that they must be controlled in such a way that system integrity is protected.

bility. Hence, software resource managers must provide appropriate dispatching mechanisms for such programs. Another example of software complexity occurs in a system with asymmetric main memory. In this case, not all processors can access all memory. This complicates the operating system software for resource management.

There is a second potential source of difference between multiprocessor and uniprocessor software. This is in the programming style peculiar to parallel applications. The basic unit of a program in execution is that of a process, an independent schedulable entity (a sequential program) that runs a processor and uses hardware and software resources. It may also execute concurrently with other processes, delayed (at least logically) only when it needs to wait to interact with one or more other processes. Hence, a parallel program can be said to consist of two or more interacting processes.

The potential of multiprocessing is achieved by enhancing its capability for parallel processing. Parallel processing can be indicated in a program explicitly or implicitly. For explicit parallelism, users must be provided with programming abstractions that permit them to indicate explicit parallelism when desired in a program. Implicit parallelism is detected by the compiler. In this case, the compiler scans the source program and recognizes the program flow. From this flow graph and other conditions, it detects nontrivial units of program statements which may be identified as a process. Some of these units may be independent and can be run concurrently with other processes.

In a multiprocessor system, synchronization takes on increased importance as it could create too high a penalty. This could significantly degrade system performance if the synchronization mechanisms are not efficient and the algorithms that use them are not properly designed. In some processors, the synchronization primitives are not implemented directly in hardware or microcode. Therefore, software alternatives must be provided. For example, the PDP-11 processors used for the C.mmp have been implemented with the semaphore-synchronization primitive in software, thereby taking a significant number of instructions. In an environment where processes need to synchronize often, this may be a major bottleneck.

Program-control structures are provided to aid the programmer in developing efficient parallel algorithms. Three basic nonsequential program-control structures have been identified. These control structures are characterized by the fact that the programmer need only focus on a small program and not on the overall control of the computation. The first example is the *message-based* organization which was used in the Cm* operating system. In this organization, computation is performed by multiple homogeneous processes that execute independently and interact via messages. The grain size of a typical process depends on the system.

The second example of a control structure is the *chore* structure. In this structure, all codes are broken into small units. The process that executes the unit of code (and the code itself) is called a *chore*. An important characteristic of a chore is that once it begins execution, it runs to completion. Hence, to avoid long waits, chores are basically small. They have relatively very little input and they reference only a few different objects. Moreover, they do not block and are noninterruptible. As part of its output, one chore might request the execution of a small set of additional chores. Examples of systems that use this structure are the Pluribus and the BCC-500.

Consider the memory-management portion of the operating system, which controls swapping between the main memory and a fixed-head disk. Sample chores may include (*a*) the disk command to request the transfer of a page of data between the disk and the memory, and (*b*) acknowledging completion of a disk-sector transmission and arranging for any subsequent action.

The third nonsequential control structure is that of *production systems*, now often used in artificial intelligence systems. Productions are expressions of the form ⟨antecedent, consequent⟩. Whenever the boolean antecedent evaluates to true, the consequent may be performed. In contrast to chores, production consequents may or may not include code which might block. In a production system, four scheduling strategies are often required (*a*) to control the selection of antecedents to be evaluated next, (*b*) to order (if necessary) the execution of selected antecedents, (*c*) to select the subset of runnable consequents to be executed, and (*d*) to order (if necessary) the execution of the selected consequents. Note that by the natures of all three control structures, they are all compatible with parallel execution.

The high degree of concurrency in a multiprocessor can increase the complexity of fault handling, especially in the recovery step. In a uniprocessor, it is always possible to eliminate parallelism by disabling interrupts and, if necessary, halting I/O activity. Software is needed to establish effective error recovery capability. This software, even with the aid of hardware mechanisms, may be quite complex. Understanding the behavior of running processes in a multiprocessor system is more complex than in uniprocessor environments. Although parallel programs may not be too complex to implement, there is a natural problem of nondeterminism in multiprocessors. Some efforts have been made to prove the correctness of parallel programs by researchers but extending these proofs to complex programs is still a formidable task.

7.4.3 Operating System Requirements

The basic goals for an operating system are to provide programmer interface (environment) to the machine, manage resources, provide mechanisms (system defined) to implement policies (user definable), and facilitate matching applications to the machine. It must also help achieve reliability. But this and other desirable attributes incur a cost that may be unacceptable. Guidelines should be established for trading performance for desirable attributes. The degree of transparency of the detailed machine that should be made available to the programmer should also be determined.

There are different levels of interaction in the specification of an operating system for multiprocessing systems. Asynchronous supervisor processes share the specification of the address-space management, process management, and synchronization levels. Efficient operating systems are designed to have a modular structure and hierarchical organization. This makes the detection and localization of errors easier. The classic functions of an operating system include the creation of objects such as processes and their domains, which include the memory segments. The management and sharing of segments, as was discussed in Chapter 2, are also important operating system functions. Other functions are the management of process communications through mailboxes or message buffers. Messages are used to define the interface between processes and help to reduce the number of ways an error can be propagated through the system.

In a multiprocessor system, processes can execute concurrently until they need to interact. Planned and controlled interaction is referred to as process communication or process synchronization. Process communication must take place through shared or global variables. Cooperating processes must communicate to synchronize or limit their concurrency. The relationship between two cooperating processes regarding a resource falls into one of two fundamental categories. They are either competitors or producers-consumers. Since process communication takes place through shared memory, competitors access this memory to seize and release permanent or *reusable* resources. Producers-consumers access this memory to pass temporary or *consumable* resources such as messages and signals.

In systems with multiple concurrent processes, the presence of resources such as unit record peripherals and tape drives which must not be used simultaneously by several processes (if program operation is to be correct) introduces the requirement for exclusive access to these devices. This requirement may also be imposed on shared objects such as a data segment during updating. Processes desiring exclusive access to a resource may compete for it. The same competition arises concerning access to what are called *virtual* resources, such as system tables and communication buffers between cooperating processes. Since it must be guaranteed that both processes do not access the buffer simultaneously, exclusive access to the buffer must be ensured. This exclusiveness of access is called *mutual exclusion* between processes. A request for mutual exclusion on the use of a resource implies the desire to reserve or release the resource. Process cooperation and competition may both be implemented if a mechanism is provided for process coordination or synchronization. This mechanism will be discussed in Section 8.1.

The above requirements for processor cooperation and competition have obvious implications on short- and medium-term scheduling of the multiple processors. If a desired resource or object is not available, the process requesting it must be suspended, blocked, or retry until it becomes available. There are often two levels of exclusiveness. One consists of the requirements for referral of access to a data structure (virtual resource) which may often be of short duration. The other is the requirement for, perhaps, a substantial delay until the physical resource, such as the processor or tape unit, becomes available. If the delay is short, it is not worthwhile to shift the attention of the processor from the process which is running on it to another process. If the delay exceeds the time required to switch the processor, the ability to shift attention may be vital for efficient utilization of the processor.

The sharing of the multiple processors may be achieved by placing the several processes together in shared memory and providing a mechanism for rapidly switching the attention of a processor from one process to another. This operation is often called *context switching*. Sharing of the processors introduces three subordinate problems:

1. The protection of the resources of one process from willful or accidental damage by other processes
2. The provision for communication among processes and between user processes and supervisor processes

3. The allocation of resources among processes so that resource demands can always be fulfilled

The goal of protection is to ensure that data and procedures are accessed correctly. When two or more processes wish to access a set of resources within the multiprocessor system, it is necessary to allocate the resources in such a way that the total resources of the system are not exceeded. Furthermore, if it is possible for a process to acquire a portion of the resources that it requires and then subsequently make a request for more, it is necessary to ensure that future demands can always be satisfied. As an example, suppose that the system has one card reader and one printer. Process one requests the card reader and process two requests the printer. If processes one and two subsequently request the printer and card reader, respectively, then the system is in a state where two processes are blocked indefinitely. This situation is called *deadlock* or *deadly embrace*. Methods for detecting and preventing deadlock, protection schemes and communication mechanisms for multiprocessor systems will be discussed in Chapter 8.

7.5 EXPLOITING CONCURRENCY FOR MULTIPROCESSING

A parallel program for a multiprocessor consists of two or more interacting processes. A process is a sequential program that executes concurrently with other processes. In order to understand a parallel program, it is first necessary to identify the processes and the objects that they share. In this section we study two approaches to designing parallel programs. One approach to be introduced below is to have explicit concurrency, by which the programmer specifies the concurrency using certain language constructs. The other approach is to have implicit concurrency. In this case, the compiler determines what can be executed in parallel. This approach is more appropriate for data-flow computations, to be discussed in Chapter 10.

7.5.1 Language Features to Exploit Parallelism

In order to solve problems in an MIMD multiprocessor system, we need an efficient notation for expressing concurrent operations. Processes are *concurrent* if their executions overlap in time. More precisely, two processes are concurrent if the first operation of one process starts before the last operation of the other process terminates. In this case, no prior knowledge is available about the speed at which concurrent processes are executed. In this section, we will discuss the concurrency explicitly indicated by the programmer.

One way to denote concurrency is to use FORK and JOIN statements. FORK spawns a new process and JOIN waits for a previously created process to terminate. Generally, the FORK operation may be specified in three ways: FORK A; FORK A, J; and FORK A J, N. The execution of the FORK A statement initiates another process at address A and continues the current process. The execution of the FORK A, J statement causes the same action as FORK A and also increments a counter at address J. FORK A, J, N causes the same action as FORK A and sets the counter at address J to N. In all usages of the FORK statements, the corresponding JOIN statement is expressed as JOIN J. The execution of this statement decrements the counter at J by one. If the result is 0, the process at address $J + 1$ is initiated, otherwise the processor executing the JOIN statement is released. Hence, all processes execute the JOIN terminals, except the very last one.

Application of these instructions for the control of three concurrent processes is shown in Figure 7.50. These instructions do not allow a path to terminate without encountering a junction point. The problem with FORK and JOIN is

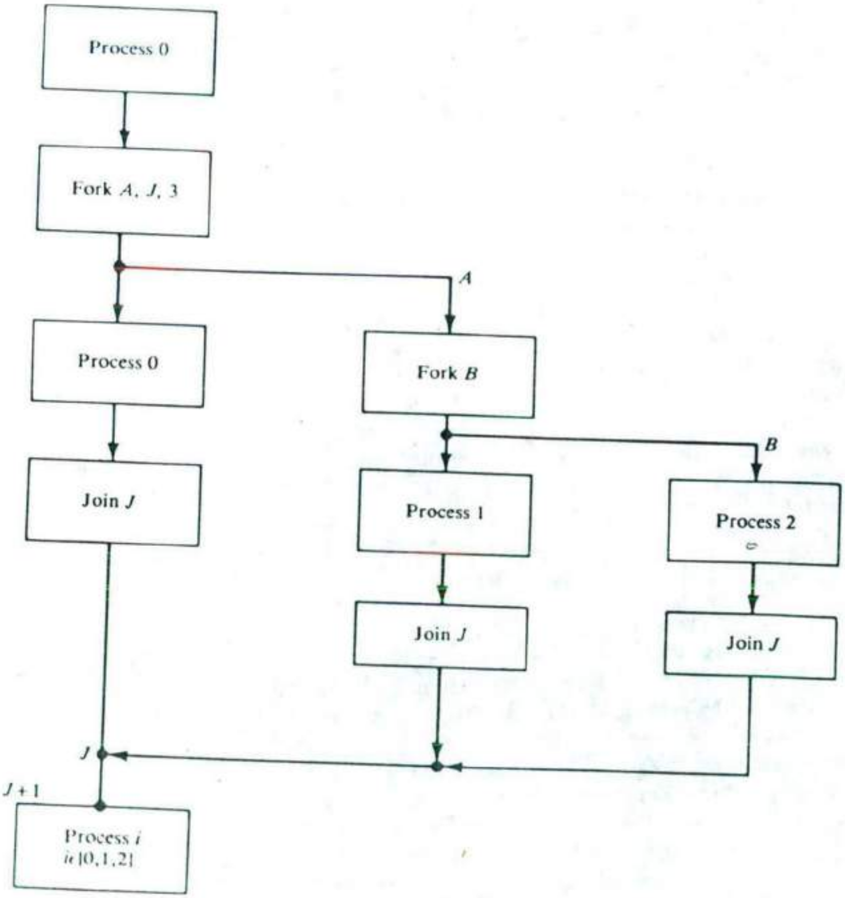


Figure 7.50 Conway's fork-join concept. (Courtesy of AFIPS Press *FJCC Proc.*, 1963.)

that, unless it is judiciously used, it blurs the distinction between statements that are executed sequentially and those that may be executed concurrently. FORK and JOIN statements are to parallel programming what the GO TO statement is to sequential programming. Also, because FORK and JOIN can appear in conditional statements and loops, a detailed understanding of program execution is necessary to enable the parallel activities. Nevertheless, when used in a disciplined manner, the statements are practical to enable parallelism explicitly. For example, FORK provides a direct mechanism for dynamic process creation, including multiple activation of the same program text.

An equivalent extension of the FORK-JOIN concept is the block-structured language originally proposed by Dijkstra. In this case, each process in a set of n processes $S_1, S_2, \cdots S_n$, can be executed concurrently by using the following **cobegin-coend** (or **parbegin-parend**) constructs:

```
begin
  $S_0$;
cobegin $S_1$; $S_2$; . . . $S_n$; coend          (7.25)
  $S_{n+1}$;
end
```

The **cobegin** declares explicitly the parts of a program that may execute concurrently. This makes it possible to distinguish between shared and local variables, which in turn makes clear from the program text the potential source of interference. Figure 7.51 illustrates the precedence graph of the concurrent program given above. In this case, the block of statements between the **cobegin-coend** are executed concurrently only after the execution of statement S_0. Statement S_{n+1} is executed only after all executions of the statements $S_1, S_2, \ldots, S_n$ have been terminated. Since a concurrent statement has a single entry and a single exit, it is well suited

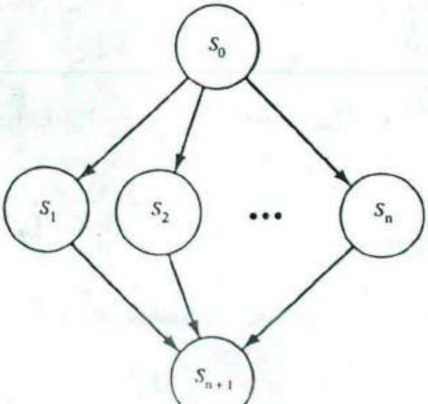


Figure 7.51 Precedence graph of the concurrent program.

to structured programming. The processes defined by the concurrent statement are completely independent of one another. The set of statements $S_1, S_2, \ldots, S_n$ are executed concurrently as disjoint processes. The disjointness implies that a variable v_i changed by statement S_i cannot be referenced by another statement S_j, where $j \neq i$. In other words, a variable subject to change by a process must be strictly private to that process, but disjoint processes can refer to common variables not changed by any of them.

Programs should be written so that it is simple for a compiler to check the degree of disjointness, possibly by scanning the program to recognize concurrent statements and variables accessed by them. The compiler must be able to distinguish between variables that can be changed by a statement and variables that can be referenced by a statement but not changed by it. These two kinds of variables are called the *variable* parameters and *constant* parameters of a statement.

To make the checking of disjointness manageable, it is often necessary to restrict the use of pointer variables and procedure parameters. For example, a pointer variable may be bound to a set of variables of a given type as

```
var i, j, k: integer: p: pointer to i or j;          (7.26)
```

This declaration indicates that variable p is a pointer to a particular set of integers. The notation enables a compiler and its run-time system to check that p always points to a variable of a well-defined type (in this case, an integer i or j).

The rule of disjointness enables the programmer to state explicitly that certain processes should be independent of one another. This depends on automatic detection of violations of this assumption. But as will be seen later, all multiprocessing systems must occasionally permit concurrent processes to exchange data in a well-defined manner. The **cobegin-coend** notation hides this communication problem from the user, but it has to be solved at some other level.

Concurrent statements can be nested arbitrarily as in the following example, which is illustrated in Figure 7.52:

```
begin
    S0;
  cobegin
    S1
    begin S2; cobegin S3; S4; S5; coend S6; end          (7.27)
    S7;
  coend
  S8;
end
```

Parallelism in the execution of statements may often be found in loops. Five primitives are listed below to allow the easy implementation of **parallel for** statements:

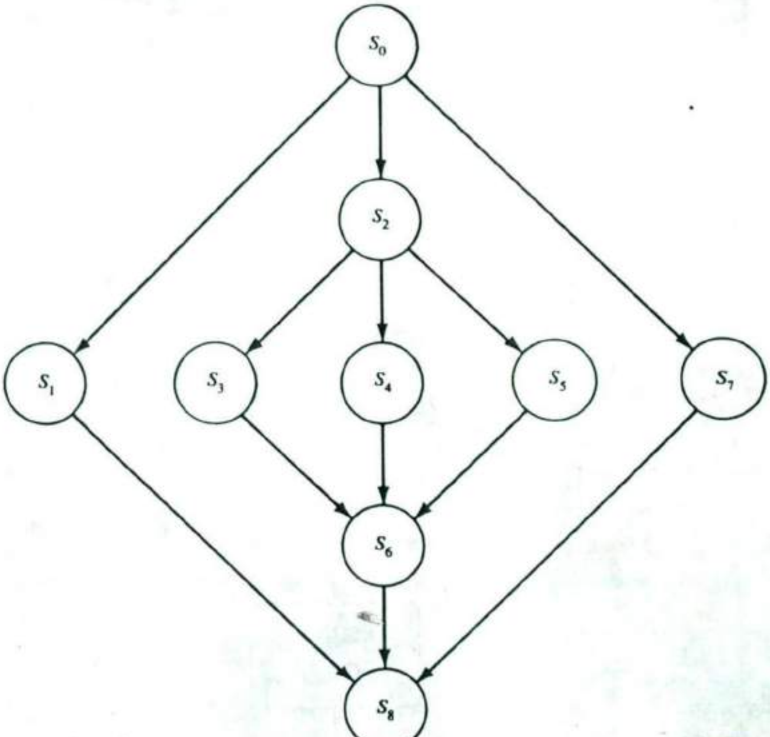


Figure 7.52 Precedence graph of nested concurrent processes.

- PREP: A parallel path counter, PPC, is initialized (PPC $\leftarrow$ 1). (A stack of PPCs is kept in case of nested loops.)
- AND(L): Two-way fork. PPC $\leftarrow$ PPC + 1. Process at address L is initiated and the current process is continued at the next instruction.
- ALSO(L): As above but without incrementing the PPC.
- JOIN: PPC $\leftarrow$ PPC $-$ 1. If PPC = 0, the PPC is "popped" and processing continues at the next instruction, else the processor executing the JOIN is released.
- IDLE: Terminates a path and releases the processor that was executing it.

Figure 7.53 shows the realization of a *parallel for* **(parfor)** statement using these instructions. Notice that statement S is executed for each value of i, and that this scheme is independent of the number of processors available in the system. Moreover, there is no need to state explicitly the relationship between the AND and JOIN primitives. Consider the matrix computation $C \leftarrow A \cdot B$, where A is

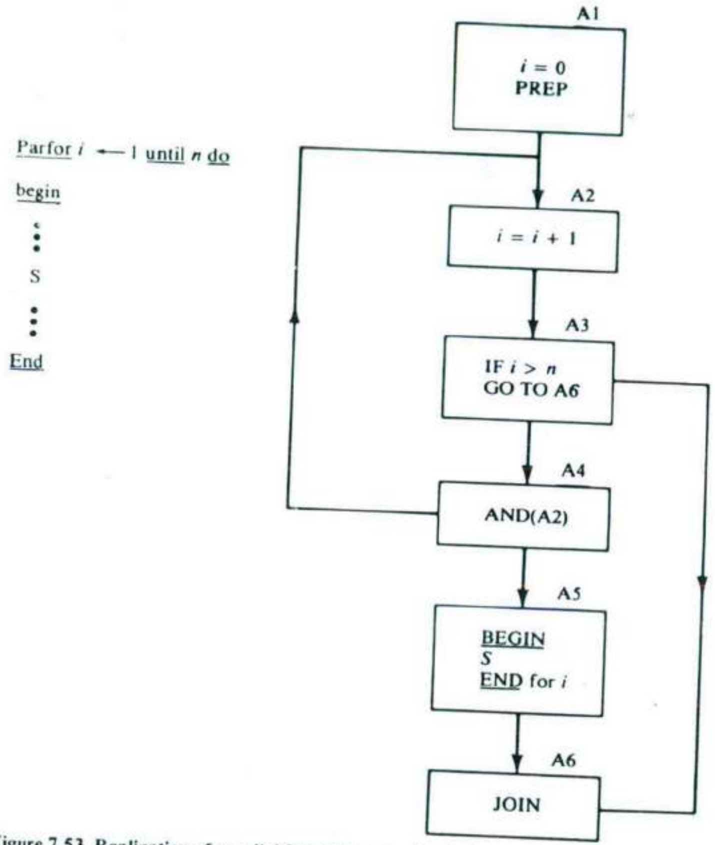


Figure 7.53 Realization of parallel for statement using defined primitives. (Courtesy of *ACM Computing Surveys*, Baer 1973.)

an $n \times n$ matrix and B and C are $n \times 1$ column vectors, for a very large n. The algorithm to compute the matrix C is given below using the **parfor** statement to spawn p independent processes. Assume that p divides n and $n/p = s$:

```
parfor i ← 1 until $p$ do
  begin
    for j ← (i − 1)$s$ + 1 until $s \cdot$ i do
    begin
      C(j) ← 0;
    for k ← 1 until $n$ do                                    (7.28)
      C(j) ← C(j) + A(j, k) · B(k);
    end
  end
```

Each process being spawned computes the statements between the outermost **begin-end** constructs for a different value of i. Hence, the computation of each group of $C(i)$ is done concurrently. Concurrent processes that access shared variables are called *communicating* processes. When processes compete for the use of shared resources, common variables are necessary to keep track of the requests for service.

A very common problem occurs when two or more concurrent processes share data which is modifiable. If a process is allowed to access a set of variables that is being updated by another process concurrently, erroneous results will occur in the computation. Therefore, controlled access of the shared variables should be required of the computations so as to guarantee that a process will have mutually exclusive access to the sections of programs and data which are nonreentrant or modifiable. Such segments of programs are called *critical* sections. The following assumptions are usually made regarding critical sections:

1. *Mutual Exclusion*: At most one process can be in a critical section at a time.
2. *Termination*: The critical section is executed in a finite time.
3. *Fair Scheduling*: A process attempting to enter the critical section will eventually do so in a finite time.

The mutually exclusive access to a set of shared variables can be accomplished by a number of constructs. An example is the MUTEXBEGIN and MUTEXEND constructs. Each construct alone does not enable a programmer to indicate whether a variable v should be private to a single process or shared by several processes. A compiler must recognize and guard any process interaction involving a shared variable v. The following is a notation used to declare a set of shared or common variables of type T: **var** v: **shared** T. Then a critical section may be defined by **csect** v **do** S. The definition associates a statement S with a common variable v and indicates that the statement S should have exclusive access to v. Critical sections referring to the same variable v exclude one another in time. By explicitly associating a critical section with the shared variable, the programmer informs the compiler of the sharing of this variable among concurrent processes, which is a deliberate exception to the rule of disjointness. At the same time, the compiler can check that a shared variable is used only inside critical sections and can generate code that implements mutual exclusion correctly. Critical sections referring to different variables can be executed in parallel, as shown in the following example:

```
var v: shared V;
var w: shared W;
cobegin                          (7.29)
  csect v do P;
  csect w do Q;
coend
```

The critical sections may also be nested as follows:

```
csect v do
begin
   . . .
   csect w do S;
   . . .
end                                        (7.30)
```

However, there is a potential danger of deadlock, in which one or more processes are blocked waiting for events that will never occur. For example, two concurrent processes P_1 and P_2 may be deadlocked in the parallel program below if P_1 enters section *v* at the same time that P_2 enters *w*.

```
cobegin
P1: csect v do csect w do S1;
P2: csect w do csect v do S2;              (7.31)
coend
```

When process P_1 tries to enter its critical section *w*, it will be delayed because P_2 is already inside its critical section *w*. And process P_2 will be delayed trying to enter its section *v* because P_1 is already inside its section *v*.

The deadlock occurs because two processes enter their critical sections in opposite order and create a situation in which each process is waiting indefinitely for the completion of a region within the other process. This *circular wait* is a condition for deadlock. The deadlock is possible because it is assumed that a resource cannot be released (preempted) by a process waiting for an allocation of another resource. From this technique, an algorithm can be designed to find a subset of resources that would incur the minimum cost if preempted. This approach means that, after each preemption, the detection algorithm must be reinvoked to check whether deadlock still exists.

A process which has a resource preempted from it must make a subsequent request for the resource to be reallocated to it. As an example, we consider a system in which one process produces and sends a sequence of data items to another process that receives and consumes them. It is an obvious constraint that these data items cannot be received faster than they are sent. To satisfy this requirement, it is sometimes necessary to delay further execution of the receiving process until the sending process produces another data item. *Synchronization* is a general term for timing constraints of this type of communication imposed on interactions between concurrent processes.

The simplest form of interaction is an exchange of timing signals between two processes. A well-known example is the use of *interrupts* to signal the completion of asynchronous peripheral operations to the processor. Another type of timing signals, *events*, was used in early multiprocessing systems to synchronize concurrent processes. When a process decides to wait for an event, the execution of its next operation is delayed until another process signals the occurrence of the event.

The following program illustrates the transmission of timing signals from one process to another by means of a shared variable *e* of type *event*. Both processes are assumed to be cyclical. Notice that the concurrent operations *wait* and *signal* both access the same variable *e*.

```
var e: shared event;
cobegin
  cycle "sender"
  begin
     . . . signal (e); . . .
  end                                        (7.32)
  cycle "receiver"
  begin
     . . . wait (e);
  end
coend
```

7.5.2 Detection of Parallelism in Programs

With reference to a sequential process, the term "parallelism" can be applied at several levels. Parallelism within a program can exist from the level of statements of procedural languages to the level of microoperations. In an MIMD multiprocessing environment, the general interest lies in parallelism of processes. The term "process" can be applied to a single statement or a group of statements which are self-contained portions of a computation.

Process parallelism can exist at a hierarchy of levels. For example, a group of statements is said to be processes at the first level. The statements within a procedure called by the main program would then be the second-level processes. If this procedure itself called another procedure, then the statements within the latter procedure would be the third level, and so on. Therefore, a sequentially organized program can be represented by a hierarchy of levels, as shown in Figure 7.54. Each block within a level represents a single process; as before, a process can represent a statement or a group of statements.

Once a sequentially organized program is resolved into its various levels, a fundamental consideration of parallel processing is recognizing processes within individual levels which can be executed in parallel. Assuming the existence of a system which can execute independent processes in parallel, this problem can be approached in two ways. We have already seen how process parallelism could be explicitly expressed by the programmer. If it is decided to make this indication independent of the programmer, then it is necessary to recognize the parallel executable processes implicitly by analysis of the source program.

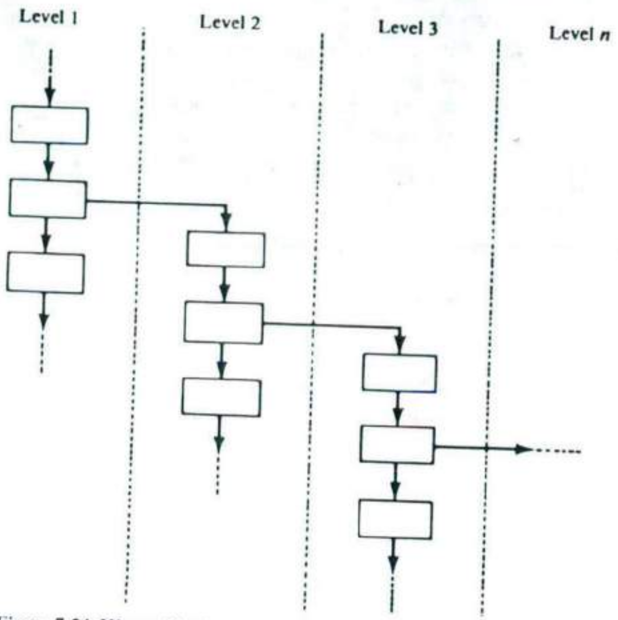


Figure 7.54 Hierarchical representation of a sequentially organized program.

Data dependency is the main factor for the interprocess detection of parallelism. Consider several statements T_i of a sequentially organized program illustrated in Figure 7.55*a*. If the execution of statement T_3 is independent of the order in which statements T_1 and T_3 are executed (Figure 7.55*a*, *b*), then parallelism is said to exist between statements T_1 and T_2. They can, therefore, be executed in parallel, as shown in Figure 7.55*c*. This *commutativity* is a necessary but not a sufficient condition for parallel execution. There may exist, for instance, two statements which can be executed in either order but not in parallel. For example, an FFT computation produces its output in a scrambled order (bit reversed) as shown in Chapter 6. Therefore, there are two ways to perform FFT computations as shown below:

1. *Method one*
   a. Bit reverse the input.
   b. Perform the FFT.
2. *Method two*
   a. Perform the FFT.
   b. Bit reverse the output.

Thus performing the FFT and bit-reversal operations are two distinct processes which can be executed in alternate order with the same result. They cannot, however, be executed in parallel.

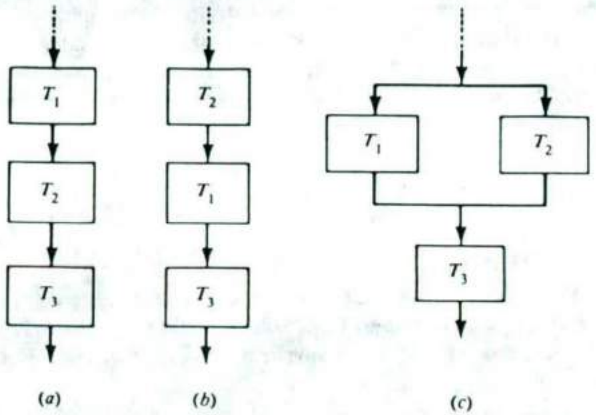


Figure 7.55 Sequential and parallel execution of a computational process.

The following *Bernstein condition* must be satisfied before sequentially organized processes can be executed in parallel. These are based on two separate sets of variables for each process T_i:

1. The *read* set I_i represents the set of all memory locations for which the first operation in T_i involving them is a fetch.
2. The *write* set O_i represents the set of all locations that are stored into in T_i.

The conditions under which two sequential processes T_1 and T_2 can be executed as two independent and concurrent processes is given below:

1. Locations in I_1 must not be destroyed by storing operations in O_2. The areas of memory from which task T_1 reads and onto which task T_2 writes should be mutually exclusive, that is,

$$I_1 \cap O_2 = \phi \tag{7.33}$$

2. By symmetry, exchanging the roles of T_1 and T_2,

$$I_2 \cap O_1 = \varphi \tag{7.34}$$

Furthermore, to maintain the state of the machine (the contents of the total memory locations) when entering T_3 independently of the mode (parallel or sequential) of execution of T_1 and T_2, I_3 must be independent of the storing operations in T_1 and T_2, that is,

$$(O_1 \cap O_2) \cap I_3 = \phi \tag{7.35}$$

If one looks at T_i as a statement of a high-level language, then I_i and O_i represent, respectively, the input (those variables which appear only at the right of an assignment statement) and output data sets of T_i, respectively. Consider the following tasks, which represent Algol statements for evaluating three matrix arithmetic expressions. A, B, C, D, X, Y, and Z are each $n \times n$ matrices.

$$T_1: X \leftarrow (A + B) * (A - B)$$

$$T_2: Y \leftarrow (C - D) * (C + D)^{-1}$$

$$T_3: Z \leftarrow X + Y$$

For tasks T_1 and T_2, $I_1 = \{A, B\}$, $I_2 = \{C, D\}$, $O_1 = \{X\}$ and $O_2 = \{Y\}$. Since $I_1 \cap O_2 = \phi$, $I_2 \cap O_1 = \phi$, and $O_1 \cap O_2 = \phi$, tasks T_1 and T_2 can be executed in parallel. However task T_3 cannot be executed in parallel with either of T_1 or T_2 since $I_3 \cap O_2 \neq \phi$ or $I_3 \cap O_1 \neq \phi$. Hence, we can write a concurrent program to execute tasks T_1, T_2, and T_3 as follows:

```
begin
  cobegin
  X ← (A + B) · (A − B);
  Y ← (C − D) · (C + D)⁻¹;                (7.36)
  coend
  Z ← X + Y;
end
```

It is based on the above conditions that systems have been written for automatic detection of parallelism in source programs written in high-level languages. But the granularity of each of the processes created is usually small. At this point, it is desirable to clarify some possible misinterpretations of the implications of this method. The method does not try to determine whether any or all of the iterations within a loop can be executed simultaneously. Rather, the iterations executed sequentially are considered as a single task. Given a Pascal FOR statement, it is possible to detect if all executions of the loop must be performed sequentially or all of them can be executed simultaneously.

The total replication test can be approached at different levels of sophistication. Let $L = \{S_1, \ldots, S_i, \ldots, S_n\}$ be the statements composing the FOR loop. Then one can form the following input and output sets: $I_L = \bigcup_{j=1}^{n} I_j, O_L = \bigcup_{j=1}^{n} O_j$, where I_j and O_j are the input and output sets formed with variables referenced within L, with each subscripted array being an individual entry. If $I_L \cap O_L = \phi$, then all loop iterations can be replicated, for example:

```
for i ← 1 until n do
  begin
    A(i) ← B(i);
    C(i) ← D(i);                (7.37)
  end
end
```

But if $I_L \cap O_L \neq \phi$, then one can look at the variables for which conflicts arise. If those are set before they are used, then the conflict is artificial and replication is permissible as, for example, in

```
for i ← 1 until n do
begin
   A(i) ← f(A(i));
   T ← g(A(i));                (7.38)
   B(i) ← h(T);
end
```

where a different T could be set aside for each replication. In practice, a compiler which incorporates an intelligent recognizer of parallelism with sufficient granularity is very difficult to implement. It is still a research problem. The recognizer often represents an overhead which may not be cost-effective for analyzing certain classes of programs in determining their parallel processability. The benefits of parallel processing obtained by using the recognizer will accrue only if the program is run many times in order that the initial overhead may be distributed over the many runs of the program.

7.5.3 Program and Algorithm Restructuring

The problem of decomposing a large program into many small concurrently executable (parallel processable) tasks has been studied for some time. Parallel processability permits faster execution times of programs and better utilization of resources in a multiprocessor computer. However, if a multiprocessor system is also capable of sequential processing of an instruction stream via a single processor, then some determination must be made as to whether or not multiprocessing will be beneficial. One of the necessary conditions of a program for parallel processing is that the program possesses many parallel paths.

A process consists of a number of computation steps. The time to execute the steps of the process is a random variable whose properties are often not well defined. When a number of processes cooperate concurrently to solve a given problem, there are a number of factors that contribute to the fluctuations in the execution time of a process. Some of these factors include memory contention, processor-scheduling policies, variations in processor speeds, and interrupts and variations in processing time due to input data distributions. This asynchronous behavior of processes leads to serious issues concerning the efficiency and correctness of the parallel algorithm. The unpredictably interleaved execution of the cooperating processes affects the correctness issue. The efficiency of the parallel algorithm may be reduced if synchronization is introduced to resolve the correctness issue. The effect of synchronization may also reduce the degree of concurrency in the algorithm.

There are a number of architectural factors that affect the decomposition of an algorithm for parallel processing. Some of these are the processor speed, memory access times, the memory bandwidth and its capacity. Depending on the architectural features, different aspects of the system may become the main bottlenecks in achieving a high level of concurrency.

The effective utilization of many multiprocessing computers is limited by the lack of a practical methodology for designing application programs to run on such computer systems. A major technique is to decompose algorithms for parallel execution. Two major issues in decomposition can be identified as partitioning and assignment. *Partitioning* is the division of an algorithm into procedures, modules and processes. *Assignment* refers to the allocation of these units to processors. These problems are among the most difficult and important in parallel processing. The emphasis below is to include communication factors as criteria for partitioning. Assignment techniques will be discussed in Section 8.4.

We will consider an example of an algorithm written for a uniprocessor and investigate the decomposition and restructuring of this algorithm for a multiprocessor system. The example is an image-processing algorithm called *histogramming*. A typical picture is represented by a rectangular array of picture elements (pixels). Each pixel has a small integer value (8 bits) between 0 and $b - 1$ (inclusive) that represents a gray scale value of a black-and-white picture or, for color images, the intensity of a primary color. Typically, $2 \le b \le 256$. Histogramming involves keeping track of the frequency of occurrence of each gray scale value; thus, for 8-bit pixels, b such frequencies (simple counters) must be maintained. Let *histog* $[0:b - 1]$ represent the array that keeps the count for the number of occurrences of the gray scale value $0, 1, \ldots, b - 1$. The rectangular picture is represented by the two-dimensional array of picture elements pixel $[0:m - 1, 0:n - 1]$, with m rows and n columns. If pixel $[i, j]$ represents the gray scale value of the pixel at coordinate i, j, then the following serially coded program will update the histogram to include the pixel at i, j:

```
    var pixel [0:m − 1, 0:n − 1];
    var histog[0:b − 1): integer;
initial histog[0:b − 1] = 0; //initialize frequency counters//          (7.39)
    for i ← 0 until m − 1 do
      for j ← 0 until n − 1 do
        histog[pixel[i, j]] ← histog[pixel[i, j]] + 1;
```

The time complexity of this program is $O(mn)$.

This program can be restructured by realizing that the operation is iterative and scans a row of the image in the inner loop. Since there are m rows of the image, we can partition the image into p nonoverlapping and equal segments (assuming p divides m), where each segment has $m/p = s$ rows. We can then spawn a set of p processes to histogram the whole image, where each process is assigned to histogram a distinct segment of the image. However, the constraint is that all processes cooperate to form the histogram of the whole image; that is, they share the histog $[0:b - 1]$ array in updating the frequency counters. The degree of decomposition is thus p. In general, the value of p affects the performance of the algorithm. Using the **parfor** statement, the parallel algorithm to histogram the image may be written as follows:

```
var histog[0:b - 1]: integer: shared; //declare shared variables//
initial histog[0:b - 1] = 0; //initialize frequency counters//
  parfor i ← 1 until p do
    begin
        var pixel[(i - 1)s:  i - 1, 0:n - 1]: pixel;                (7.40)
            for k ← (i - 1)s until si - 1 do
              for j ← 0 until n - 1 do
                csect histog[pixel[k, j]] do
                histog[pixel[k, j]] ← histog[pixel[k, j]] + 1;
    end
```

Figure 7.56 illustrates the partitioning of the histogramming problem. Since the p processes share the histog $[0{:}b - 1]$ frequency counters, there may be a considerable degree of memory contention, assuming a tightly coupled system.

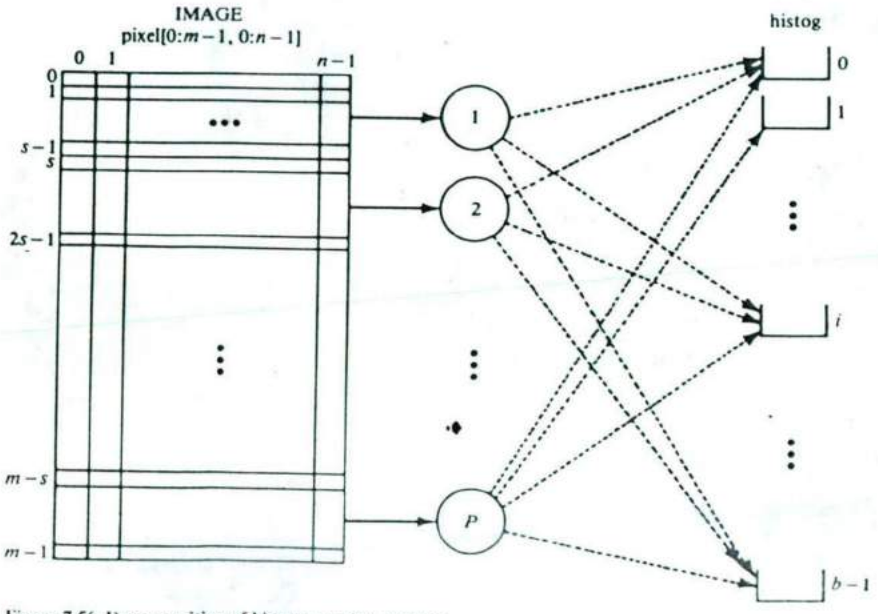


Figure 7.56 Decomposition of histogramming problem.

Also, the updating of each "*histog*" counter must be done in a mutually exclusive manner to avoid incorrect results. Hence, the counter update statement must be enclosed as a critical section. The degree of decomposition influences the degree of contention in this case. Without memory contention, the potential time complexity of each process is $O(sn)$. Since $s = m/p$, the decomposition of the problem into p processes has a potential speedup of p. This speedup is never achievable in practice, however.

We illustrate the effect of the placement of the process code, picture segments, and bins by considering the execution of this algorithm on three different multiprocessor architectures, each with p processors. In the first case, each of the p PEs consists of a processor and its local memory, which is attached to a time shared bus. In addition, a main memory, which is also attached to the bus, is shared by all the processors. Each PE contains the process code and its segment of the picture. The b bins are stored in the main memory. Since each PE contains a segment of the picture, the processors access the pixels without conflict. However, this architecture will cause excessive conflicts to the main memory because of concurrent accesses to the bins.

In the second case, the processors share the main memory through a crossbar switch. The process code and the entire picture elements are in memory. The bins are distributed across the memory modules. Therefore, in addition to conflicts of accesses to the process code and pixels, there are also conflicts of accesses to the bins.

In the third case, each processor of the second case has a private cache. Assume that the process code is small enough to reside in the cache; hence, we have faster access to code. Also assume that the cache is not large enough, so that the blocks of read only pixels are fetched into the cache on demand. The pixels are thus accessed at slightly faster than main memory speeds. However, the bins are shared writeable memory locations. Therefore, accesses to them cause excessive "ping-ponging" as copies of these bins are bounced from one cache to the other because of references for updates.

The effect of ping-ponging is considered by accesses to the bin histog[k] in a cache with write-back memory update policy. Suppose a remote cache C_j has the latest copy of bin k, which is now referenced by a local cache C_i. The reference in C_i results in a miss and causes C_j to update memory copy of bin k and also invalidate C_j's copy of bin k. Furthermore, C_i gets the copy of bin k from memory and increments it. In effect, processor i waits for two memory cycles each time a reference to a bin results in a miss when another processor has a copy of it. A subsequent reference by another processor to this bin ping-pongs the bin to the processor's cache.

If the degree of decomposition is greater than the number of available processors, the processing time of the histogramming problem is as slow as the completion time of the last of the p processes. The memory contention and mutual exclusion problems could be eliminated if each process has b bins to generate its local histogram for its segment of the picture in its local memory. The algorithm below illustrates the modification required to eliminate these problems.

```
var histog[0:b − 1]: integer; initial histog[0:b − 1] = 0;
var lhistog[1:;, 0:b − 1]: integer; //local bins//
initial lhistog[1:p, 0:b − 1] = 0;
parfor i ← 1 until p do
begin
  var pixel[(i − 1)s:si + 1, 0:n − 1]:pixel;
  var lhistog[i, 0:b − 1]:integer;                              (7.41)
  for k ← (i − 1)s until si + 1 do
    for j ← until n − 1 do
       lhistog[i, pixel[k, j]] ← lhistog[i, pixel[k, j]] + 1;
end
for j ← 0 until b − 1 do //sum individual histograms//
  for i ← 1 until p do
    histog[j] ← histog[j] + lhistog[i, j];
```

However, extra overhead for synchronization is needed after the completion of all p processes to sum the individual histograms to obtain the overall histogram. The cost of the overhead is constant regardless of the size of the picture being processed. Slightly more local memory space is required to store the individual histograms, but this too is independent of the size of the picture being processed.

In general, the efficient implementation of an algorithm in a particular machine is largely shaped by the architecture of that machine. Relatively subtle changes in architecture can have extensive changes in the performance of the algorithm. The relationship between communication and computation is an important factor in designing effective parallel algorithms. One of the motivations for including communication in designing parallel programs is that the communication time can be greater than the computation time, based on data dependencies alone. The way in which data is distributed among processor memories can have a significant effect on the amount of required communication. By choosing the right data distribution, one may be able to design algorithms requiring less communication and thus reduce execution time.

The histogramming problem is unique in that a single copy of the data structure must be continually updated. If, for example, 2^{20} pixels (a 1024×1024 image) must be placed in b bins, the memory-contention problems in a parallel architecture can be formidable. As the processor speed increases relative to memory speed, it becomes necessary to reorganize the algorithms in which the number of data references made is minimized. For a machine like the C.mmp, where instruction execution and data references take about the same amount of time, an attempt to minimize the total number of executed instructions and data references should be equally made. As the processor speed increases, a larger proportion of the total time is spent in referencing data. This term soon dominates the total time and must be reduced in order to obtain significant speedup.

Also, as the capacity of the shared memory decreases, the algorithm should be decomposed to minimize the number of page faults between shared memory and disk. This means avoiding multiple passes through the data which increases the frequency of disk accesses. It is desirable to execute as many instructions as possible on the data in shared memory before bringing in a new page. In spite of this, if little computation is involved, an algorithm which makes efficient use of the data in shared memory will still run slowly, since the processor is forced to wait more often for data to be brought in from mass storage. A decrease in the bandwidth of the shared memory disk link or an increase in the disk access time will necessitate the same type of reorganization. Since more time is required to fill a buffer in shared memory, the algorithm must do all the computing that it can before initiating another transfer to or from file memory. Careful decomposition of the algorithms to be implemented on a machine may permit the use of slower disks and lower bandwidth channels between shared memory and disk.

In discussing the various approaches to partitioning, we emphasize the role of communication in program partitioning techniques. Unless the communication problem is effectively solved, delays due to communication bottlenecks may result. Partitioning methodologies for parallel programs evolved from two sources: (*a*) extensions to concepts that play a central role in sequential program design, especially data abstraction and information hiding concepts, and (*b*) techniques for synchronizing concurrent processes. The combination of the two sources has led to the concept of structuring systems as a set of concurrent processes that interact through monitors, which are discussed in Chapter 8. These approaches have been successful for real-time systems and operating systems designs which are based on a multiprogrammed uniprocessor system.

We formally define the partitioning problem as follows: Given an algorithm, specify the set of program units (modules or processes) that will implement the algorithm on a specified multiprocessor system in the most efficient manner. Efficiency can be measured by such criteria as the utilization of the processor or the speedup of the algorithm. One general partitioning guideline is based on the concept of *computation-communication* trade-offs. This concept is similar to the idea of space-time trade-offs in sequential programming. In partitioning, one can attempt to reduce the communication complexity by increasing the computation complexity. Another general guideline for partitioning algorithms is *clustering*. This method can be effectively applied to loosely coupled processing nodes. The basic idea is to form groups of modules in which the number of message transfers within groups is much greater than the number of transfers between groups.

A partitioning technique termed *recursive* or *iterative compute-combine* method can also be used in some cases. This approach is applicable to a broad class of problems. Examples are sorting and computing the maximum of a vector or unimodal function. The structure of the algorithm is a tree with the leaves representing the basic computations and the internal nodes representing the combination of results produced by the node descendants. Another technique is the *large army* technique. Taking advantage of parallel processing, one creates a large number of processes, each performing precisely the same function, such as in a search operation. When one of the processes achieves the collective goal, it notifies the others to cease the iteration and to reinitialize themselves for further iterations, if there are any.

Multiprocessors may not provide an effective means of increasing the execution speed of certain computationally expensive algorithms. The algorithm under examination may not lend itself to decomposition for a multiprocessing environment. The problems come from one or more critical sections of unevenly sized codes, which have to update a common data base (e.g., the same copy of a histogram). In these cases, synchronization can introduce significant overhead and the time lost by processes waiting for the same memory location may be large. Here, a uniprocessor which is N times as fast would be preferable to an N-processor system.

Conversely, algorithms which allow favorable scheduling of processes and/or the data to be partitioned into independent chunks often can run N times as fast given N processes. The extra synchronization overhead is small compared to the increase in the number of instructions executed per second. Maximum performance from an algorithm can only be obtained when it has been coded with the particular machine architecture in mind. Effective decomposition of algorithms is dependent on the programmer knowing the hardware capabilities and coding the algorithm to take advantage of the machine's capabilities.

7.6 BIBLIOGRAPHIC NOTES AND PROBLEMS

Multiprocessor hardware organizations and operating system configurations were surveyed in Enslow (1974, 1977). Commercial multiprocessors were comparatively studied in Satyanarayanan (1980), in which an annotated bibliography was given. Theoretical aspects of multiprocessor systems were discussed in Baer (1973, 1976). Experience using multiprocessor systems was summarized in Jones and Schwarz (1980). Reliability modeling of various multiprocessor architectures can be found in Hwang and Chang (1982). The Cm* material is based on the reports by Jones and Crehringer (1980).

An example of a crossbar design can be found in Wulf et al. (1981). Bain and Ahuja (1981) provided a comprehensive treatment of various arbitration algorithms for single bus structures. Patel (1981) introduced the delta network. Studies on packet switched networks can be found in Dias and Jump (1981) and in Chin and Hwang (1984). The Banyan networks were studied by Goke and Lipovski (1973). Parallel memory organizations for multiprocessing were treated by Briggs and Davidson (1977) and their uses in a system with caches by Briggs and Dubois (1983) and Yeh et al. (1983). The home memory concept was due to Smith (1978). Multicache coherence problems were studied by Dubois and Briggs (1982) and Censier and Feautrier (1978).

Concurrent programming techniques can be found in Andrews and Schneider (1983). Communication issues in developing parallel algorithms were studied in Lint and Agerwala (1981). An example of a multiprocessor operating system was given in Ousterhoust et al. (1980) for the Cm* system. Operating systems for multiprocessors were also treated in Ritchie (1973), Sites (1980), Habermann (1976), and Denning (1976). Load balancing among multiple processors was modeled in Ni and Hwang (1983). The FORK-JOIN concept is due to Conway

(1963). The primitives used to implement a PARALLEL FOR were studied in Gosden (1966). Synchronization primitives in concurrent systems are proposed by Dijkstra (1965, 1968). The extensions of semaphores can be found in Hoare (1974) and Hansen (1977). Automatic detection of parallelism can be found in Baer (1973). The conditions that determine whether two tasks can be executed in parallel were proved in Bernstein (1966). Techniques for recognition of parallel processable tasks were presented in Russel (1969) and Ramamoorthy and Gonzalez (1969). Program decomposition for multiprocessors is discussed in Hon and Reddy (1977).

Problems

7.1 Briefly describe the following terms associated with a multiprocessor system:
- (*a*) Multiple computer system
- (*b*) Multiprocessor system
- (*c*) Loosely coupled multiprocessors
- (*d*) Tightly coupled multiprocessors
- (*e*) Homogeneous multiprocessors
- (*f*) Heterogeneous multiprocessors
- (*g*) A cluster of computer modules
- (*h*) Private caches versus shared caches
- (*i*) Context switching
- (*j*) Semaphore for synchronization
- (*k*) Time shared common buses
- (*l*) Crossbar switches
- (*m*) Multiport memory
- (*n*) The delta network
- (*o*) The L-M memory organization
- (*p*) Multicache coherence problem
- (*q*) Master-slave operating system
- (*r*) Floating-supervisor system
- (*s*) Mutual exclusion between processes
- (*t*) Bus-arbitration algorithms
- (*u*) Explicit parallelism
- (*v*) Critical sections
- (*w*) Computation-communication trade-offs

7.2 Assume a uniprocessor with cache. The main memory consists of 16 modules which can be interleaved in various ways. For each case below, indicate the number of memory cycles required per block transfer. Also indicate the reliability of each system by an integer *k*. A system is said to be *k*-reliable with respect to memory if it can keep functioning after *k* modules *taken at random* are disconnected. In the table, LSB means the least significant bit and MSB the most significant bit.

Interleaving	Block size	Bus width	Number of memory cycles	Reliability
4LSB	16 words	1 word		
2MSB-2LSB	16 words	8 words		
4	8 words	8 words		
1MS	4 words	8 words		
3MSB-1LSB	4 words	2 words		

7.3 The functions $S_{q,r}(i)$ and $f_{q,r}(i)$ from the Delta networks are defined as follows:

$$S_{q,r}(i) = \left(qi + \left\lfloor \frac{i}{r} \right\rfloor\right) \bmod qr \qquad \text{for } 0 \le i \le qr - 1$$

and

$$f_{q,r}(i) = \begin{cases} qi \bmod (qr-1) & \text{for } 0 \le i \le qr - 2 \\ i & \text{for } i = qr - 1 \end{cases}$$

(*a*) Prove that $S_{q,r}(i) = f_{q,r}(i)$, $\forall$, $0 \le i \le qr - 1$.
(Hint: i can be written as $i = k_1 r + k_2$, where $0 \le k_2 < r$).

(*b*) Then show that $f_{r,q}$ is the inverse of $f_{q,r}$. That is,

$$f_{r,q}(f_{q,r}(i)) = i$$

(*c*) Consider an $a^n \times b^n$ delta network which is implemented with $a \times b$ switch boxes.

(1) How many $a \times b$ switch boxes are required?

(2) What is the delay through the network if the delay through one box is D?

7.4 A three-processor system uses three multiport memory modules in a shared memory system in which each processor can access each memory module. Memory modules are assumed to be 100 percent reliable, but the processors fail with distressing frequency. 84 percent of the time, all three processors are working correctly, 15 percent of the time only two processors are usable, and 1 percent of the time only a single processor is functioning. The fraction of time during which no processor is working properly is negligible.

What is the average throughput in this system, as measured by the average number of memory modules active during a memory cycle? Use all of the assumptions of the multiprocessor model presented first in class. In particular, memory requests queued at a module at the start of a memory cycle but not serviced during that cycle remain queued at the module until they are eventually serviced. Do not worry about what happens immediately after a processor fails or is repaired; i.e., assume that the system is in steady state (with one, two, or three processors functioning) virtually all of the time.

7.5 In the common bus (time shared bus) organization shown in Figure 7.57, there are p parallel-pipeline processors and m memory modules. The arbiter randomly selects a request from the processors and puts it on the bus. After one time unit delay, the address (and data, if any) reaches all the memory parts. The appropriate module then gates-in the information. The module goes through a memory cycle of c time units long. The arbiter does not issue the selected request at time t if the addressed module is still going to be busy at time $t + 1$. Based on this description and the following assumptions, derive the memory bandwidth (average number of requests accepted per memory cycle) of this organization. Memory requests are random and uniformly distributed among all modules. The rejected requests are ignored. One request is issued by each processor at each time unit. No data dependency exists between any two requests.

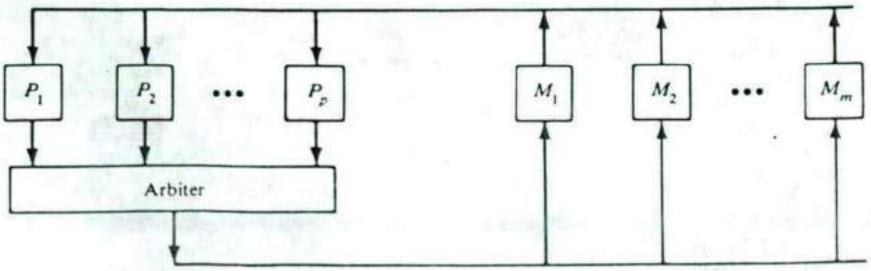


Figure 7.57 A bus-structured multiprocessor system with a centralized bus arbiter.

7.6 Consider a $p \times m$ crossbar switch connecting p processors to m memory modules. Assume only one input AND gate and OR gate (no wired-OR). Assume also that all variables are available in true and complemented form.

(*a*) Estimate the number of gates in the switch, ignoring the decoders and the arbiter. Assume the data width to be w bits.

(*b*) Design the decoder and arbiter for the above crossbar. Assume that the processor P_i has priority over P_j if $i < j$. Estimate the number of gates for this circuit.

7.7 We have considered an 8×8 delta network in Figure 7.35. Answer the following questions related to this interconnection network.

(*a*) Does the network have a path between any processor and any memory module?

(*b*) Let $(d_2 d_1 d_0)_2$ be the address of a memory module (MM) generated by a processor whose number is $(p_2 p_1 p_0)_2$. Let the control variables at stages 0, 1, and 2 be x_0, x_1, and x_2 respectively. The convention is

$$x_i = \begin{cases} 0 & \text{for straight connection} \\ 1 & \text{for crossed connection} \end{cases}$$

The requested MM address is passed through the successive stages to set up the path. Find the logic equations for x_0, x_1, x_2 as functions of d_0, d_1, d_2, and p_0, p_1, and p_2.

(*c*) Assume that processor zero accesses MM2, processor four accesses MM4, and processor six accesses MM3. Show the paths for these three requests on Figure 7.35. Do these requests conflict?

7.8 Briefly characterize the *multicache coherence* problem and describe various methods that have been suggested to cope with the problem. Comment on the advantages and disadvantages of each method to preserve the coherence among multiple shared caches used in a multiprocessor system.

7.9 Distinguish among the following operating system configurations for multiprocessor computers. In each system configuration, name two example multiprocessor computers that have implemented an operating system similar to the configuration being discussed. Comment on the advantages, design problems, and shortcomings in each operating system configuration.

(*a*) Master-slave operating system.

(*b*) Separate supervisor system per processor.

(*c*) Floating-point supervisor system.

7.10 Consider a computer with four processors P_1, P_2, P_3, P_4 and six memory modules M_1, $M_2, \ldots, M_6$. The four processors can be configured as an MIMD machine or as an SIMD machine. The illustrated memory access patterns are generated by the processors for the computation of six instructions with the data dependence graph shown in Figure 7.57. Each instruction needs to access the memory modules in, at most, three consecutive memory cycles. For SIMD mode, only the same instruction can access different modules simultaneously. For MIMD mode, no such restriction exists. When two or more processors access the same module in the same cycle, the request of the lower-numbered processor is granted, and the rest of the requests must wait for a later memory cycle.

(*a*) For MIMD operation, different instructions can be executed by the four processors at the same time, subject only to data dependency. What is the average *memory bandwidth* (words/cycle) used in the execution of the above program in MIMD mode?

(*b*) Repeat the same question for using the computer in SIMD mode. The four processors must execute the same instruction at the same time under SIMD mode.

7.11 Supposing each task is an *assignment statement*, restructure the following assignment statements using Bernstein's conditions so that we have maximum parallelism among tasks. Specify which of the three conditions you are using for the restructuring.

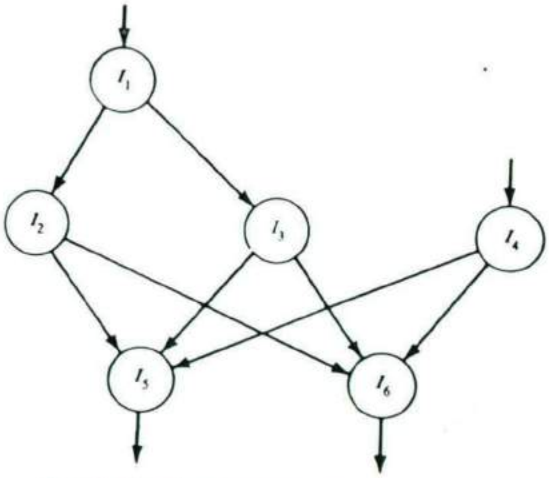
$$A = B + C$$

$$C = D + E$$

$$F = G + E$$

$$C = L + M$$

$$M = G + C$$



The data-dependency graph for the six instructions

Instructions \ Processors	P_1	P_2	P_3	P_4
I_1	M_2	M_4	M_3	M_2
I_2	M_1	M_3	M_2	M_6
I_3	M_4	M_6	M_5	M_6
I_4	M_3	M_4	M_4	M_2
I_5	M_2	M_2	M_2	M_1
I_6	M_1	M_5	M_5	M_1

(Entries are memory modules being requested to access by the processors)

Figure 7.58 Program graph and the memory access pattern for either an SIMD computer with 4 PEs or an MIMD multiprocessor with 4 processors in Problem 7.10.

Write your answer so that, when two assignment statements are put on the same line, it indicates that they can be processed in parallel. It is easier if you reset the order of precedence in a precedence graph. (Note: The statements executed after the restructuring should have the same result as the statements executed before restructuring.)

7.12 A *parallel computation* on an n-processor system can be characterized by a pair $\langle P(n), T(n)\rangle$, where $P(n)$ is the total number of unit operations to be performed and $T(n)$ is the total execution time in steps by the system. In a *serial computation* on a uniprocessor with $n = 1$, one can write $T(1) = P(1)$, because each unit operation requires one step to be executed. In general, we have $T(n) < P(n)$, if there is more than one operation to be performed per step by n processors, where $n \geq 2$. Five performance indices have been suggested below by Lee (1980) in comparing a parallel computation with a serial computation.

$$S(n) = \frac{T(1)}{T(n)} \qquad \text{(The speedup)}$$

$$E(n) = \frac{T(1)}{n \cdot T(n)} \qquad \text{(The efficiency)}$$

$$R(n) = \frac{P(n)}{P(1)} \qquad \text{(The redundancy)}$$

$$U(n) = \frac{P(n)}{n \cdot T(n)} \qquad \text{(The utilization)}$$

$$Q(n) = \frac{T^3(1)}{n \cdot T^2(n) \cdot P(n)} \qquad \text{(The quality)}$$

(*a*) Prove that the following relationships hold in all possible comparisons of parallel to serial computations.

(1) $1 \leq S(n) \leq n$

(2) $E(n) = \dfrac{S(n)}{n}$

(3) $U(n) = R(n) \cdot E(n)$

(4) $Q(n) = \dfrac{S(n) \cdot E(n)}{R(n)}$

(5) $\dfrac{1}{n} \leq E(n) \leq U(n) \leq 1$

(6) $1 \leq R(n) \leq \dfrac{1}{E(n)} \leq n$

(7) $Q(n) \leq S(n) \leq n$

(*b*) Based on the above definitions and relationships, given physical meanings of these performance indices.

CHAPTER

EIGHT

MULTIPROCESSING CONTROL AND ALGORITHMS

This chapter covers interprocess synchronization mechanisms, system deadlocks, protection schemes, multiprocessor scheduling, and parallel algorithms. These are important topics in developing a sophisticated operating system for a multiprocessor system. The parallel algorithms form the basis in using MIMD computers. For other related issues that have not been covered below, readers are advised to check the attached bibliographic notes.

8.1 INTERPROCESS COMMUNICATION MECHANISMS

Various interprocess communication schemes have been proposed by computer designers. This section enumerates some of the process-synchronization mechanisms implementable at the instruction level. High-level mechanisms such as the **P** and **V** primitives and conditional critical regions are then presented. Examples are given on the producer-consumer processes and the reader-writer problem using these mechanisms. The extension of conditional critical regions to monitors is also discussed.

8.1.1 Process Synchronization Mechanisms

Cooperating processes in a multiprocessor environment must often communicate and synchronize. Execution of one process can influence the other via communication. Interprocess communication employs one of two schemes: use of shared variables or message passing. Often the processes that communicate do so via a synchronization mechanism. A process executes with unpredictable speed and generates actions or events which must be recognized by another cooperating process. The set of constraints on the ordering of these events constitutes the set of synchronization required for the operating processes. The synchronization mechanism is used to delay execution of a process in order to satisfy such constraints.

Two types of synchronization are commonly employed when using shared variables. These are *mutual exclusion* and *condition synchronization*. We recall that mutual exclusion ensures that a physical or virtual resource is held indivisibly. Another situation occurs in a set of cooperating processes when a shared data object is in a state that is inappropriate for executing a given operation. Any process which attempts such an operation should be delayed until the state of the data object changes to the desired value as a result of other processes being executed. This type of synchronization is sometimes called condition synchronization. The mutual-exclusive execution of a critical section, *S*, whose access is controlled by a variable gate can be enforced by an entry protocol denoted by MUTEXBEGIN (gate) and an exit protocol denoted by MUTEXEND (gate). Alternatively, the effect of the entry and exit protocols can be expressed as **csect** gate **do** S.

There are certain problems associated with implementing the MUTEXBEGIN/MUTEXEND construct. Execution of the MUTEXBEGIN statement should detect the status of the critical section. If it is busy, the process attempting to enter the critical section must wait. This can be done by setting an indicator to show that a process is currently in the critical section. Execution of the MUTEXEND statement should reset the status of the critical section to idle and provide a mechanism to schedule the waiting process to use the *critical section* (CS). One implementation is the use of the LOCK and UNLOCK operations to correspond to MUTEXBEGIN and MUTEXEND respectively. For these, consider that there is a single *gate* that each process must pass through to enter a CS and also leave it. If a process attempting to enter the CS finds the gate unlocked (open) it locks (closes) it as it enters the CS in one indivisible operation so that all other processes attempting to enter the CS will find the gate locked. On completion, the process unlocks the gate and exits from the CS. Assuming that the variable *gate* = 0(1) means that the gate is open (closed), the access to a CS controlled by the gate can be written as

```
LOCK (gate)
execute critical section
UNLOCK (gate)
```

The LOCK (x) operation may be implemented as follows:

```
        var x: shared integer;
LOCK (x): begin
             var y: integer;
               y ← x;
             while y = 1 do y ← x; // wait until gate is open //
               x ← 1; // set gate to unavailable status //
             end
```

The UNLOCK(x) operation may be implemented as

```
UNLOCK(x): x ← 0;
```

The LOCK mechanism as shown is not satisfactory because two or more processes may find $x = 0$ before one reaches the $x \leftarrow 1$ statement. This can be remedied if the processor has an instruction that both tests and sets (modifies) a word. Such an instruction, called TEST_AND_SET(x) and available on the IBM S/370, tests and sets a shared variable x in a single *read-modify-write* memory cycle to produce a variable y. The read-modify-write operation must take place in one cycle so that the memory location, x, is not accessed and modified by another processor before the current processor completes the test-and-set operation. The indivisibility is usually accomplished by the requesting processor which holds the bus until the cycle is completed. Therefore the set of operations $\{y \leftarrow x; x \leftarrow 1\}$ is indivisible in the following definition of TEST_AND_SET(x):

```
                 var x: shared integer;
      TEST_AND_SET(x): begin
                            var y: integer;
                            y ← x;
                            If y = 0 then x ← 1;
                            end
```

The LOCK operation may be rewritten as

```
         var x: shared integer;
   LOCK(x): begin
            var y: integer;
               Repeat {y ← TEST_AND_SET(x)} until y = 0;
            end
```

An important property of locks is that a process does not relinquish the processor on which it is executing while it is waiting for a lock held by another process. Thus, it is able to resume execution very quickly when the lock becomes available. However, this property may create problems for the error-recovery mechanism of the system when the processor which is executing the lock fails. The error-recovery procedure has to be sophisticated enough to ensure that deadlocks are not introduced as a result of the recovery process itself.

Another instruction used to enforce mutual exclusion of access to a shared variable in memory location *m*_addr is the *compare-and-swap* (CAS) instruction. This instruction is available on the IBM 370/168. A typical syntax of this instruction uses the two additional operands *r*_old and *r*_new, which are processor registers

(CAS *r*_old, *r*_new, *m*_addr). The action of the CAS instruction is defined as follows:

```
   var m_addr: shared address;
   var r_old, r new: registers;
   var z: CAS flag;
CAS: if r_old = m_addr then
      {m_addr ← r_new; z ← 1}
   else
      {r_old ← m addr; z ← 0}
```

Notice that associated with the CAS instruction is a processor flag *z*. The flag is set if the comparison indicates equality. Again, the execution of the CAS instruction (that is, the IF statement) is an indivisible operation. We illustrate the use of the CAS instruction with a shared singly linked queue data structure (Figure 8.1), which is accessed concurrently by the two processes P_1 and P_2. The two operations which can be performed on the queue are ENQUEUE(X) and DEQUEUE. ENQUEUE(X) adds a node X to the "TAIL" of the queue and DEQUEUE returns a pointer to the deleted "HEAD" of the queue. HEAD and TAIL are shared global variables. Assuming that the queue is never empty (for simplicity), the ENQUEUE(X) primitive for a nonconcurrent system can be described as

```
Procedure ENQUEUE(X);
var P: pointer;    //P is local to each invocation //
begin
LINK(X) ← Λ;       //terminate last node's link//
P ← TAIL;
TAIL ← X;
LINK(P) ← X;       //attach new node to queue//
end
```

Suppose process P_1 requests to enqueue node X. While P_1 is executing the primitive, it gets interrupted by P_2, which requests to enqueue node Y to the same queue. Assume that the interruption occurs at the end of statement $P \leftarrow$ TAIL. Figure 8.1*a* illustrates the state of the queue at the time of interruption. If P_1 executes the procedure to completion after P_2 returns control to P_1, node X will be attached to the queue. However, node Y, which was added by P_2, would have been detached from the queue unintentionally. This error occurs because pointer *P* was not updated to point to the last node attached by process P_2. We can avoid this problem by using the CAS instruction to update *P* to point to the last attached node. This can be accomplished by replacing the TAIL ← X statement with

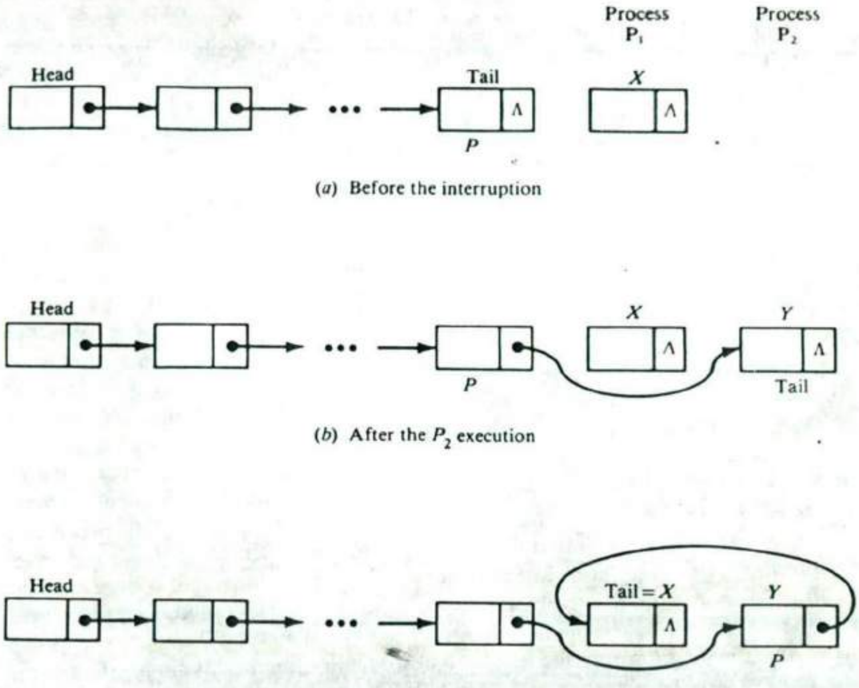

Process P₁
Process P₂
Head
Tail
X
P
(a) Before the interruption
Head
X
Y
P
Tail
(b) After the P₂ execution
Head
Tail = X
Y
P
(c) Followed by the P₁ execution

Figure 8.1 Interleaved execution of ENQUEUE by process P_1 and process P_2.

"**repeat** CAS P, X, TAIL **until** TAIL = X". The modified ENQUEUE(X) primitive is shown below:

```
Procedure ENQUEUE(X);
var P: pointer;
begin
   LINK(X) ← Λ;
   P ← TAIL;
   repeat CAS P,X,TAIL until TAIL = X;
   LINK(P) ← X;
   end
```

The CAS instruction ensures that the logical state (P) of the interrupted program is maintained on resumption of the interrupted program. Otherwise it updates the state P to the most recent value of TAIL.

Figure 8.1*b* shows the outcome of the execution of the primitive by P_2 followed by the completion of the execution of the primitive by P_1 (Figure 8.1*c*). The CAS instruction is more useful than the test-and-set instruction. An extension of the CAS instruction is the *compare double and swap*, also available on the IBM 370/168. There are other variations which enforce mutual exclusion. For example, the Honeywell 60/66 has the *load-a*ccumulator-and-*c*lear-memory-location (LDAC) instruction.

The LOCK instruction using TAS has a drawback in that processes attempting to enter critical sections are busy accessing and testing common variables. This is called *busy-wait* or *spin-lock*, which results in performance degradation. The process cannot normally be context-swapped off its processor while it is waiting. Hence, the processor is said to be *locked out*. Such lock-out is only permitted in supervisor mode. In general, LOCK and UNLOCK primitives are not usually allowed to be executed in user mode because the user process may be swapped out while holding a critical section. On the other hand, if the user makes a supervisor call each time it attempts to access a critical section, the overhead will be greatly increased. Hence the CAS instruction was provided as an excellent mechanism of letting the user do some synchronization in user mode.

The performance degradation due to spin-locks is two-fold. When a processor is spinning, it actively consumes memory bandwidth that might otherwise have been used more constructively. If the spinning period is too long, a processor is not effectively utilized during that period. A number of methods have been proposed to reduce the degradation due to spin-locks. The first method is aimed at reducing the request rate to memory and, hence, the degree of memory conflicts. This is accomplished by delaying the reissuance of the lock request for an interval *T*. Thus, the LOCK(x) primitive, for example, can be modified as

```
LOCK(x): begin
             y ← TEST-AND-SET(x);
             while y ≠ 0 do
                begin
                   PAUSE(T); //
                   y ← TEST-AND-SET(x);
                end
             end
```

Note that the processor issuing the request may not be released unless *T* is large enough. The choice for *T* depends on the granularity of the resource being requested.

The second method is directed at relieving the processor of performing the lock access by incorporating a separate mechanism which processes lock requests. This can be accomplished in one of several ways. For example, the mechanism can continuously access the lock until it is available, as in the HEP machine.

Concurrently, the processor can execute another ready-to-run process in its local memory. When the processor is signaled by the mechanism that the lock has been allocated, it immediately resumes execution of the waiting process. Note resumption is immediate because the process was not swapped out. The busy-wait can be avoided if, in the first access to the lock, it is found busy. In this case, the process requesting it is blocked. When the lock becomes available, the mechanism is signaled so that the blocked process can be readied to resume execution. The latter scheme seems adequate for a mutually exclusive access to a resource with large granularity.

The distribution of locks in memory is an important factor in the performance of concurrent processes accessing lockable resources. For example, if all locks are stored in one memory module, the contention for these locks can become excessive. In a multiprocessor with private caches, the accesses to locks by the processors can cause excessive overhead because of consistency checks. However, contention for these locks can be partly relieved by distributing the locks into many blocks of memory.

Two primitive operations can be defined to *block* a process attempting to enter a busy critical section and *wake_up* the blocked process when the critical section becomes free. These primitives are

wake_up (p): if process p is logically blocked (that is, dormant), change its state to active; else set up a *wake-up waiting switch* (wws) to remember the wake-up call

block (p): if process p's wws is set, reset it and continue execution of the process; else change p's state to dormant

Using these operations requires a wake-up list which is updated dynamically. In order to prevent loss of information, wake-up signals that occur while a process is executing must be saved and a process should not be allowed to become dormant until all its wake-up signals have been serviced. The wake-up waiting switch (wws) is the mechanism used to save wake-up signals. A process identification tag is appended to the wake-up signal, which is used to route the signal to the appropriate receiving process. Hardware or software mechanisms may be used to implement the wws, which stores the wake-up signals on arrival until they are acknowledged. This may result in a potential race condition if the mechanism is improperly designed. Note that the blocking and unblocking operations constitute an overhead which may be significant if designed improperly.

Lock conflicts are resolved in the implementation of the busy-wait because a request that finds the lock busy waits until the lock is released. Serialization is thus enforced. Another synchronization primitive was proposed to permit some form of concurrency of access to a memory location while still enforcing some serialization. The format of this primitive is *fetch-and-add* (X, e), where X is a shared integer variable and e is an integer expression. Let the value of X be denoted by Y. We abbreviate the primitive as **F & A**(X, e), which is defined to return the old value of X and replace the contents of X by the sum $Y + e$ in one indivisible operation. If several fetch-and-add operations are initiated simultaneously by different processors on the shared variable X, the effect of these operations is exactly what it would be if they occurred in some unspecified serial order. That is, X is modified by the appropriate total increment and each operation yields the intermediate value of X corresponding to its position in the serial order. As an example, consider the two processors P_i and P_j which issue:

$$S_i \leftarrow \mathbf{F\&A}(X, e_i),\ S_j \leftarrow \mathbf{F\&A}(X, e_j)$$

respectively. Then S_i and S_j may contain Y and $Y + e_i$, respectively, or S_i and S_j may contain $Y + e_j$ and Y, respectively, depending on the priorities of P_i and P_j and the order of arrival of their requests. In either case, X becomes $Y + e_i + e_j$.

The fetch-and-add primitive can be implemented within the processor memory switch, as shown in Figure 8.2 for the example with two simultaneous fetch-and-adds directed at the same memory location X. In the example, it is assumed that P_i's request has a higher priority than P_j's. The switch forms $e_i + e_j$ and transmits $\mathbf{F\&A}(X, e_i + e_j)$ to the memory. At the same time, e_i is stored in the switch's register. On receipt of Y from memory as a result of $\mathbf{F\&A}(X, e_i + e_j)$, the switch transmits Y and $Y + e_i$ in response to requests $\mathbf{F\&A}(X, e_i)$ and $\mathbf{F\&A}(X, e_j)$, respectively.

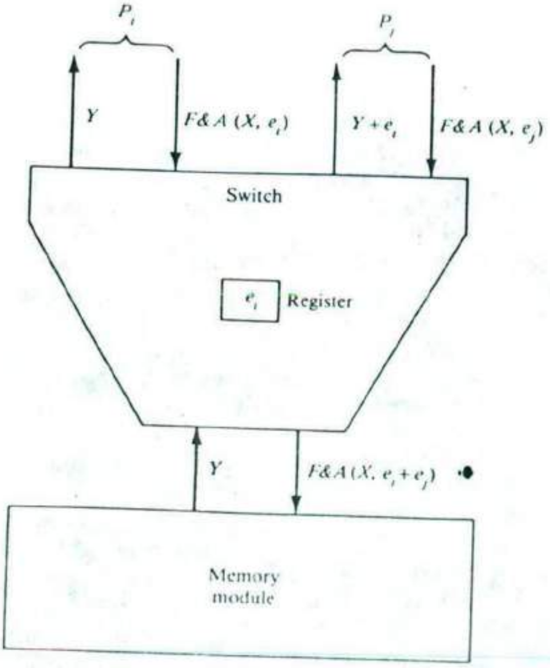


Figure 8.2 Implementation of the *Fetch-and-Add* primitive (F & A).

We illustrate a simple application of **F&A** in a multiprocessor environment. Suppose it is required that all processors which intend to access a given resource must first indicate their intentions by incrementing a common counter, X, in memory. An accurate count of these requests can be maintained if each processor indicates its intention by the statement **F&A**(X, 1). Hence if two or more processors execute the statement simultaneously, X will contain the correct count of requests at the completion of all instructions. Note that the value returned to each processor as a result of execution of **F&A**(X, 1) can be used as its position in the request queue.

Another synchronization primitive uses the semaphore, which consists of a counter, a process routine queue, and the two functions **P** and **V**. This is described in the next subsection. Although simple, semaphores are known to be sufficient solutions to synchronization problems for permanent-resource competitors and temporary-resource producers-consumers. However, semaphores are often very inconvenient in representing communication between processes. For this reason, most operating systems also provide other process-communication mechanisms. Two examples are events and messages.

Event primitives are typically provided by the two functions *wait* and *signal*. A process can wait on an event or a combination of events to be true. When another process signals an event, all processes waiting on that event are placed on the ready queue. Other variations are also possible. One potential problem with events is that a process has the possibility of waiting on an event that either never becomes true or was signaled earlier. A slight variation of waiting on an event, used especially in real-time systems, is waiting on a timing queue administered by the operating system for a specified time period to elapse.

Messages provide an even more flexible and direct method of interprocess communication, especially for producer-consumer relationships. Typical primitives are the functions *send* and *receive*, which allow a string of characters to be passed between processes. Implementation variations are numerous. For example, send may or may not wait for an acknowledgement. Receive usually waits if no message has been sent. The Intel iAPX 432 multiprocessor system uses the send and receive primitives.

8.1.2 Synchronization with Semaphores

Dijkstra invented the two operations **P** and **V**, which can be shared by many processes and which implement the mutual-exclusion mechanism efficiently. The **P** and **V** operations are called primitives and are assumed indivisible. They operate on a special common variable called a *semaphore*, which indicates the number of processes attempting to use the critical section:

```
var s: semaphore
```

Then the primitive **P**(s) acts as an open bracket or MUTEXBEGIN of a critical section; that is, it acts to acquire permission to enter. The **V**(s) primitive is the MUTEXEND and records the termination of a critical section:

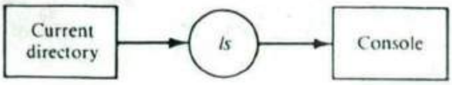
```
P(s): MUTEXBEGIN (s)
        s ← s − 1;
        If s < 0 then
        begin
        Block the process executing the P(s) and put it
        in a FIFO queue associated with the semaphore s;
        Resume the highest priority ready-to-run process;
        end
        MUTEXEND
V(s): MUTEXBEGIN (s)
        s ← s + 1;
        If s ≤ 0 then
        begin
        If an inactive process associated with semaphore s exists, then
        wake up the highest priority blocked process associated with s
        and put it in a ready list.
        end
        MUTEXEND
```

The semaphore s is usually initialized to 1. When s can take values of 0 or 1, it is called a *binary semaphore*, since it acts as a lock bit, allowing only one process at a time within an associated critical section. If s takes any integer value, it is called a *counting semaphore*. Notice that the **P**(s) and **V**(s) operations are modifying s and testing its status. **P**(s) and **V**(s) can be implemented in hardware or in software using locks.

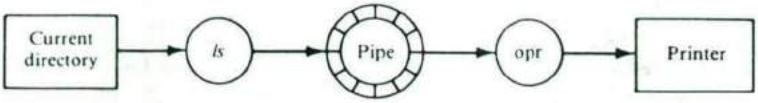
One common use of synchronization mechanisms is to permit concurrent processes to exchange data during execution. The data or messages to be exchanged are usually stored in a circular buffer which is used to synchronize the speeds of the sending and receiving processes. Such a circular buffer is usually called a message buffer or *mailbox*.

For example, the Unix operating system provides an elegant form of message buffers called *pipes*. These are used as channels to stream data from one process to another. The typing of the "ls" command on the console in a Unix environment causes the files in the current directory to be "listed" on the console by running the "ls" process. If the user wishes to print the listing of the files on the printer, the two concurrent processes "ls" and "opr" may be used and are specified as "ls|opr." The "|" symbol specifies that a pipe should channel the output of "ls" to become the input to "opr." The "ls" process *produces* the list as an output into a pipe or buffer from which it is *consumed* and printed by the "opr" process, as illustrated in Figure 8.3.

Whenever a process produces sequences of output which are consumed by another process as input, there is said to be a producer-consumer relationship. A



(*a*) Command: *ls*



(*b*) Command: *ls*|opr

Figure 8.3 Flow of data between two processes in UNIX.

message buffer may be considered to consist of a finite number of identical slots which are used for communication between the producer and consumer processes. If the number of slots is finite, the buffer is arranged as a circular buffer.

To demonstrate the communication between the producer and consumer processes, consider a finite buffer BUFFER of size n arranged as a circular queue in which the slot positions are named $0, 1, \ldots, n-1$. There are the two pointers c and p, which correspond to the "head" and "tail" of a circular queue, respectively, as shown in Figure 8.4. The consumer consumes the message from the head c by updating c and then retrieving the message. Hence, c points to an empty slot before each consumption. The producer adds a message to the buffer by updating p before the add operation. Therefore, pointers p and c move counterclockwise and there can be a maximum of n message slots for consumption. Initially, $p = c = 0$, which indicates that the buffer is empty. Let the variables *empty* and *full* be used to indicate the number of empty slots and occupied slots, respectively. The *empty* variable is used to inform the producer of the number of available slots, while the

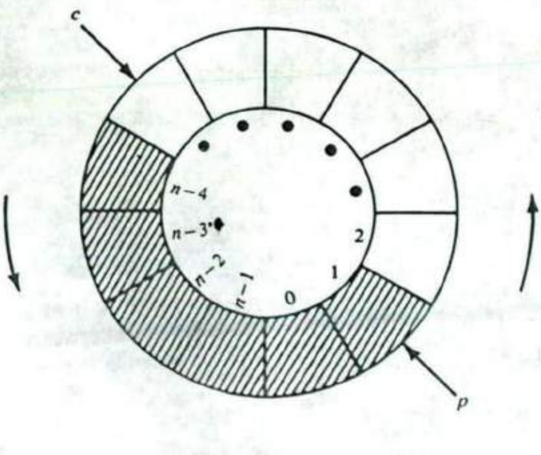


Figure 8.4 A circular message buffer with producer pointer p and consumer pointer c.

full variable informs the consumer of the number of messages needed to be consumed. The concurrent program below illustrates the actions of the producer and consumer processes. The producer or consumer will be suspended when *empty* = 0 or *full* = 0, respectively.

Example 8.1

```
      shared record
             begin
             var p, c: integer;
             var empty, full: semaphore;
             var BUFFER [0:n - 1]: message;
              nd
    initial empty = n, full = 0, p = 0, c = 0;
   cobegin
   Producer: begin
             var m: message;
               Cycle
                  begin
                    Produce a message m;
                    P(empty);
                    p ← (p + 1) mod n;
                    BUFFER [p] ← m; // place message in buffer//
                    V(full)
                  end
            end
  Consumer: begin
             var m: message;
               Cycle
                  begin
                    P (full);
                    c ← (c + 1) mod n;
                    m ← BUFFER [c]; // remove message from buffer //
                    V (empty);
                    Consume message m;
                  end
               end
  coend
```

The P and V operations may be extended for ease of problem formulation and clarity of solutions. The extended primitives **PE** and **VE** developed by Agerwala (1977) are indivisible and each operates on a set of semaphores which must be initialized to nonnegative values.

```
PE($s_1$,$s_2$, ... ,$s_n$ $\bar{s}_{n+1}$, ... ,$\bar{s}_{n+m}$):
  MUTEXBEGIN
if for all i, $1 \leq i \leq n$, $s_i > 0$ and for all j, $1 \leq j \leq m$, $s_{n+j} = 0$
    then for all i, $1 \leq i \leq n$, $s_i \leftarrow s_i - 1$
      else the process is blocked and put in a set of queues associated
           with the set of semaphores $s_1, \ldots, s_n$;
  MUTEXEND
VE($s_1$,$s_2$, ... , $s_n$):
  MUTEXBEGIN
for all i, $1 \leq i \leq n$, $s_i \leftarrow s_i + 1$;
    wake-up highest priority process
    associated with set of semaphores ($s_1, \ldots, s_n$);
  MUTEXEND
```

There is no association between s_i and $\bar{s}_j$. The $\bar{s}_j$ symbol is used for convenience to represent the semaphore s_j where $j > n$. The following examples are used to illustrate the application of the extended primitives.

Example 8.2: *N* processes, equal priority, *m* resources Each of N processes requires exclusive access to a subset of m distinct resources. The processes are granted access without any consideration of priorities. If two processes use disjoint subsets of resources, they may execute simultaneously. The solution is given below:

```
           var $r_1$,$r_2$, ... ,$r_m$: semaphore
          initial $r_1 = r_2 = \cdots = r_m = 1$;
Process i: begin
             PE ($r_a$,$r_b$, ... ,$r_x$);
             Use resource a, b, ... , x;
             VE ($r_a$,$r_b$, ... ,$r_x$)
           end
```

Semaphore r_i is associated with resource i. If the PE primitive is completed successfully by a process, it indicates that resources $a, b, \ldots, x$ are available and hence are allocated to the process.

Consider the application of this example in which processes X, Y, Z compete for card reader R, printer P, and tape unit T, as shown in Figure 8.5. Each process requires two of the resources simultaneously: X requires R and P, Y requires R and T, and Z requires P and T. The "Dining Philosophers' Problem" can be expressed as a special case of the above example (see Problems 8.6 and 8.7).

Example 8.3: *N* processes, *N* priorities, one resource Process i has higher priority than process $i + 1$, for $1 \leq i \leq N - 1$. The processes request access to a resource and are allocated the resource in a mutually exclusive manner

```
            var $r_R$, $r_P$, $r_T$: semaphore;
            initial $r_R = r_P = r_T = 1$;
            cobegin
Process X: begin PE($r_R$, $r_P$); use resource R and P; VE($r_R$, $r_P$); end
Process Y: begin PE($r_R$, $r_T$); use resource R and T; VE($r_R$, $r_T$); end
Process Z: begin PE($r_P$, $r_T$); use resource P and T; VE($r_P$, $r_T$); end
            coend
```

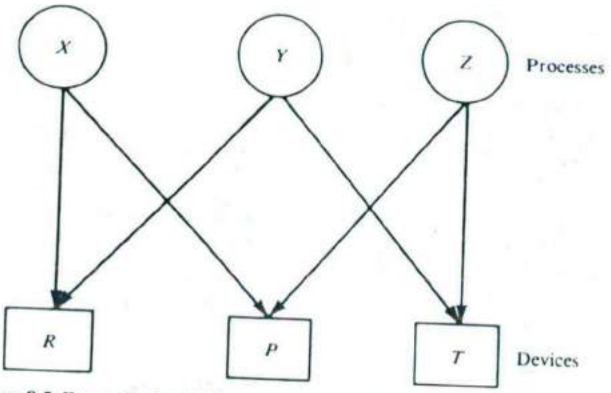


Figure 8.5 Example of multiple resource allocation.

based on the priorities. A request by a process is not honored until all higher priority requests have been granted. The resource is used nonpreemptively:

```
        var $s_1, \ldots, s_n$, R: semaphore
        initial $s_1 = s_2 = \cdots = s_n = 0$;
        initial R = 1;
Process i; begin
            VE ($s_i$); // register request of process i //
            PE (R,$\bar{s}_1, \ldots, \bar{s}_{i-1}$); // check to see if resource is available //
                  // and if there are any outstanding requests //
                  // made by higher priority processes //
            PE ($s_i$); // if not, grant resource to process i //
                  // and withdraw outstanding request //
            Use resource;
            VE (R); // Return or deallocate resource //
            end
```

Note that the requesting process cannot be blocked on **PE**(s_i) since a **VE**(s_i) was executed earlier to register request. This example may be used in servicing prioritized interrupts. In this case the processes represent the interrupts and R represents the processor which services the interrupts.

The above example can be clarified further by considering a two-processor system that runs a supervisor process and a user process. If these two processes compete for a certain resource in the system *simultaneously*, the supervisor process should be given higher priority. The program segments of the supervisor and user processes that access the shared resource are shown below:

```
var    s, u, R: semaphore
initial s = u = 0;
initial R = 1;
Supervisor: begin                 User: begin
              VE(s);                      VE(u);
              PE(R);                      PE(R, s);
              PE(s);                      PE (u);
              Use Resource;               Use Resource;
              VE(R);                      VE(R);
              end                         end
```

Notice that the constructs differ mainly in the second **PE** statements. Since the supervisor process is of a higher priority than the user process, it only checks to see if the resource is available [**PE**(R)], whereas the user process also checks to see if there is an outstanding request from the supervisor [**PE**(R, s)]. Since we are considering simultaneous execution of the user and supervisor codes, the execution of the **PE**(R, s) statement will find $s = 1$, which was set by the **VE**(s) operation in the supervisor process. Hence, the user process will be blocked until the resource is released by the supervisor.

Although, semaphores can be implemented using locks, they are more commonly accessed by system calls to the supervisor. The supervisor maintains two sets of lists or queues: *blocked* and *ready*. Descriptors for processes that are blocked on a semaphore are added to a block queue associated with that semaphore. For the generalized **P** and **V**, the set of blocked queues may be quite complex. However, execution of a **PE** or **VE** operation causes a trap to a supervisor routine which completes the operation. The ready list contains descriptors of processes that are ready to be assigned to a processor for execution. In a multiprocessor, with master-slave operating system, a single processor may be responsible for maintaining the ready list and assigning processes to the slave processors. The ready list may be shared in a multiprocessor with a distributed supervisor. In this case, the ready list may be accessed concurrently. Therefore, mutual exclusion must be ensured and can be accomplished by spin-locks, since enqueue and dequeue operations are fast on the ready list. Moreover, a processor that is attempting to access the ready list cannot execute any other process.

Semaphores are quite general and can be used to program almost any kind of synchronization. However, the use of the **P** and **V** primitives in a parallel algorithm makes the algorithm rather unstructured and prone to error. For example, omitting a **P** or **V**, or accidentally invoking a **P** on one semaphore and a **V** on another can have disastrous effects, since mutual exclusion would no longer be ensured. Also, when using semaphores, a programmer can forget to include in critical sections all statements that reference the shared modifiable objects. This, too, could cause errors in execution. Another problem with using semaphores is that both condition synchronization and mutual exclusion are implemented using the same pair of primitives. This makes it difficult to identify the purpose of a given **P** or **V** operation without a detailed trace of other effects on the semaphore.

8.1.3 Conditional Critical Sections and Monitors

Conditional critical section (CCS) was proposed by Hoare (1972) and Hansen (1972) to overcome most of the difficulties encountered with **P** and **V**s. This is a structured and highly user-oriented tool for specifying communication among concurrent processes. Their use allows direct expression of the fact that a process has to wait until an arbitrary condition on the shared variables holds. Interprocess communication in a system of concurrent processes is done by means of a shared variable v, which is composed of the component variables $v_1, v_2, \ldots, v_n$, as defined by:

var v : **shared record** $v_1, \ldots, v_n$: ⟨type⟩ **end**

The variable v is used to name a given resource. The global state of a system of processes is determined by the values of the shared variable v and the program counters of the single processes. The variables in v may only be accessed within CCS statements that name v. A CCS statement is of the form

csect v **do await** C:S

where C is a boolean expression and S is a statement list. Note that variables local to the executing process may also appear in the CCS statement:

A CCS statement delays the executing process until the condition, C, is true; S is then executed. The evaluation of C and execution of S are uninterruptible by other CCS statements that name the same resource. Thus, C is guaranteed to be true when execution of S begins. Mutual exclusion is provided by guaranteeing that execution of different CCS statements, each naming the same resource but not overlapped. Condition synchronization is provided by explicit boolean conditions in CCS statements.

We illustrate the use of the conditional critical sections by two applications. The first example is a solution to the producer-consumer problem. Assume that the two classes of processes (producers and consumers) communicate via a bounded circular buffer as in Figure 8.4. Access to this buffer must be mutually exclusive. Seven shared variables which are associated with the critical section v are used to indicate the global status of the system of processes.

Example 8.4 The variables *p* and *c* are as in Example 8.1. Variables *empty* and *full* are also integer variables denoting the number of slots empty or occupied respectively. Variables *np* and *nc* indicate the number of producers and consumers respectively, which are working on the buffer.

```
var v: shared record
            begin
            var p, c, empty, full, np, nc: integer;
            var BUFFER [0: n − 1]: message;
            end
initial empty = n, full = 0, p = 0, c = 0;
Procedure Enqueue (m: message)
  begin
    csect v do await empty > 0 and np = 0:
        begin
          np ← np + 1;
          empty ← empty − 1;
        end
    p ← (p + 1) mod n;
    BUFFER [p] ← m;
    csect v do full ← full + 1;
  end
Procedure Dequeue (m: message)
  begin
    csect v do await full > 0 and nc = 0:
        begin
          nc ← nc + 1;
          full ← full − 1;
        end
    c ← (c + 1) mod n;
    m ← BUFFER [c];
    csect v do empty ← empty + 1;
end
```

The second example on the use of the conditional critical section is the solution of the reader-and-writer problem. Improper reading and writing of shared variables is the classic cause of difficulty in finding operating system bugs. The basic problem is that two sets of processes executing concurrently may interleave read and write operations in such a way that improper decisions are made and the shared variables are left in an improper state. This kind of bug is insidious, for it may only show up infrequently—and then the symptoms occur rarely or never repeat since they depend on a particular concurrency relationship.

In the reader-and-writer problem, there are reader and writer processes which share a common data segment. Any number of readers may access the segment simultaneously, but a writer must have exclusive access to it. To prevent a writer from waiting indefinitely long, it is necessary that no more readers be able to acquire the resource from the moment that a writer first wants to acquire it until the time it actually does acquire it. The variable *aw* indicates the number of writers that want to acquire the resource; *nw* and *nr* indicate the number of writers and readers, respectively, that have acquired the resource.

Example 8.5

```
        var v shared record aw, nw, nr   : integer end
        Initial aw = nw = nr = 0;
    Reader: begin
            csect v do await aw = 0: nr ← nr + 1;
            read segment;
            csect v do nr ← nr − 1;
            end
    Writer: begin
            csect v do
              begin
              aw ← aw + 1;
              await nr = 0 and nw = 0: nw ← nw + 1;
              end
              write to segment;
              csect v do begin
                nw ← nw − 1;
                aw ← aw − 1;
                end
              end
            end
```

All the process-synchronization methods we have discussed are logically equivalent in performing the synchronization or scheduling problem. However, some of them implement the solution to certain problems in a more complicated and inefficient manner than others. Therefore, they are not practically equivalent.

Monitors—*extension of conditional critical sections* A monitor is a shared data structure and a set of functions that access the data structure to control the synchronization of concurrent processes. This general definition includes semaphores, events, and messages as specific implementations. The notion of a monitor is not more powerful than these other techniques—just more general. While a process is a useful abstraction for multiprogramming, a monitor is a useful abstraction for process communication. Consequently, a programmer can ignore the implementation details of the resource when using it and can ignore how it is used when programming the monitor that implements it.

To assure the correctness of a program, it is useful to associate data structures with the operations performed on them. A monitor provides a body in which to

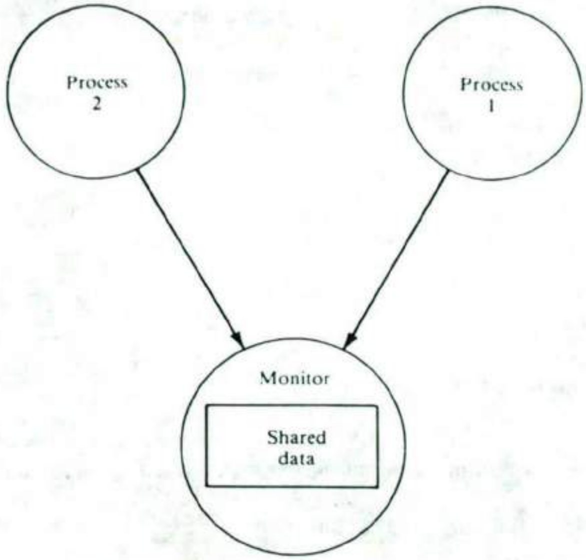


Figure 8.6 Monitor representation.

associate shared data structures with their critical sections. By so doing, the data structures are no longer shared or global, but local or hidden within the body of a monitor. In addition, process functions no longer contain critical sections. Instead, the critical sections are centralized and protected within the monitor functions. The restricted access to shared data structures provided by a monitor is even more attractive if it can be checked by a compiler. Many high-level languages today provide the means for controlling the scope of variable names.

Monitors provide support for processes to form a multiprogramming system. While a process is active in the sense that it performs a job, a monitor is passive in the sense that it only executes when called by a process. A monitor is necessary only when two or more processes communicate to ensure that they communicate properly. Figure 8.6 is a representation of two processes communicating through shared data encapsulated by a monitor.

A monitor consists of a set of *permanent variables* used to store the resource's state, and some procedures, which implement operations on the resource. A monitor also has initialization code for the permanent variables. This code is executed once before any procedure body is executed. The values of the permanent variables are retained between activations of monitor procedures and may be accessed only from within the monitor. Monitor procedures can have parameters and local variables, each of which takes on new values for each procedure activation. The structure of a monitor with name *mname* and procedures OP1, . . . , OPN is shown below.

```
mname monitor;
  var declarations of permanent variables
  procedure OP1 (parameters)
    var declarations of variables local to OP1
    begin
      code to implement OP1
    end
      ⋮
    procedure OPN (parameters)
    var declarations of variables local to OPN
    begin
      code to implement OPN
    end
  begin
    code to initialize permanent variables
  end
```

The procedure OPJ within monitor *mname* can be invoked by executing

call *mname* · OPJ (arguments).

The execution of the procedures in a given monitor is guaranteed to be mutually exclusive. This ensures that the permanent variables are never accessed concurrently.

Pushing the monitor concept to its logical limit suggests that systems should be designed as collections of processes and monitors only. In this case, every data structure is local to either a process or monitor. This decomposition is valuable in large systems since it simplifies the problems of program validation and maintenance. If a data structure changes, it is clear which functions are affected, and the addition of a new process or monitor does not require the revalidation of unchanged components.

The current state of a monitor is defined by the monitor image, that is the memory associated with the monitor program. A monitor image represents either permanent or temporary resources that are the elements of process interaction. A process image is that portion of memory belonging to a process and defining its states. The process image changes with the execution of a program associated with the process. In the absence of process activity, process images and monitor images differ significantly. In this idle state, process images are of no importance and may vanish. However, monitor images—at least those representing permanent resources—must remain and resume a nonassigned state.

Monitor functions are reentrant but contain nonreentrant sections—i.e., critical sections. Indeed, monitor functions must be designed to protect against this. Monitor functions need not be considered as part of the monitor image. In fact, if two different monitor variables are accessed in the same way, a single copy of a program may be shared between the two monitors.

When a monitor function is called but is blocked from handling the request immediately, it may take several actions. It may immediately return a blocked indication, it may loop or busy-wait until the request can be handled, or it may place the process on a waiting queue for the resource requested. In the latter case, the waiting queue must be a part of the monitor data structure. In real-time systems, it is sometimes best to return a blocked indication and let the process decide whether to try again later or give up.

An operating system contains a kernel or nucleus which contains a few special processes to handle initialization and interrupts and a basic monitor to support the concept of a process. The basic monitor includes functions to switch environments between processes and to create, spawn, or fork a new process. The kernel is also one part of an operating system that executes in the privileged state.

Besides the kernel, an operating system consists of many monitors and a few processes. The processes include several kinds of I/O processes that are activated as needed and at least one active process to look for new jobs and create user processes for them. All monitors are part of the operating system and form the bulk of the system. They are used to manage the resources of the system. For example, monitors transmit messages between processes, control competing processes, enforce access rights, and communicate with I/O processes.

8.2 SYSTEM DEADLOCKS AND PROTECTION

With a high degree of concurrency in multiprocessors, deadlocks will arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group. We shall present some effective techniques for detecting, preventing, avoiding, and recovering from deadlocks. Also, we will discuss various protection mechanisms which can be used in a multiple process environment to ensure only authorized access to resources.

8.2.1 System Deadlock Problems

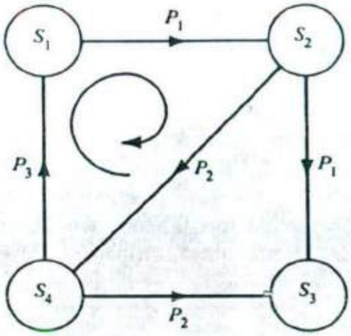
We use the following example to explain the cause of a system deadlock and the means to break a deadlock.

Example 8.6 Consider the three concurrent processes P_1, P_2, and P_3 sharing four distinct resources controlled by the four semaphores S_1, S_2, S_3, and S_4. All semaphores have an initial value 1. P-V primitives are used to specify the resource-request patterns shown in Figure 8.7*a*. Assume one unit of each resource type.

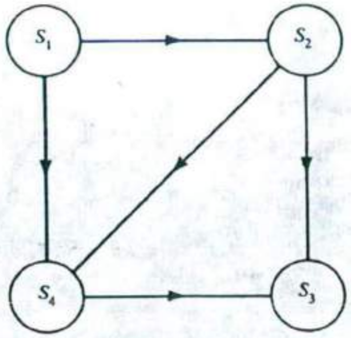
We use a directed graph in Figure 8.7*b* to show the possible resource-allocation ordering. The nodes correspond to resource semaphores, one per type. An edge (labeled P_k) from S_i to S_j means that resource S_i has been allocated to process P_k and P_k is requesting resource S_j. Following this rule, we

P_1	P_2	P_3
⋮	⋮	⋮
$P(S_1)$	$P(S_2)$	$P(S_4)$
⋮	⋮	⋮
$P(S_2)$	$P(S_4)$	$P(S_1)$
⋮	⋮	⋮
$P(S_3)$	$P(S_3)$	
⋮	⋮	
$V(S_1)$	$V(S_4)$	$V(S_1)$
$V(S_2)$	$V(S_2)$	$V(S_4)$
$V(S_3)$	$V(S_3)$	

(*a*) Three concurrent processes



(*b*) Resource allocation graph corresponding to part (*a*)



(*c*) Resource allocation graph after changing the order of $P(S_4)$ and $P(S_1)$ in part (*a*)

Figure 8.7 Concurrent processes for the deadlock study in Example 8.5.

obtain the *resource-allocation graph* in Figure 8.7*b*. The existence of a loop in the graph shows the possibility of a deadlock.

For example, the three nodes S_1, S_2, and S_4 form a loop in the clockwise direction. This means a deadlock situation in which resources S_1, S_2, and S_4 have been allocated to processes P_1, P_2, and P_3, respectively. P_1 is waiting for P_2 to release S_2, while P_2 is waiting for P_3 to release S_4, and P_3 is waiting for P_1 to release S_1. These three processes thus enter a *circular wait* situation, so that no process among the three can proceed. This situation is called *system deadlock* or "deadly embrace".

Suppose we modify the request pattern in process P_3 by exchanging the order of requests $P(S_4)$ and $P(S_1)$. A new resource-allocation graph results in Figure 8.7*c*, where the edge direction has been reversed from node S_1 to S_4. Since there are no loops in this modified graph, no circular wait will be possible. Thus deadlock can be avoided. Because of data dependencies or other reasons, the change of request order may not be permitted in general. Thus better techniques are needed to avoid deadlock. We shall show some of these techniques later.

In general, a deadlock situation may occur if one or more of the following conditions are in effect:

1. *Mutual exclusion*: Each task claims exclusive control of the resources allocated to it.
2. *Nonpreemption*: A task cannot release the resources it holds until they are used to completion.
3. *Wait for*: Tasks hold resources already allocated to them while waiting for additional resources.
4. *Circular wait*: A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain.

The existence of these conditions effectively defines a state of deadlock in the previous example. In many ways, the first three conditions are quite desirable. For consistency, data records should be held until an update is complete. Similarly, preemption (the reclaiming of a resource by the system) cannot be done arbitrarily and must be supported by a *rollback* recovery mechanism, especially when data resources are involved. Rollback restores a process and its resources to a suitable previous state from which the process can eventually repeat its transactions.

Solutions to the deadlock problem have been classified as *prevention*, *avoidance*, *detection*, and *recovery* techniques. Prevention is the process of constraining system users so that requests leading to a deadlock never occur. For deadlock prevention, the system is designed so that one or more of the necessary conditions outlined above never hold. The scheduler then allocates resources so that the deadlocks will never occur. For deadlock avoidance, the scheduler controls resource allocation on the basis of advance information about resource usage so that deadlock is avoided. With deadlock detection and recovery, the scheduler gives resources to the process as soon as they become available and, when a deadlock is detected, the scheduler preempts some resources in order to recover the system from the deadlock situation.

Empirical observations have suggested that deadlock prevention mechanisms tend to undercommit resources while detection techniques give away resources so freely that prolonged blocking situations arise frequently. Avoidance schemes fall somewhere in between. Avoidance or prevention mechanisms must ensure that a deadlock will never occur for every request, resulting in undue process waits and run-time overhead. A prevention mechanism differs from an avoidance scheme in that the system need not perform run-time testing of potential allocations. In both prevention and avoidance cases, recovery from a system implementation error needs a rollback mechanism.

8.2.2 Deadlock Prevention and Avoidance

By restricting the behavior of the processes so that one of the necessary conditions for the occurrence of a deadlock is violated, deadlocks will never occur. This approach has been called *static prevention*, since the rule for allocating resources does not depend upon the current state of the system. One method is to relax the mutual-exclusion condition by permitting simultaneous access to resources by processes. This, for example, is possible in a read-only data object. However, in general the physical or logical properties of the resources may not always permit simultaneous access. Examples of these resources are card readers, printers, and critical sections of code. Hence, prevention of deadlock permitting simultaneous access to resources is impractical.

Similarly, the denial of the nonpreemption condition in a general case is unrealistic since physical resources such as line printers cannot be preempted. However, the nonpreemption condition can be waived for processors and memory pages which may be time-multiplexed among several processes. The forcible deallocation of a resource and the allocation of the resource to the preempting process may cause an intolerable overhead which, if not carefully controlled, may result in an inefficient utilization of resources. Another approach to preventing deadlock is to constrain each task to completely allocate all its required resources in advance of its execution. This limits the degree of concurrency and also under-utilizes the resources. This technique derives from the wait-for condition.

Finally, by constraining the resource types into a linear ordering, circular waiting can be prevented. In this method, the resource types are ordered into a resource hierarchy so that, for a system with m resource types, $R_1 < R_2 < \cdots < R_m$. If a task has been allocated a resource of type R_i, it can only request a resource R_j which is at a higher level so that $R_i < R_j$. The rule for releasing resources is that resource R_j must be released before R_i, if $R_i < R_j$. By this means, circular chains of blocked tasks cannot occur since each task requests resources in the same orderly way. A deadlock cannot occur in a system with linearly ordered resources. The feasibility of enforcing resource ordering by compile-time checks is a major advantage of this scheme. Restrictions on the allowable sequences of task requests force knowledgable use of the ordering rule. This technique is used in the IBM 370 series MVS operating system.

To avoid deadlocks in a multiprogrammed multiprocessing system in which the necessary conditions for deadlocks can exist, it is usually necessary to have some advance knowledge of the resource usage of processes. Sometimes deadlock avoidance techniques are called *dynamic prevention* methods since they attempt to allocate resources depending upon the current state of the system. These methods lead to better resource utilization.

One basic model that is assumed consists of a sequence of task steps during each of which the resource usage of the task remains constant. The execution of a task step first involves the acquisition of those resources needed by the given task step but not passed on by the previous task step. Next follows a period of execution during which the resource requirement remains invariant. Finally, at the completion of execution, all those resources not needed by the subsequent task step are released and returned to a pool of available resources.

Before discussing the avoidance techniques, it is convenient to describe another model for the resource-allocation system in a multiprocessor. A *resource-allocation system* (RAS) includes a set of n independent processes $P_1, P_2, \ldots, P_n$ $(n \geq 1)$, a set of m different types of resources $R_1, R_2, \ldots, R_m$ $(m \geq 1)$ so that each R_i has a fixed number of units c_i. The RAS also includes a scheduler that allocates the resources to the processes according to certain rules fulfilling some specified criteria.

The system state of a RAS is defined by $(\mathbf{W}, \mathbf{A}, \mathbf{f})$, where $\mathbf{W} = (\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n)$ is the *request matrix*, which has the dimensions n by m. The entry $\mathbf{w}_{ij} = \mathbf{w}_i(j)$ is the maximum number of additional units of resource R_j that the process P_i will need at one time to complete its task. $\mathbf{w}_i$ is the *want vector* for process $P_i \cdot \mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$ is the *allocation matrix* $(n \times m)$. The entry $\mathbf{A}_{ij} = \mathbf{a}_i(j)$ is the number of units of resource R_j allocated to process P_i. $\mathbf{a}_i$ is the *allocation vector* for process P_i. The vector $\mathbf{f} = (f_1, f_2, \ldots, f_m)$ is the *free* (available) *resource vector* and $\mathbf{c} = (c_1, c_2, \ldots, c_m)$ is the *system capacity vector*. Since $f_j \leq c_j$ is the number of available units of resource type R_j, we can find f_j to be

$$f_j = c_j - \sum_{i=1}^{n} \mathbf{a}_i(j) \tag{8.1}$$

That is, the sum of the resources allocated and those available of type R_j must be equal to the total number of units of that type in the system.

When $\mathbf{A} = 0$, the system is in the initial state. In this state, $\mathbf{D} = (\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n) = W$ is called the *demand matrix* and $\mathbf{c} = \mathbf{f}$, where $\mathbf{d}_i$ is the demand vector for process P_i.

Certain basic assumptions are made regarding avoidance methods. Before a process enters the system, it is required to specify for each resource the maximum number of resource units it will ever need. There is no preemption and a process releases a resource after it has completed its task. Moreover, $\mathbf{d}_i \leq \mathbf{c}$ for all i.

A sequence of task steps $P_{e(1)}P_{e(2)} \cdots P_{e(k)}$ is called a *terminating sequence* for $(\mathbf{W}, \mathbf{A}, \mathbf{f})$, where $e(j)$ is the index of the process in the jth place, if

$$\mathbf{w}_{e(1)} \leq \mathbf{f}$$

and (8.2)

$$\mathbf{w}_{e(i)} \leq \mathbf{f} + \sum_{j=1}^{i-1} \mathbf{a}_{e(j)} \qquad \text{for } 1 < i \leq k$$

Note that each occurrence of a process in the sequence goes through the following cycle: request resource, use resource, release resource. Hence the want vector for a process must not be greater than the free resource vector plus the "released" resource vector for the process to be run. A terminating sequence is *complete* if all processes are in the sequence.

The system state (**W**, **A**, **f**) is *safe* if there is a complete terminating sequence for it; that is, if there is a way to allocate the resources claimed by the processes so that all of them can finish their task. The safeness of a state can be expressed as a *safe request matrix* **S**, where S_{ij} is the maximum number of units of resource R_j that can be granted safely if process P_i requests them.

Example 8.7 If we restrict each process to making a single request for a finite number of units and the state of the system (**W**,**A**,**f**) for three processes and two resource types is:

$$\mathbf{W} = \begin{bmatrix} 1 & 0 \\ 2 & 2 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{f} = (1, 1)$$

then the system is in a safe state. Notice that from Eq. 8.1, $\mathbf{c} = (2, 2)$ and $P_1 P_3 P_2$ is a complete terminating sequence. If P_2 requests only one unit of R_1, the system will be in a safe state, since in that case $P_3 P_1 P_2$ is a terminating sequence containing P_2. Therefore, $S_{2,1} = 1$. Similarly, if P_2 requests only one unit of R_2, $S_{2,2} = 1$. However, if P_2 requires at the same time one unit of R_1 and one unit of R_2, the request cannot be granted safely.

This example shows that the matrix **S** can be used only for single requests. If a process requires more than one resource, a rule stating the order in which the resources will be requested must be defined. In general, the computation of **S** may be time consuming. However, it is possible to compute **S** concurrently with process execution in a multiprocessor system. This reduces the overhead significantly.

8.2.3 Deadlock Detection and Recovery

The deadlock detection algorithm uses the information contained in a state of the system to decide whether or not a deadlock exists. For the case of a system with only one unit of each resource type, it is sufficient for the detection mechanism to find a circuit (circular wait) in the directed graph. Thus the deadlock detection mechanism maintains and examines a directed state graph to determine whether a circuit exists each time a resource is requested, acquired, or released by a task.

We can extend this technique to systems that have more than one unit of each resource type. Since a circuit in the state graph is only a necessary condition for a deadlock, a more elaborate state-detection mechanism is required. Using the notation developed previously, we recall that the free-resource vector **f** is such that its component f_j satisfies Eq. 8.1. Let **0** indicate a (row) zero vector, in which all components are zero. Also, $\mathbf{x} \leq \mathbf{y}$, where **x** and **y** are vectors, holds if and only if each pair of corresponding elements is related by ≤. The algorithm presented below is designed to reveal a deadlock by simply accounting for all possibilities of sequencing the tasks that remain to be completed. Suppose at time t the state of the system is given by (**W**, **A**, **f**), where **W** is the want matrix, **A** the allocation matrix, and **f** the free-resource matrix. The following algorithm determines the existence of a deadlock at time t:

```
Procedure DETECT DEADLOCK
begin
Initialize Rows ← {1,2,. . . ,n};
while i ∈ ROWS such that $\mathbf{w}_i \le \mathbf{v}$ do
   begin
      ROWS ← ROWS − {i};
      $\mathbf{v} \leftarrow \mathbf{v} - \mathbf{a}_i$;
   end
If ROWS ≠ $\phi$ then DEADLOCK;
end
```

The "while" statement searches for an index i in ROWS such that $\mathbf{a}_i \le \mathbf{v}$. If none is found, a nonempty set in ROWS indicates the existence of a deadlock; otherwise, there is no deadlock. An important point to remember is that a program bug can cause a deadlock even in situations where deadlocks theoretically cannot occur. An ultimate time out can be a simple defensive check on the correctness of the system as well as a way to prevent indefinite deadlocks.

Given that a deadlock has occurred, perhaps the simplest approach to recovering from it would involve aborting each of the deadlocked tasks or, less drastically, aborting them in some sequence until sufficient resources are released to remove the deadlocks in the set of remaining tasks. Algorithms could also be designed that search for a minimum-sized set of tasks which, if aborted, would remove the deadlocks. A more general technique involves the assignment of a fixed cost b_i to the removal (forced preemption) of a resource of type R_i from a deadlocked task that is being aborted.

8.2.4 Protection Schemes

Protection is a mechanism which checks whether the concurrent processes are not trying, in case of error or malicious action, to exceed their rights in accessing the set of objects in the system. Protection should be distinguished from *security*, which is a policy used to denote mechanisms and techniques that control who may use or modify the computer system or the information stored in it. The protection mechanism protects one process from another. This includes the protection of the supervisor from the users and vice versa. The inclusion of an efficient protection scheme in a system also prevents the wide propagation of errors. The techniques of error confinement are based on hardware and firmware mechanisms of access control to entities or objects of the system. These mechanisms do not prevent occurrences of errors but are intended to limit their incidence on the protected objects. An addressing mechanism that decomposes the space of objects into *protected domains* is one scheme that makes the confinement and the non-propagation of errors easier. A *domain* is the set of objects that may be directly accessed by the process to which authorization is granted.

One common set of objects that is protected in a computer system is the set of memory locations. It is imperative to protect the address space of one task from the other by enforcing a separation of address space among different tasks. The protection of shared memory is classified into the protection of the physical and local address spaces.

The protection of physical address space could be accomplished by partitioning the physical memory into nonoverlapping blocks (page frames), each of which is assigned a lock. Each process has a key (often called an *access key*) as part of its process-identification word. Access to a block of memory by a process is granted if the process' access key fits the lock of the block. In fact, any process with a matching key can have access to the block. Hence, this scheme is not an effective protection mechanism if the blocks are shared by several processes.

Memory protection on the virtual address space is more effective. In systems that use base or relocation registers for mapping the virtual address to the physical address, protection is accomplished with *bounds registers*. Bounds registers specify the lower (base address) and upper bounds of the address space. An access by a process to memory outside the predefined bounds is trapped by the system as an access violation. This technique assumes that the address space for the process is in contiguous memory locations. The protection of the address space is further ensured since systems do not permit the modification of the bounds register by a user process. Sharing of address space by more than two concurrent processes is still difficult since the address space of each process is contiguous and the bounds registers perform linear mapping of addresses.

Another method for memory protection is for each task to have a segment table (ST), which consists of entries that include the set of access rights and the base address to pages or segments of the task. During the execution of a task, it uses the segment table base register (STBR) to obtain access by authenticating the privileges of the processes, as shown in Figure 8.8. If the access-code field (AKR) of the virtual address matches the permitted access rights for the segment (acc) in STE(s), the segment is accessed. When a task switch occurs, the STBR is modified to point to the ST of the new task.

In process systems, there is a classic distinction between user privileges and supervisor privileges. The protection of supervisor processes from user processes is enforced partly by a hardware mechanism which defines two operating modes: supervisor mode and user mode. This can be accomplished by a single bit in the processor's control register. Generally, there are machine instructions, called *privileged instructions*, which are not executable by user processes. These instructions can be considered the set of objects S which needs to be protected from the user that attempts execute access. The user domain in this case lacks all rights in the set $R = \{r | r = (s, \text{execute}), s \in S\}$. The protection is enforced by trapping the

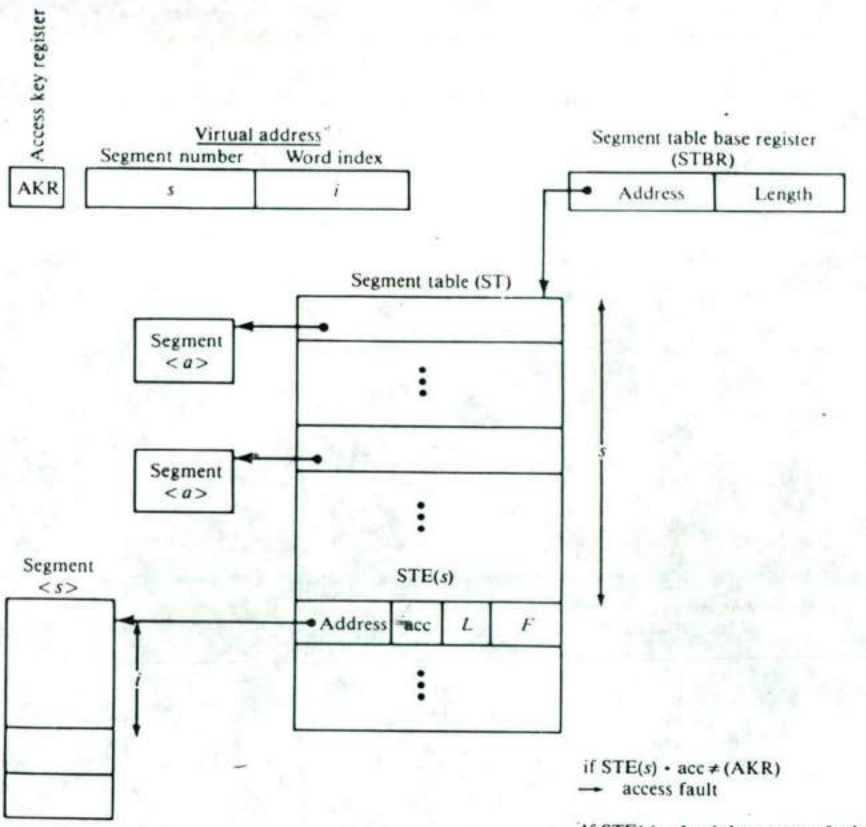


Figure 8.8 Address mapping in a segmented system.

user process that attempts to use $r \in R$. The user process can request to use a system procedure that contains privileged instructions by migrating from the user domain to the supervisor domain, where the supervisor can access $r \in R$ on behalf of the user process.

The multilevel protection scheme as implemented in Multics can be used effectively in multiprocessors. In this scheme, which is often called the *layered protection system*, the basic unit of protection is a segment. Segments are grouped into a set of n levels or classes. The layering scheme results in a nested ring structure consisting of n rings, as shown in Figure 8.9. Each level or class of segments is assigned to a distinct ring. Therefore, the implementation of the virtual address and the ST entry will have an extra field for representing the ring number of the segment.

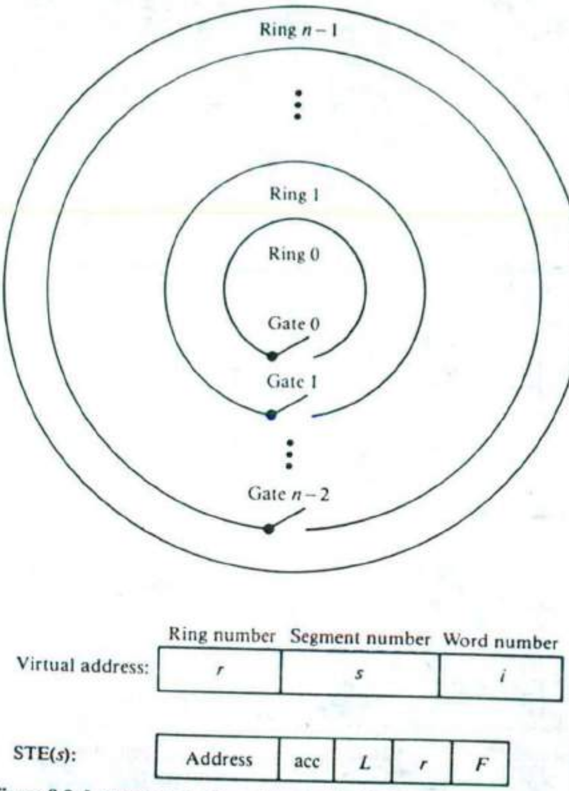


Figure 8.9 Layered protection scheme in the MULTICS and segment table entries.

The access capabilities of ring r_i is a subset of ring r_j whenever ring $r_i > r_j$, for $r_i, r_j \in \{0, 1, \ldots, n-1\}$. Access control is performed between classes and not within a class. Hence, one segment can reference another segment without a validation check if both segments are in the same class. However, the crossing of a ring boundary results in an access fault, which invokes the operating system to perform a validation check. If the crossing is from a ring r_i to ring $r_i + 1$, then the access is permitted. If the crossing is from ring r_i to ring $r_i - 1$, the operating system validates the permission. To call an inner ring, only certain entry points are permitted. These entry points are called *gates*. The segment which makes the call must present a valid key to match one of the locks at the gate. The validation process is performed by a *gatekeeper process*. The list of entry points corresponding to the set of locks is called a *gate list*.

In this system, segments that belong to a process can, for example, be assigned to a given ring. If the segments are shared by multiple processes, the disjointness of the classes of segments can be relaxed. A shared segment is assigned to a consecutive set of rings called an *access bracket*. In this case, the ring field in Figure 8.9 contains a set of two integers (low, high). A call from a segment in ring r_i to a segment with access bracket (n_1, n_2), will not fault if $n_1 \leq r_i \leq n_2$. However, if $r_i < n_1$, the access fault still occurs. A modified version of the layered protection scheme is used in the S-1 multiprocessor system. In most cases, however, the ring concept is not practical for more than three rings.

In order to provide useful conventions for sharing among processes, it is necessary to have a systematic way of describing what is to be shared and of controlling access to shared objects from various processes. This machinery can be described in terms of an idealized system called the object system, which consists of three major components: a set of objects X, a set of domains D, and an access matrix **A**. Objects are the protected entities. Typical objects are domains, files, processes and segments. Objects are assigned a unique name in the system, for example, by using a 64-bit counter. Recall that domains are entities which have access to objects. The property of a domain is that it has potentially different access than other domains. Note that objects do not necessarily belong to a domain but can be shared between domains.

The access of a domain to objects is determined by the access matrix **A**. Its rows are labeled by domain names and its columns by object names. Element $A[i, j]$ specifies the access which domain i has on object j. Each element consists of a set of strings called access attributes. Typical attributes are read (r), write (w), execute (x), wakeup (s). A domain has a y-access to an object if y is one of the attributes in that element of **A**. Associated with each attribute is a bit called the *copy flag* which controls the transfer of access.

The system itself attaches no significance to any access attribute except "owner" or to object names. Thus the relationship between, say, the file-handling module and the system is typical of the following: A user calls on the file-handler to create a file. The file-handler asks the system for a new object name n, which the system delivers from its stock of 2^{64} object names (e.g., by incrementing a 64-bit counter). The system gives the file-handler owner access to object n. The file-handler enters n in its own private tables, together with other information about the file which may be relevant (e.g., its disk address). It also gives its caller owner access to n and returns n to the caller as the name of the created file. Later, when some domain d tries to read from file n, the file-handler will examine $A[d, n]$ to see if "read" is one of the attributes of domain d, and refuse to do the read if it is not.

The sparsity of the access matrix A makes it impractical to represent as an ordinary matrix in memory. An alternative method, which is implementable, is to have a list T of pairs $\langle y: A[d, y]\rangle$ which is searched whenever the value of $A[d, x]$ is required. This pair is usually called *capability*. In general, a capability defines the rights (set of operations allowed) that a process has on an object. A capability is a system-maintained unforgettable ticket which, when presented, can be taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket according to the access rights defined.

A finite sequence of capabilities is called a capability list or *C list*. A C list associated with a process defines the running environment (running domain) of the process at a given time; that is, the set of objects that it may use and the operations which it may perform on these objects, as shown in Figure 8.10. Process execution corresponds to running over a succession of domains: migrating from one domain to another expresses the variation of the flexibility of the process. Addressing by capabilities and system structuring by domains permit the solution of protection and error-confinement problems. These classic notions are the basis of numerous capability-based addressing systems such as the C.mmp Hydra and the Intel iAPX 432 system.

The resources in a domain of a process as shown in Figure 8.10 consist of the processor *r*egister *f*ile (RF), the *l*ocal *m*emory (LM), and the local capability list. Resources outside the domain are accessed only through capabilities stored in the local C list. Any domain can have access to any object if a capability for the object appears in its local C list. On a segment machine, capability lists can be supported by special segments (C list segments). A capability can be accessed by indexing into the C list, as shown in Figure 8.11. In this illustration, the capability is divided into four parts:

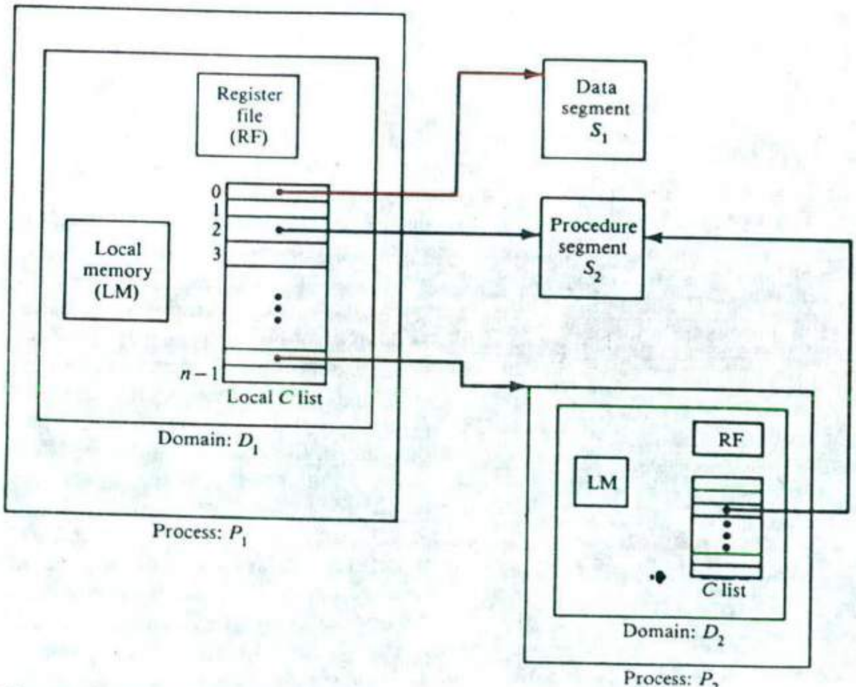


Figure 8.10 Domains, capabilities, and objects.

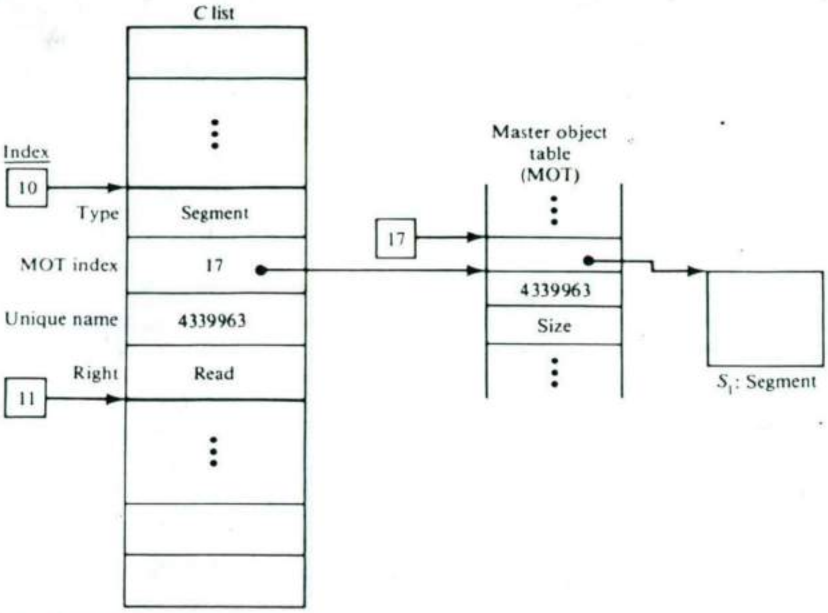


Figure 8.11 C-list and capabilities.

- A type definition of the designated object
- A pointer to a Master Object Table (MOT)
- An object identifier (unique name)
- A set of rights which defines the operations allowed on the object

The MOT entry pointed by the capability contains the absolute address of the object and its unique identifier. The MOT concept makes addressing of the object easy and a relocation of the object needs only the updating of an address in the MOT, even if this object is shared between several processes. It also makes the access slow.

In general, protection can be applied on an object or a path to the object. For example, the access-matrix concept applies the protection on the path, while the entry in the segment table applies it on the object. Protection placed on the access path to objects generally requires less overhead than protection placed on the objects. However, in some cases, both methods are required.

Capabilities have an advantage over privilege-checking in that the protection check is performed at the beginning of the object name interpretation without leaving the execution environment. Hence the error is confined to the execution environment. If a process refers to an object through C lists, it is impossible to name any object to which the process has no access rights. However, it can become wasteful in storage space and the overhead in loading and saving a C list upon activation and deactivation of the process can become substantial. If a parent process adds or deletes access rights to its C list regarding objects that are shared by its children processes, then the access rights must be modified for the several C lists of the children processes. This problem does not occur if the protection is on the object, since the change in access rights is made in one place—at the object.

8.3 MULTIPROCESSOR SCHEDULING STRATEGIES

In this section, we discuss the processor management techniques used in multiprocessor systems. The introduction of multiple processors complicates the scheduling problem. Deterministic and probabilistic models have been used to evaluate some scheduling schemes. Generally, finding an optimal algorithm for the processor scheduling problem in multiprocessors is computationally intractable. However, some dynamic-scheduling algorithms are close to optimal.

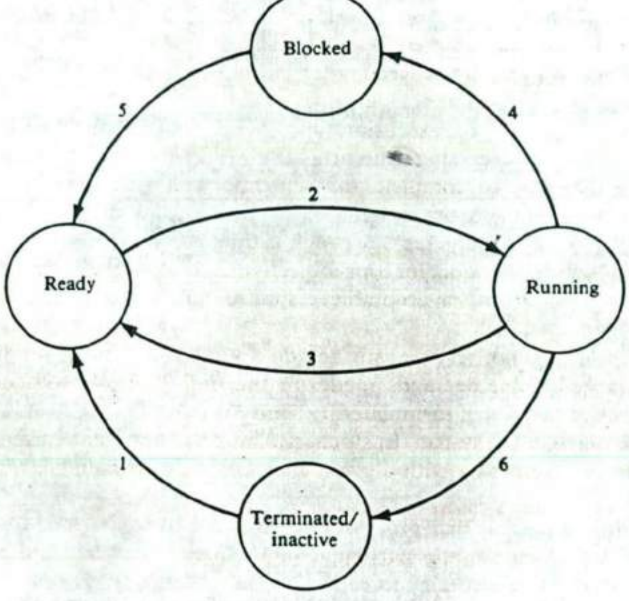
8.3.1 Dimensions of Multiple Processor Management

Multiprocessor management and scheduling have been a fertile source of interesting problems for researchers in the field of computer engineering. Replicated components, particularly those that are nonhomogeneous (heterogeneous) or asymmetric, increase the amount of management that must be provided by either the operating system or the application or both. In its most general form, the problem involves the scheduling of a set of processes on a set of processors with arbitrary characteristics in order to optimize some objective function. This involves the selection of a process for execution from a set of processes.

Basically, there are two resource-allocation decisions that are made in multiprocessing systems. One is where to locate code and data in physical memory—a *placement decision*; and the other is on which processor to execute each process—an *assignment decision*. These decisions are often trivial for a uniprocessor system in which assignments are dictated. Furthermore, the physical memory address space is accessible to the single processor in a uniprocessor system, hence the question of accessibility never occurs and memory-contention problems can be minimized by interleaving. Oftentimes, assignment decision is called processor management. It describes the managing of the processor as a shared resource among external users and internal processes. As a result, processor management consists of two basic kinds of scheduling: long-term external load scheduling and short-term internal process scheduling.

In general, active processes undergo different state transitions in the course of their lifetimes in the system. A process is in the *run* state if it is using a processor. A suspended process may enter the pool of *blocked processes*. A process is blocked if it cannot run because it is waiting for some external response, such as a wake-up signal, which may arrive to unblock it. The unblocking operation changes the status of the process to a *ready-to-run* or *ready* state, where it is eventually scheduled to a processor. Figure 8.12 illustrates the possible state transition experienced by a process. The scheduler at this level performs the *short-term process-scheduling* operation of selecting a process from the set of ready-to-run processes. The selected process is assigned to run on a processor. The *medium* and *long-term load-scheduling* operation is used to select and activate a new process to enter the processing environment. The activation of the new process causes it to be put on the ready list. Long-term load-scheduling also acts to control the degree of multiprocessing (that is, the number of active processes) in the system which, if excessive, may cause *thrashing* (see Chapter 2).

The process scheduler or dispatcher performs its function each time the running process is blocked or preempted. Its purpose is to select the next running process from the set of ready queues. The process scheduler resides in the kernel and can be considered a monitor for the ready queues. Since it is probably the most frequently executed program in the system, it should be fairly efficient to minimize operating system overhead.



Transition	Event
1	Activate process
2	Run process
3	Preempt process
4	Block process
5	Wakeup process
6	Terminate process

Figure 8.12 States of a process and their state transitions.

The ready list can either be local or global. A local ready list may be associated with each multiprogrammed processor which has a local memory. Thus a process, once activated, may be bound to a processor. The local ready list reduces the access time of the list and, hence, the overhead encountered by the dispatcher. However, the local ready list concept discourages process migration. Moreover, under light system load, the processor utilization may not be equally distributed among all processors. To permit process migration, a global ready list which resides in the shared memory may be used. This has the disadvantage of requiring more overhead in saving and restoring process states by the process scheduler. However, the standard deviation of the processor utilization is small.

The general objectives of many theoretical scheduling algorithms are to develop processor assignments and scheduling techniques that use minimum numbers of processors to execute parallel programs in the least time. In addition, some develop algorithms for processor assignment to minimize the execution time of the parallel program when processors are available.

There are basically two types of models of processor scheduling, deterministic and nondeterministic. In *deterministic* models, all the information required to express the characteristics of the problem is known before a solution to the problem, that is, a *schedule*, is attempted. Such characteristics are the execution time of each task and the relationship between the tasks in the system. The objective of the resultant schedules is to optimize one or more of the evaluation criteria. For example, in deterministic models, the execution time of each process can either be interpreted as the maximum processing time or as the expected processing time. In the former case, the time to complete the schedule would be considered the maximum time to complete the system of processes, and in the latter case, the length of the schedule represents a rough estimate of the mean length of the computation. The motivation for this objective is that, in many cases, a poor schedule can lead to an unacceptable response time or utilization of system resources.

Deterministic models are not very realistic and do not take into consideration the irregular and unpredictable demands made on the multiprocessor system. Hence, stochastic models are often formulated to study the dynamic-scheduling techniques that take place in the system. In stochastic models, the execution time of a process is a random variable t with a given *cumulation distribution function* **(cdf)** F.

Processor scheduling implies that processes or tasks are to be assigned to a particular processor for execution at a particular time. Since many tasks can be candidates for execution, it is necessary to represent the collection of tasks in a manner which conveniently represents the relationships (if any) among the tasks. Generally, we will refer to a set of related processes as a *task system* or a *job*. A job, which consists of a set of processes, is represented as a precedence graph, as shown in Figure 8.13. The nodes in the graph are tasks which may represent independent operations or parts of a single program which are related to each other in time. The collection of nodes represents a set of processes $T = \{T_1, \ldots, T_n\}$, and the directed edge between nodes implies that a partial ordering or precedence

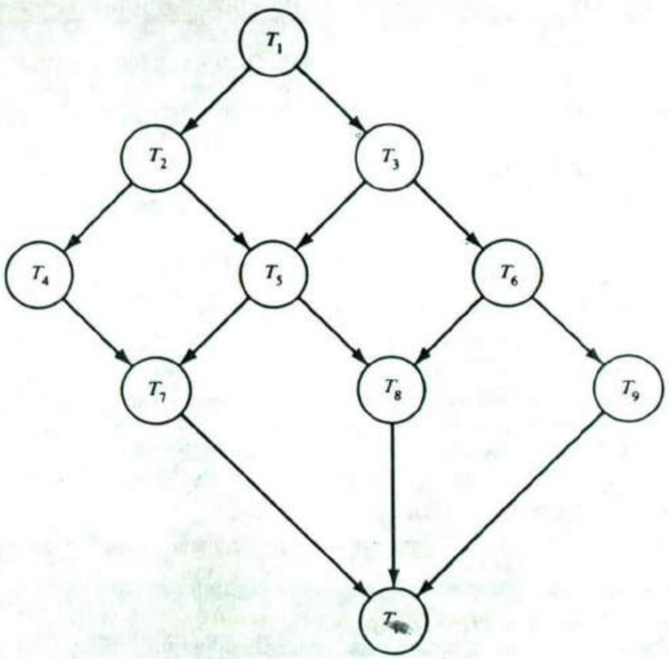


Figure 8.13 Representation of a task system, $T = (T_1, \ldots, T_{10})$.

relation $<$ exists between the processes. Therefore, if $T_i < T_j$, process T_i must be completed before T_j can be initiated. Processes with no predecessors are called *initial* processes (e.g., T_1), and those with no successors are called *final* processes (e.g., T_{10}). The individual nodes within a graph can be related to each other in a number of ways.

For example, it is possible for all processes in a graph to be independent of each other. In this case, there is no precedence relation or partial ordering between the processes and all the processes can be scheduled concurrently, provided there are enough processors available. The *width* of the task graph G, denoted by width(G), is the maximum size of any independent subset of processes. In Figure 8.13, $T_1 < T_2$, $T_1 < T_3$, $T_4 < T_7$, and $T_5 < T_7$. The width of the graph is 3. Associated with each node is a second attribute which refers to the time required by a hypothetical processor to execute the code represented by the node. Sometimes, this attribute is called the *weight* of the node. In a deterministic model, this attribute is a constant for each node, whereas, in a stochastic model, it is a random variable with a mean and standard deviation or a known distribution.

Given a computation graph and a multiprocessor system with p processors, a task assignment or a schedule must be developed such that it gives a description of the processes to be run and in what order as a function of time. The schedule must not violate any of the precedence relationships or the requirement that no more than one processor can be assigned to a task at any time.

In a multiprocessor system, one may associate a *process descriptor node* (PDN) with each executable and active process. This process may consist of a number of fields, as shown in Figure 8.14. The parent field is a pointer to the set of processes which initiated the process, the child field points to the set of processes to be initiated by the current process and the register state defines the register values of the given process. The kernel of the operating system uses the concept of PDNs to monitor the status of the processes.

In some systems, when processes are created, they exist as unrelated units, independent of each other. In other systems, the order of creation is remembered and a parent-child relationship is maintained between one process and the new process it creates. Both approaches have advantages and disadvantages. Typically, a child process is limited to using only those resources owned by its parent and is deallocated if its parent terminates. In most general-purpose systems, when a process is destroyed, its process image is returned to a pool of unallocated memory. However, in many dedicated or real-time systems, processes are never destroyed. Instead, they are created at compile time or initialization time and run forever, even at times when there is no work to do.

During the execution of a process on a processor, external stimuli such as interrupts may arrive and require urgent service because of input or output device constraints. If the interruption and subsequent resumption of the process in execution is permitted before its termination, *preemptive* scheduling is used. If

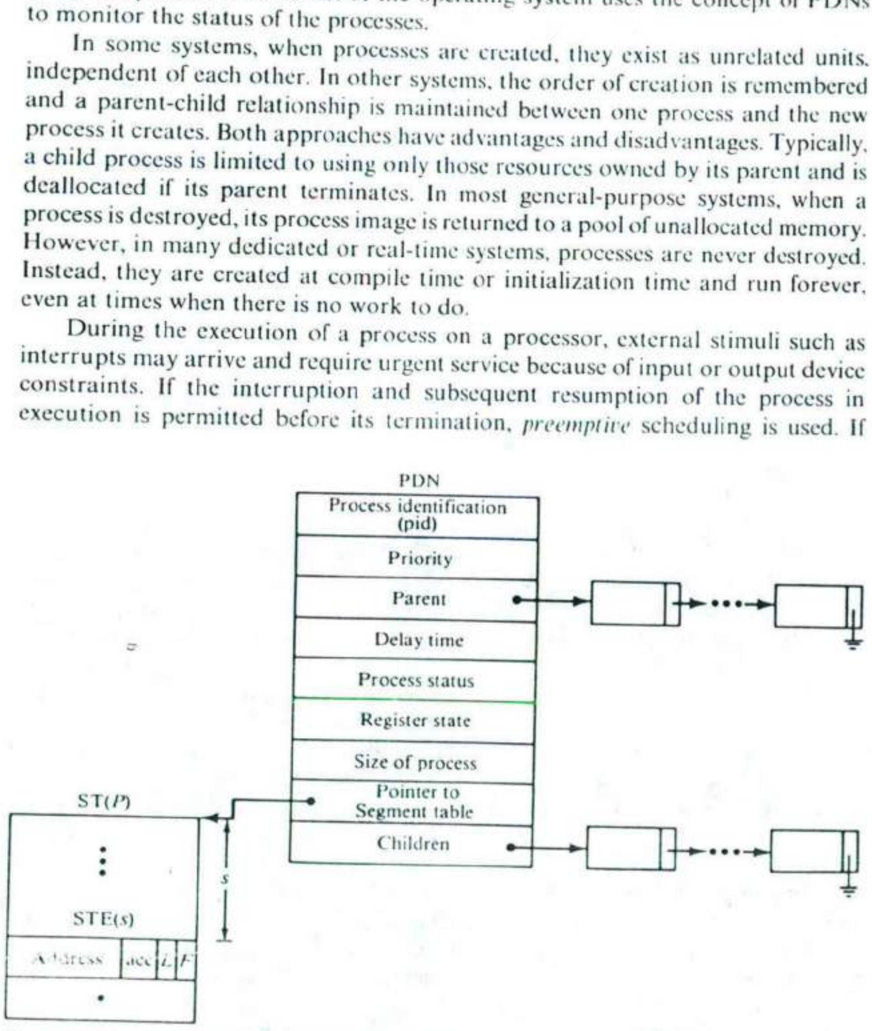


Figure 8.14 Process descriptor node of an active task.

interruption before process completion is not permitted, *nonpreemptive* or *basic* scheduling is applied. In general, preemptive disciplines generate schedules that are better than those generated by nonpreemptive disciplines. It is also true, however, that a certain penalty exists for preemptive schedules that do not exist in the nonpreemptive case. The penalty lies in the context-switching overhead, which consists of the system-interrupt processing and additional memory required to preserve the state of the interrupted process. This overhead may be acceptable if it occurs infrequently, however, in an environment in which preemption occurs frequently, unacceptable performance degradation may result.

Oftentimes, certain tasks may get preferential treatment over the others in a computer system. Systems in which this is true provide *priority scheduling*. In a priority-scheduling system, tasks are grouped into priority classes and numbered from 1 to n. It is usually assumed that the lower the priority-class number, the higher the priority; that is, tasks in priority class i are given preference over tasks in priority class j, if $i < j$. A good scheduling policy should be fair and maximize throughput; giving preference to high priority tasks, yet preventing "starvation" of low priority tasks.

Priority scheduling can be combined with preemptive or nonpreemptive scheduling to provide control policies. They are used to resolve a situation wherein a task of class i becomes a ready-to-run task when a task of class j is running in a processor, for $i < j$. In a *preemptive priority scheduling*, the running task is interrupted and the new task runs on that processor. A further refinement of this policy is the *preemptive-resume priority scheduling*, in which the task whose execution was interrupted continues execution at the point of interruption when the task is reassigned to run on the processor. In a *nonpreemptive priority* scheduling, the new ready-to-run process waits until the process currently running on the processor terminates its execution before it gains access to the processor.

Many parallel algorithms require the concurrent execution of multiple processes to achieve a significant performance speedup. The scheduling strategy plays a very important role in meeting the concurrency requirements. Usually the scheduler is designed as a mutually exclusive program which can only be executed by one process at any time. The problem which may be encountered is whether the scheduling of the set of concurrent processes cooperating in a job should be performed as a group or individually. Each process in the set could be assigned to a processor as it becomes available and runs the process immediately. Since this strategy depends on the dynamic availability of processors, the execution time of the job may be large and depends on the time the last process in the set is assigned.

If preemption of processes is permitted, the parallel processes of a job may be scheduled as a group. Processors executing low priority jobs may be preempted to release the processors for the parallel process system. The required number of processors for that job must be available before any process in the set is assigned for execution. It is, however, inefficient to allocate a processor to a process which has to delay its execution because its siblings (cooperating processes) have not yet been assigned to processors. This strategy limits the degree of decomposition to the total number of processors in the system.

A number of measures have been developed to evaluate the effectiveness of processor schedules. Some of these measures are (*a*) response or completion time, (*b*) speed-up ratio, and (*c*) processor utilization. The objectives for multiprocessor resource management and scheduling are the same performance objectives as for their uniprocessor counterparts, namely, maximizing throughput, minimizing response time, or completing processing of tasks in order of priority. Consequently, the multiprocessor schedulers have much in common with single processor schedulers.

8.3.2 Deterministic Scheduling Models

Deterministic schedules are usually displayed with timing diagrams called Gantt charts. We define some measures of performance based on Gantt charts. The flow time of a process is equal to the time its execution is completed. The flow time of a schedule is defined as the sum of the flow times of all processes in the schedule. For example, the flow times of processes T_1 and T_4 in Figure 8.15 are seven and two, respectively, while the flow time of the schedule is 25.5. The *mean flow time* is obtained by dividing the flow time of a schedule by the number of processes in the schedule. The *utilization* (or fractional busy time) of processors P_1, P_2, and P_3 is 0.93, 1.00, and 0.86, respectively. These utilization values are obtained by dividing the time during which the processor was busy by the total time during which it was available for execution. The *idle time* of P_1, P_2, and P_3 is 0.5, 0.0, and 1.0, respectively.

Figure 8.16 shows a process system schedule for a given program graph on two processors. The numbers associated with each node in the process graph represent the execution time of the process. Figure 8.16*b* gives the optimal schedule for the graph using two processors. Note that this schedule is achieved by keeping a processor idle even when there is a process to execute. Figure 8.16*c* shows that activating the schedulable process as soon as possible does not necessarily achieve an optimum schedule. The total execution time ET_1 of the process graph G on a uniprocessor is the sum of the numbers (weights) associated with each node. Hence, $\mathrm{ET}_1 = 27$. From Figure 8.16*b*, the execution time on the two-processor system is $\mathrm{ET}_2 = 15$. Therefore the speedup, $S_p = \mathrm{ET}_1/\mathrm{ET}_2 = \frac{27}{15} = 1.8$, for $p = 2$.

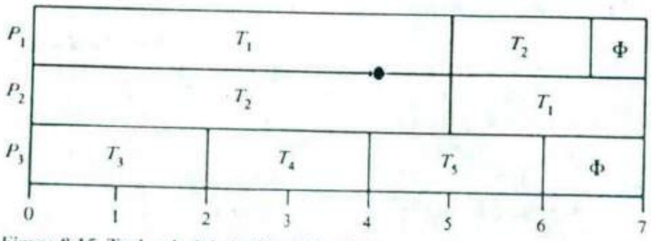
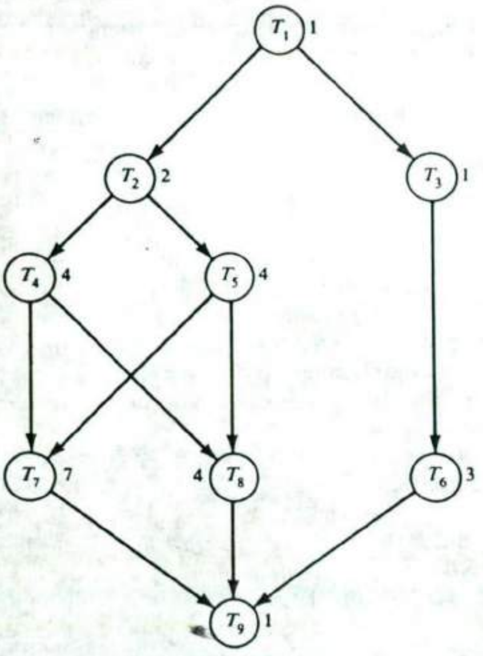
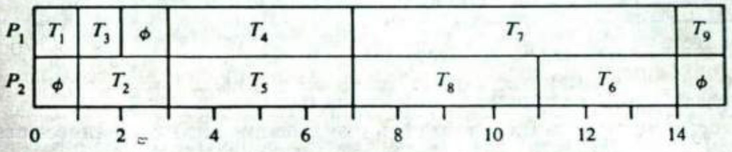


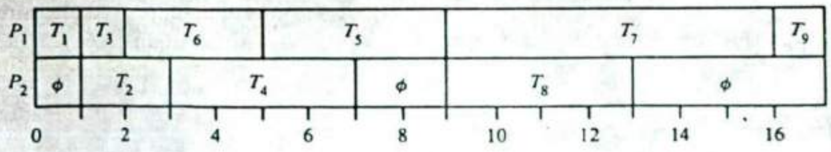
Figure 8.15 Task schedule in Gantt chart form.



(*a*) Task graph G for a set of tasks



(*b*) Optimal schedule



(*c*) Schedule when processors are activated as soon as possible

Figure 8.16 Task schedule in chart form, using $p = 2$ processors. (Courtesy of *ACM Computing Surveys*, Gonzalez, Sept. 1977.)

The mean utilization U_p of the p processor system in the case of Figure 8.16*b* is $U_p = (30 - 3)/30 = 0.9$, for $p = 2$. The reader can easily show that by increasing the number of processors to 3, the speedup does not increase. In fact, the utilization reduces to 0.6. Hence, the execution of the process graph in Figure 8.16*a* is most cost-effective on a two-processor system. The rationale behind the minimization of finishing or completion time is that system throughput can be maximized if the total computation time of each set of processes is minimized. *Throughput* is defined as the number of process sets processed per unit of time.

There are at least two reasons for minimizing the number of processors required to process a process system. The first and most obvious is cost. The second reason is the processor utilization. If the number of processors required to execute a set of processes in a given time is less than the total number of processors available, then the remaining processors can be used as backup processors for increased reliability and as background processors for noncritical computations.

A key issue in the study of processor scheduling is the amount of overhead or computation time needed to locate a suitable schedule. A scheduling algorithm is a procedure that produces a schedule for every given set of processes. An efficient scheduling algorithm is one that can locate a suitable schedule in an amount of time that is bounded in the length of the input by some polynomial. Construction of *optimal* schedules is NP-complete in many cases. NP-complete implies that an optimal solution may be very difficult to compute in the *worst* possible input case. However, construction of *suitable* schedules, that is, computing a reasonable answer for the typical input case, is not NP-complete. Therefore suitable schedules can be obtained for concurrent processes.

In this subsection, we examine deterministic schedules which can be used to optimize measures of performance. Unless stated explicitly, we assume a scheduling environment which consists of a number of identical processors, a set of processes with equal or unequal execution times and a (possibly empty) precedence order. First we consider preemptive schedules using two processors.

In order to understand the preemptive schedule (PS) on p processors, we define process graphs with mutually commensurable node weights. A set of nodes is said to be *mutually commensurable* if there exists a t such that each node weight is an integer multiple of t. In a preemptive schedule, a processor may be preempted from an executing process if such an action results in an improved measure of performance. The PS algorithms are due to Muntz and Coffman (1966).

Assume that the process graph consists of n independent processes with weights (process duration or execution times) of $t_1, t_2, \ldots, t_n$ and p processors. The optimal PS has a completion time of:

$$\omega = \max\left\{ \max_{1 \le i \le n} \{t_i\}, \frac{1}{p} \sum_{i=1}^{n} t_i \right\} \tag{8.3}$$

The optimal PS length cannot be less than the larger of the longest process or the sum of the execution times divided by the number of processors.

For their optimal algorithm, the set of nodes of unit weight in a graph are partitioned into a sequence of disjoint subsets such that all nodes in a subset are independent. All nodes in the same subset or at the same level are candidates for simultaneous execution or group scheduling. In a graph of N subsets or levels, the terminal node occupies the first level exclusively. Those nodes which may be executed during the unit time period preceding the execution of the terminal node occupy the second level, and so on. The initial or entrance node in the graph occupies the Nth level. Such an assignment of levels generate what is called *precedence partitions*.

In particular, the assignment procedure outlined above corresponds to the latest precedence partitions. That is, the assignment of nodes to levels is done in a manner which defers process initiation to the latest possible time without increasing the minimum completion time. Such a schedule is called the *latest-scheduling* strategy. This strategy assumes that the number of processors available is greater than or equal to the maximum number of processes at any level (width of G). This strategy may be contrasted with the *earliest-scheduling strategy*, which schedules a process as soon as a processor is available and the precedence constraints have been satisfied. Note that the earliest strategy produces earliest-precedence partitions.

For any arbitrary graph G, a precedence relation will exist between the subsets of the latest strategy due to the precedence which exists between the nodes in the original graph. A PS can be constructed for graph G by first scheduling the highest-numbered subset, then the subset at the next lower level, and so on. Note that when a subset consists of only one node, a node from the next lower subset is moved up if it does not violate the precedence constraints of the original graph. If each of the subsets is scheduled optimally, a *subset schedule* results. For two processors and equally weighted nodes, an optimal subset schedule for G is an optimal PS for G.

This result is extended to the case of graphs having mutually commensurable node weights. In order to generate the optimal result, it is necessary to convert graph G into another graph G_w in which all nodes have equal weights. This is done by taking a node of weight t_i and creating a sequence of n nodes such that $t_i = nt$, as illustrated in Figure 8.17. Note that the integrity of the original graph must be maintained. It can then be shown that an optimal subset schedule for G_w is an optimal PS for G, with $k = 2$.

In this approach, one must note whether the number of processes at any level is even or odd. If it is even, then all processes at that level can be executed in the minimum amount of time with no idle time for either of the two processors. If the number of processes is not a multiple of two, then the last three processes to be scheduled at that level can be executed in no less than $\frac{1}{2}$ unit, since all processes in G_w are of unit duration. By using the form shown in Figure 8.18, three processes in a given level can be executed in minimum time without processor idle time. Since scheduling in this manner ensures that no processor is idle, the subset sequence can be seen to generate a minimal-length PS. An example of the optimal PS algorithm is shown in Figure 8.19. For this example, the optimal subset sequence for G is $\{T_1\}, \{T_2, T_3\}, \{T_5, T_6, T_7\}, \{T_4, T_8\}, \{T_9, T_{10}\}, \{T_{11}\}$.

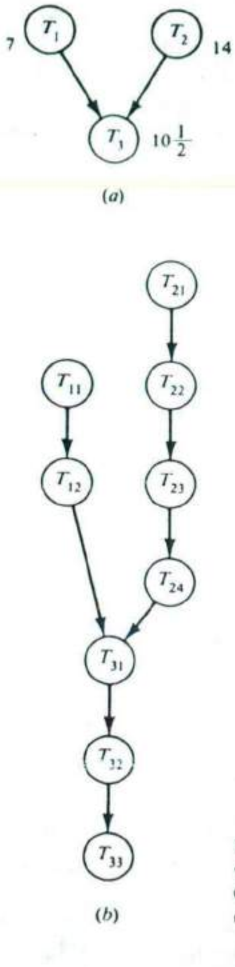


Figure 8.17 Comparison of a graph with mutually commensurable node weights with the corresponding graph having nodes of equal weight. (*a*) Graph *G* node weights $w_1 = 7$, $w_2 = 14$, $w_3 = 10\ 1/2$; (*b*) graph G_w, $w = 3\ 1/2$.

The optimal results derived above can be extended to the case in which any number of processors are allowed when the computation graph is a *rooted tree* and the node weights t_i are mutually commensurable. A *rooted tree* is one in which each node has at most one successor, with the exception of the root or terminal node, which has no successors. We discuss below some techniques for nonpreemptive schedules.

Recall that, in nonpreemptive or basic schedules, a processor assigned to a process is dedicated to that process until it is completed. The initial investigations discussed here develop optimal nonpreemptive two-processor schedules for arbitrary process orderings in which all processes are of unit duration. A particular

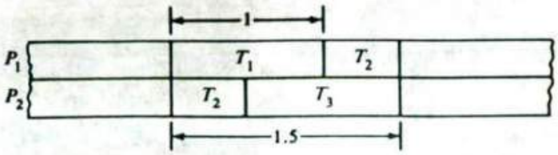
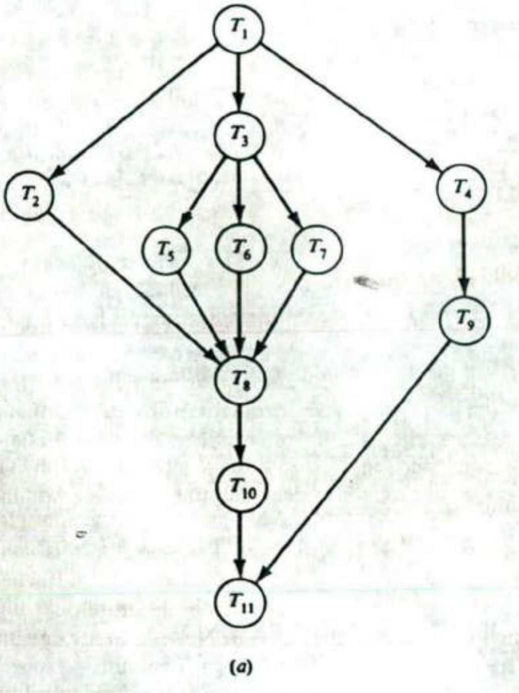


Figure 8.18 Minimum-time execution format for three unit tasks with two processors.



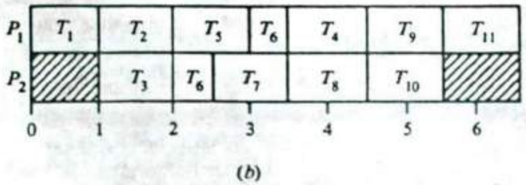


Figure 8.19 Illustration of subset sequence algorithm. (*a*) Graph *G* for a set of tasks, with all nodes having unit weight; (*b*) optimal preemptive schedule. (Courtesy of *ACM Computing Surveys*, Gonzales, Sept. 1977.)

simple class of scheduling algorithms for nonpreemptive schedules is the class of *list-scheduling algorithms.* A list-scheduling algorithm assigns distinct priorities to processes and allocates resources to the processes with highest priorities among those runnable at any time instant when the resource becomes free. A list schedule or list L for a graph G of n processes is denoted by $L = (T_1, T_2, \ldots, T_n)$ and represents some permutation of the n processes. A process is said to be ready if all of its predecessors have been completed. In using a list to generate a schedule, an idle processor is assigned to the first ready process found in the list. An algorithm for generating such an optimal list is described below.

The algorithm is a recursive procedure which begins by assigning subscripts in ascending order to the process (processes) which is (are) executed last owing to precedence constraints in the process graph. Notice that the set of successors of these processes is empty. Assignment proceeds "up the graph" in a manner that considers as candidates for the assignment of the next subscript all processes whose successors have already been assigned a subscript. Consideration of processes in this manner amounts to examining processes in a given latest-precedence partition, although the processes are not executed at a time that corresponds to this partition. In effect, the processes in a graph can be initially assigned subscripts in an arbitrary manner. This algorithm then reassigns subscripts in the method outlined above. The list is formed by listing the processes in decreasing subscript order, beginning with the last subscript assigned. The optimal schedule is formed by assigning ready processes in the list to idle processors. The algorithm is illustrated in Figure 8.20 by means of a process graph with reassigned subscripts, the resultant list L^*, and the optimal schedule.

The above algorithm does not always yield optimal results when the number of processors is increased beyond two, or when the number of processors is two and processes are allowed to have arbitrary durations. We describe a nonpreemptive scheduling method by Hu (1961). Two problems for process of unit duration were addressed. In the first case, given a fixed numbers of processors, it is required to determine the minimum time required to execute a process graph. The second case determines the number of processors required to process a graph in a given time.

We begin to arrive at a solution to these problems if we develop a labeling scheme for the nodes of the graph. A node T_i is given the label $\alpha_i = X_i + 1$, where X_i is the length of the longest path from T_i to the final node in the graph. Labeling begins with the final node, which is given the label $\alpha_1 = 1$. Nodes that are one unit removed from the final node are given the label 2, and so on. This labeling scheme makes it clear that the minimum time $\omega_{\min}$ required to execute the graph is related to $\alpha_{\max}$, the node(s) with the highest numbered label, by

$$\omega_{\min} \geq \alpha_{\max}$$

The optimal solutions by Hu are limited to rooted trees. Using the labeling procedure described above, one can obtain an optimal schedule for p processors by processing a tree of unit-length processes in the following manner:

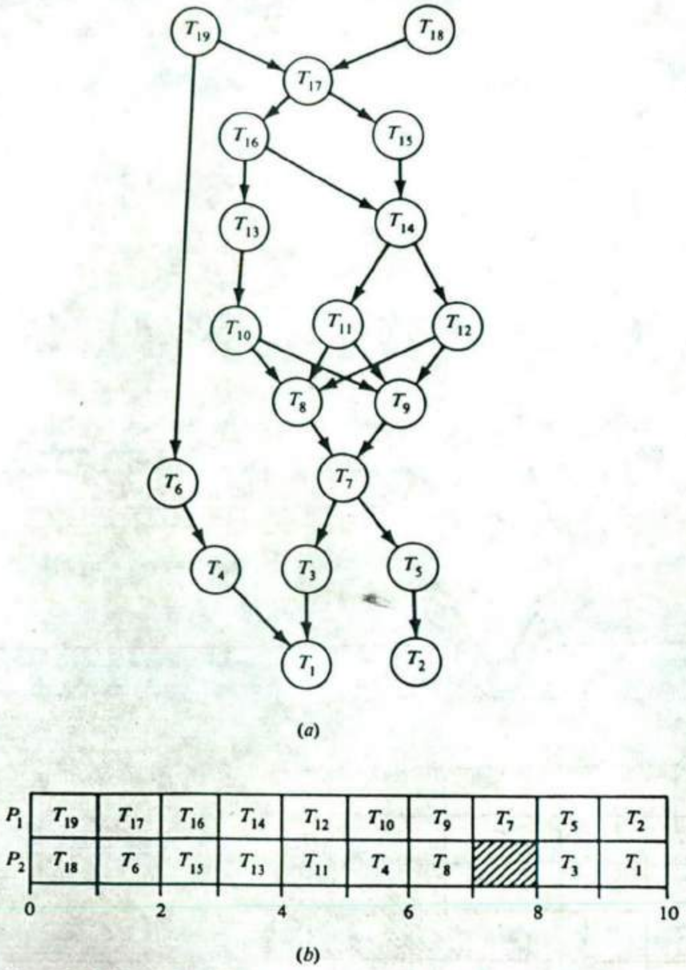
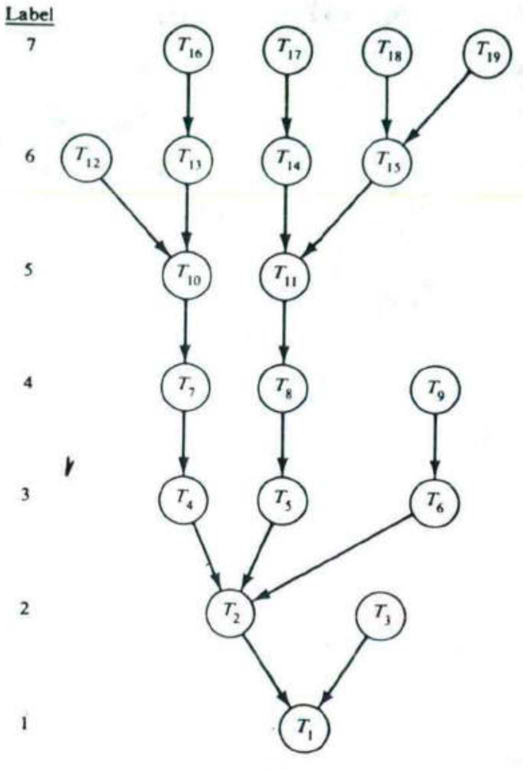

(a)
(b)
P_1: T_{19} T_{17} T_{16} T_{14} T_{12} T_{10} T_9 T_7 T_5 T_2
P_2: T_{18} T_6 T_{15} T_{13} T_{11} T_4 T_8 ▨ T_3 T_1
0 2 4 6 8 10

Figure 8.20 Illustration of Coffman and Graham algorithm. (*a*) Task graph with reassigned subscripts $L^* = (T_{19}, T_{18}, \ldots, T_1)$; (*b*) optimal schedule. (Courtesy *ACM Computing Surveys*, Gonzales, Sept. 1977.)

1. Schedule first the p (or fewer) nodes with the highest numbered label, i.e., the starting nodes. If the number of starting nodes is greater than p, choose p nodes whose α_i is greater than or equal to the α_i of those not chosen. In case of a tie, the choice is arbitrary.
2. Delete the p processed nodes from the graph. Let the term "starting node" now refer to a node with no predecessors.
3. Repeat steps 1 and 2 for the remainder of the graph.



(*a*)

P_1	T_{19}	T_{16}	T_{15}	T_{11}	T_8	T_5	T_2	T_1
P_2	T_{18}	T_{14}	T_{13}	T_{10}	T_7	T_4	▨	▨
P_3	T_{17}	T_{12}	T_9	T_6	T_3	▨	▨	▨

(*b*)

Figure 8.21 Illustration of Hu's optimal algorithm. (*a*) Rooted tree labeled according to Hu's procedure; (*b*) optimal schedule for three processors. (Courtesy *ACM Computing Surveys*, Gonzalez, Sept. 1977.)

The schedules generated in this manner are optimal under the stated constraints. The labeling and scheduling procedures are quite simple to implement and are illustrated in Figure 8.21.

Recall that the minimum time required to execute a task graph by Hu's procedure is $\alpha_{\max}$. Suppose we wish to process a graph within a prescribed time t, where $t = \alpha_{\max} + C$ and C is a nonnegative integer. The minimum number of processors p required to process the graph in time t is given by

$$p - 1 < \frac{1}{\gamma^* + C} \sum_{j=1}^{\gamma^*} \rho(\alpha_{\max} + 1 - j) < p \tag{8.4}$$

where $\rho(i)$ denotes the number of nodes in the graph with label α_i and γ^* is the value of the constant γ, which maximizes the given expression. To illustrate this result, consider Figure 8.21. For $C = 0$, for example, value γ^* occurs when $\gamma = 1$ or $\gamma = 2$. This indicates that, in order to process the graph in minimum time, four processors are needed. For $C = 1$, $t = 8$ and γ^* occurs when $\gamma = 2$ or $\gamma = 5$, and three processors are required. Varying C further, we find that three processors are required when the processes must be processed within nine units, but only two processors are needed for a maximum processing time of 10 units.

Another study by Graham shows that, for a computing system with n identical processors in which processes are assigned arbitrarily to the processors, the completion time of the set of processes will not be more than twice the time required by an optimal schedule. This bound was derived in connection with the so-called *multiprocessor anomalies*. These anomalies are derived from the counterintuitive observation that the existence of one of the following conditions can lead to an increase in execution time:

1. Replace a given process list L by another list L', leaving the set of process times μ, the precedence order $<$, and the number of processors n unchanged.
2. Relax some of the restrictions of the partial ordering.
3. Decrease some of the execution times.
4. Increase the number of processors.

A general bound has been obtained by executing a set of processes twice. During the first execution, the processes are characterized by the parameters μ, $<$, L, n, and ω (the length of the schedule), and during the second execution by μ', $<'$, L', n', and ω' such that $\mu' \le \mu$ and every constraint of $<'$ is also in $<$, i.e., $<'$ is contained in $<$. The result of this general bound is that

$$\frac{\omega'}{\omega} \le 1 + \frac{n - 1}{n'} \tag{8.5}$$

This bound is the best possible, and, for $n = n'$, the ratio $2 - 1/n$ can be achieved by the variation of any one of L, μ, or $<$.

The above result was extended to a nonhomogeneous processor system by Liu and Liu. Suppose a multiprocessor system consists of n_i processors of speed μ_i, for $i = 1, 2, \ldots, k$, such that $\mu_1 > \mu_2 > \cdots > \mu_k \ge 1$, then

$$\frac{\omega'}{\omega} \le \frac{\mu_1}{\mu_k} + 1 - \frac{\mu_1}{\sum_{i=1}^{k} n_i \mu_i} \tag{8.6}$$

Example 8.8 Consider a system with one processor of speed five and five processors of speed one. By Eq. 8.6, we have

$$\frac{\omega'}{\omega} \leq \frac{5}{1} + 1 - \frac{5}{10} = \frac{11}{2}$$

Comparing this bound with that in Eq. 8.5 for a multiprocessor system with 10 identical processors of speed 1 (by substituting five identical processors of speed 1 for the processor of speed 5), the ratio $2 - \frac{1}{10}$ is achieved. The determination of a close to optimal schedule is more important for a heterogeneous system than for a homogeneous system.

Because of the limitations on optimal algorithms, bounds have been derived for the behavior of nonoptimal algorithms. The concept of precedence partitions can be used to generate bounds for processing time and the number of processors for graph structures whose nodes require unit-execution time. As indicated earlier, precedence partitions group processes into subsets to indicate the earliest and latest times during which processes can be initiated and still guarantee minimum execution time for the graph. This time is given by the number of partitions and is a measure of the longest path in the graph. For a graph of N levels, the minimum execution time is $\omega = N$ units. In order to execute a graph in this minimum time, the lower bound on the number of processors p required is given by

$$p \geq \max\left[\max_{1 \leq i \leq N} |L_i \cap E_i|,\ \max_{1 \leq i \leq N}\left[\frac{1}{i}\sum_{i=1}^{N} |L_i|\right]\right] \tag{8.7}$$

and the upper bound on the number of processors p required is given by

$$p \leq \min_{1 \leq i \leq N}\left[\max_{1 \leq i \leq N} |L_i|,\ \max_{1 \leq i \leq N} |E_i|\right] \tag{8.8}$$

In both cases, L_i and E_i refer to the ith latest and earliest precedence partitions, respectively, and $|x|$ represents the cardiality of the set x. The processes contained in $L_i \cap E_i$ are called *essential processes*. Those processes contained in the ith subset given by $L_i \cap E_i$ must be initiated $i - 1$ units after the start of the initial process in the graph to guarantee minimum execution time.

8.3.3 Stochastic Scheduling Models

Using nondeterministic techniques, the execution time of a process T_i is given by the random variable t_i, with cumulative distribution function F_i. Given a process graph G, let t_G be the random variable representing total execution time (the time from when all processes are started until the last process terminates). Assume t_G has a **cdf** F_G. There is a class of process graphs for which F_G can be expressed simply in terms of F_i. Below, we give a methodology, developed by Robinson (1979), for estimating F_G for this class.

Process graphs In order to determine the possible execution of job T, we define chains and simple process graphs. A subgraph of a process graph G is a *chain* if the set of processes in the subgraph are totally ordered. The length of a chain is the number of processes in the chain. If, in a chain, the initial process is T_i and the final process is T_j, we say it is a chain from T_i to T_j. A subgraph of a process graph G that is a chain is said to be a chain in G. In the following definition, a class of process graphs is defined for which F_G can be expressed simply in terms of the **cdf**s of process execution time F_i. Let $C_1, C_2, \ldots, C_m$ be all chains from initial to final processes in G. For each chain C_i containing processes $T_{i_1}, T_{i_2}, \ldots$, let X_i be the expression $x_{i_1}x_{i_2} \ldots$, formed by concatenating the polynomial variables $x_{i_1}, x_{i_2}, \ldots$, associated with processes $T_{i_1}, T_{i_2}, \ldots$, respectively. Then G is said to be *simple* if the polynomial $X(G) = X_1 + X_2 + \cdots + X_m$ can be factored so that each variable appears exactly once. Examples of simple and nonsimple process graphs are shown in Figure 8.22.

The class of parallel algorithms represented by simple process graphs are exactly those that can be written in block-structured languages with parallel blocks, provided no synchronization takes place between any of the components of a parallel block.

A set of processes is *independent* if, for any two processes T_i and T_j in the set, neither $T_i < T_j$ not $T_j < T_i$. In this example (Figure 8.22*a*), processes in set $\{T_1, T_2, T_4\}$ are independent. So also are processes in set $\{T_3, T_4\}$. Figure 8.22 shows some process graphs to explain the simplicity of graphs.

Let K be the number of processors in the system. If $K \geq \text{width}(G)$, each process in G begins execution immediately after the last predecessor completes. Let $C_1, C_2, \ldots, C_m$ be all the chains from initial to final processes in graph G. Also let the execution time of process T_i be t_i with **cdf** F_i. Then the total execution time of task system G is the maximum of the execution times of all the chains in G. That is,

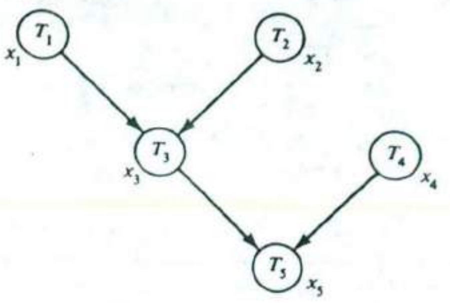
$$t_G = \max_{1 \leq i \leq m} \sum_{T_j \in C_i} t_j \tag{8.9}$$

Note that + and max are commutative and associative operations, respectively. Moreover, + distributes over max. For example, $\max(a, b) + c = \max(a + c, b + c)$. Thus, if G is simple, the expression for t_G above can be factored in terms of max and + so that each random variable appears only once. Then, if the t_i's are independent, the expression for F_G may be found by substituting F_i for t_i, * (convolution) for +, and $\cdot$ (multiplication) for max in the expression for t_G. The convolution of **cdfs** F_1 and F_2 is written as follows:

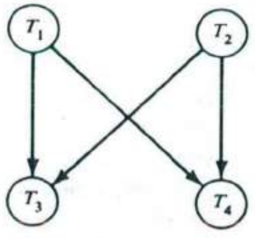
$$F_1 * F_2(t) = \int_{-\infty}^{\infty} F_1(t-u)F_2\, du \tag{8.10}$$

For the example in Figure 8.23, there are three chains, $C_1 = T_1T_3T_5$, $C_2 = T_2T_3T_5$, and $C_3 = T_3T_5$. Therefore

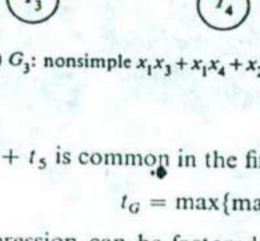
$$t_G = \max\{(t_1 + t_3 + t_5), (t_1 + t_3 + t_5), (t_4 + t_5)\}$$



(*a*) G_1: simple $x_1x_3x_5 + x_2x_3x_5 + x_4x_5 = [(x_1 + x_2)x_3 + x_4]x_5$



(*b*) G_2: simple $x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 = (x_1 + x_2)(x_3 + x_4)$



(*c*) G_3: nonsimple $x_1x_3 + x_1x_4 + x_2x_4$

Figure 8.22 Examples of simple and nonsimple task graphs. (Courtesy *IEEE Trans. Software Engg*. Robinson, January 1979.)

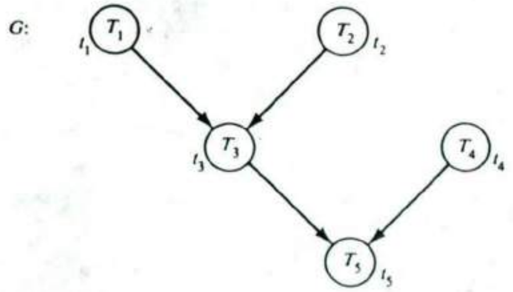
Since $t_3 + t_5$ is common in the first two summations,

$$t_G = \max\{\max(t_1, t_2) + t_3 + t_5, t_4 + t_5\}$$

This expression can be factored further by noting that t_5 is common to both summations:

$$t_G = \max\{\max(t_1, t_2) + t_3, t_4\} + t_5 \tag{8.11}$$

Chains:
$C_1 = T_1T_3T_5$
$C_2 = T_2T_3T_5$
$C_3 = T_4T_5$



$$t_G = \max \{\Sigma\, t_j\} = \max \{(t_1 + t_3 + t_5), (t_2 + t_3 + t_5), (t_9 + t_5)\}$$

$$T_j \epsilon C_i = \max \{\max(t_1, t_2) + t_3, t_4\} + t_5 \quad \text{and} \quad 1 \le i \le m$$

$$F_G = \{(F_1F_2) * (F_3F_4)\} * F_5$$

Figure 8.23 Computation of t_G and F_G for task graph G. (Courtesy *IEEE Trans. Software Engg.*, Robinson, Jan. 1979.)

In Eq. 8.11, each random variable appears only once. Hence, F_G may be found by the substitution rule:

$$F_G = (((F_1 \cdot F_2) * F_3) \cdot F_4) * F_5 \tag{8.12}$$

As another example, consider the four-process merge-sort depicted by the process graph in Figure 8.24. Process S_i performs the sorting of one distinct subfile a_i, which is a fourth of file A. After the pair of either subfiles a_1 and a_2 or a_3 and a_4 are sorted, they are merged by the execution of process M_1 or M_2, respectively.

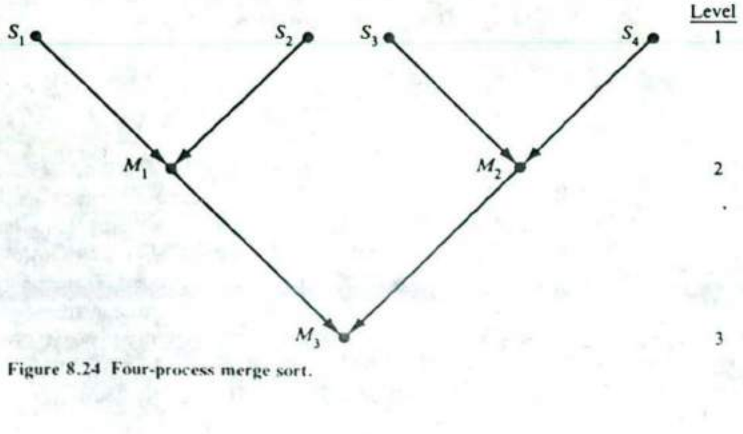


Figure 8.24 Four-process merge sort.

Merging is a method of combining two or more sorted files into a composite sorted file. In Figure 8.24, each M_i is a merge of the sorted subfiles produced by its immediate predecessors.

If all permutations of keys are equally likely, then the execution times of S_1, S_2, S_3, and S_4 have the same **cdf** and the execution times of M_1 and M_2 have the same **cdf**. Let the **cdf** of the execution times of the S_i's be F_1 and that of M_1 and M_2 be F_2. Furthermore, let the **cdf** of the execution time of M_3 be F_3. Then width$(G) = 4$. G is simple and the process-execution times are independent. Let the execution times of the S_i's and M_j be t_{S_i} and t_{M_j}, respectively. Hence, the execution time of the four-process merge-sort is

$$t_G = \max\{\max(t_{S_1}, t_{S_2}) + t_{M_1}, \max(t_{S_3}, t_{S_4}) + t_{M_2}\} + t_{M_3}$$

Since $t_{S_1}, t_{S_2}, t_{S_3}, t_{S_4}$ have the same **cdf** F_1 and t_{M_1}, t_{M_2} have the same **cdf** F_2, the **cdf** of t_{M_3} is F_3:

$$F_G = (F_1^2 * F_2) \cdot (F_1^2 * F_2) * F_3 = (F_1^2 * F_2)^2 * F_3 \tag{8.13}$$

This should be compared with the **cdf** of the execution time for a one-process (sequential) merge-sort:

$$F_{\text{seq}} = F_1 * F_1 * F_1 * F_1 * F_2 * F_2 * F_3 \tag{8.14}$$

since the execution time for the sequential merge-sort is

$$t_{\text{seq}} = t_{S_1} + t_{S_2} + t_{S_3} + t_{S_4} + t_{M_1} + t_{M_2} + t_{M_3} \tag{8.15}$$

Notice that Eq. 8.15 assumes that the processing environment of the sequential merge-sort is the same as the concurrent merge-sort. In practice, this is not true, since the sequential merge-sort does not encounter interprocess-communication problems or memory conflicts which create overheads in the concurrent merge-sort. Hence, in practice, t_{seq} is usually less than that predicted by Eq. 8.15. In the next section, we consider the effect of these overheads on the performance of the algorithm.

Let μ_G and μ_{seq} be the mean execution times of the probability density functions $f_G = F'_G$ and $f_{\text{seq}} = F'_{\text{seq}}$, respectively. We can then estimate the theoretical speedup of the four-process merge-sort as

$$S_p = \frac{\mu_G}{\mu_{\text{seq}}} \tag{8.16}$$

Equation 8.9 is not very useful when the **cdfs** of the process execution time are not known. Bounds can be derived for the mean execution time by using more limited knowledge about the execution times of processes. Let us denote the expected value of a random variable x by $E(x)$. The level of a process T in a process graph G is the maximum length of any chain in G from an initial process to T. The *depth* of G, denoted by depth(G), is the maximum level of any process. Given a process graph G with the number of available processors $K \geq$ width(G) and with the t_i independent, let $C_1, C_2, \ldots, C_m$ be all chains in G from initial to final processes. Also let H_i be the set of all processes of level i, for $1 \le i \le L$, where $L = \text{depth}(G)$. For any set of n random variables $\{x_i\}$,

$$E\left(\max_{1 \le i \le n} \{x_i\}\right) \ge \max_{1 \le i \le n} \{E(x_i)\} \tag{8.17}$$

from which the lower bound follows. For the upper bound, let $t_0 \equiv 0$ and define $f(i, j) = 0$ if $C_i \cap H_j$ is empty; otherwise $f(i, j)$ is the index of the single process in $C_i \cap H_j$. Then from Eq. 8.9,

$$t_G = \max_{1 \le i \le m} \left(\sum_{1 \le j \le L} t_{f(i,j)} \right) \le \sum_{1 \le j \le L} \max_{1 \le i \le m} (t_{f(i,j)}) \tag{8.18}$$

Therefore

$$\max_{1 \le i \le m} \left(\sum_{T_j \in C_i} E(t_j) \right) \le E(t_G) \le \sum_{1 \le i \le L} E\left(\max_{T_j \in H_i} t_j \right) \tag{8.19}$$

The upper bound in Eq. 8.19 is useful only if something can be said about $E(\max\{t_j\})$. An applicable result from *order statistics* is that, if the random variables $x_1, x_2, \ldots, x_m$ are independent and identically distributed (i.i.d.) with the mean μ and standard deviation σ, then

$$E\left\{ \max_{1 \le i \le m} \{x_i\} \right\} \le \mu + \frac{m-1}{\sqrt{2m-1}} \sigma \tag{8.20}$$

Hence, if the number of available processors $K \ge \text{width}(G)$, the t_i's are independent, $\text{depth}(G) = L$ and the m_j processes on level j have identically distributed execution times with the mean μ_j and standard deviation σ_j, then

$$\sum_{1 \le j \le L} \mu_j \le E(t_G) \le \sum_{1 \le j \le L} \left(\mu_j + \frac{m_j - 1}{\sqrt{2m_j - 1}} \sigma^j \right) \tag{8.21}$$

Queueing model Probabilistic models are often formulated to investigate the properties of dynamic scheduling methods that take the form of queueing systems. These require the specification of certain characteristics and attributes of the queueing system, such as the probability distribution functions of the interarrival times of processes, the service times of the processes, and the specification of the service discipline. The service or queueing discipline is the scheduling rule which determines both the sequence in which processes are executed and the processor-occupancy period each time a process is selected for service. A number of assumptions are usually made regarding queueing systems to make the analytical model tractable. These include the independence of processes and the statistical independence of the interarrival and service times.

A very simple model of scheduling in a multiprocessing system consists of p identical processing elements and a single infinite queue to which processes arrive. This model may be appropriate for a system with a global ready list and no preemptions. The mean processing time of processes on each processor is $1/\mu$ and the mean interarrival time of processes to the system is $1/\lambda$. Assuming that the service and interarrival times are exponentially distributed and the service discipline for processes is the common first-come-first-serve (FCFS), various performance factors can be obtained. Figure 8.25 illustrates the resulting queueing model in which processes arrive at a rate λ and are serviced at a rate μ. The utilization of the processors is

$$\rho = \frac{u}{p} \tag{8.22}$$

where u is the traffic intensity and is defined by $u = \lambda/\mu$. The mean response time of processes is [Kleinrock (1976)]

$$\overline{R}(\rho, u) = \frac{C(\rho, u)}{\mu p(1-\rho)} + \frac{1}{\mu} \tag{8.23}$$

where $C(\rho, u)$ is Erlang's C formula and is given by

$$C(\rho, u) = \frac{u^p}{u^p + p!(1-\rho)\sum_{n=0}^{p-1} \frac{u^n}{n!}}$$

There are a number of other scheduling algorithms, such as *round robin* (RR), preemptive, and nonpreemptive priority service disciplines, which can be modeled by the use of queueing systems. In an RR service discipline, each time a process is selected for execution, it is selected from the head of the ordered queue and allocated a fixed duration of run time called the *time slice* or *quantum*. If a process terminates execution before the end of the quantum, it departs from the processor. If at the

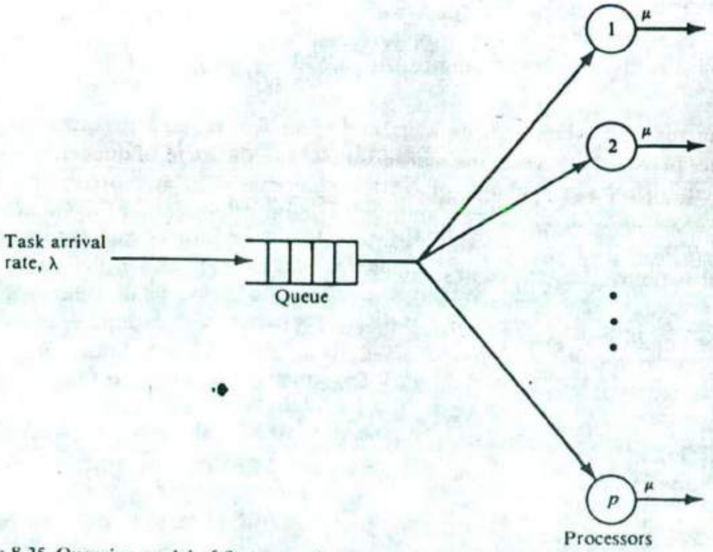


Figure 8.25 Queueing model of first-come-first-serve scheduling discipline in a multiprocessor system.

end of the quantum the process has not completed its execution (requires additional quantum), it is recycled to the end of the queue to await its next selection. New process arrivals simply join the end of the queue. The RR service discipline can be combined with the preemptive priority discipline to create a multilevel round-robin scheduling discipline. This discipline is used to give higher priority processes more frequent control than lower priority ones. Policies based on priority can be static (if the priority of a process remains fixed) or dynamic (if the priority of a process is allowed to change).

In the RR service discipline, a process in the run state is interrupted at the end of its quantum and may enter the ready state. An external event may cause the blocking of a running process. These transitions may necessitate a context switch. Furthermore, a running process can cause an explicit process switch by invoking a privileged instruction. For example, in the case of a fault, the process can cause a trap which switches context to the operating system, as in the IBM 370 supervisor call (SVC) instruction to be described in Chapter 9.

Long-term scheduling operations are used to control the load on the multiprocessor system by making decisions on activating new processes. One method to implement the schedule is to use priority queues for incoming processes. Prioritization of processes in a system may result in indefinite postponement of low priority processes if the arrival rate of the high priority processes is high. A set of processes which cooperate to solve a problem may be given higher priority than a single independent process.

Since there are many processors as well as memory modules to be scheduled, it may be useful to perform *group scheduling*, in which a set of related processes are assigned to processors to run simultaneously. Group scheduling can be extended to make placement decisions for groups of objects at a time, or to swap groups of related objects in and out. These different group schedulers have several possible advantages. First, if closely related processes run in parallel, blocking due to synchronization and frequency of context switching may be reduced. These will in effect aid in increasing performance. Second, if placement decisions are made for a group of objects with known reference patterns, the "distance" between the various processes and their referenced objects might be minimized. Hence, effective memory management for a set of related processes is easier since the time period for sharing is restricted to the short presence of the processes in the system. In general, a group assignment will not be very successful in lessening the number of context switches unless the processes within the group are "in step" so that few of them will be blocked from lack of input or other synchronization requirements.

8.4 PARALLEL ALGORITHMS FOR MULTIPROCESSORS

In this section, we describe and classify the various types of parallel algorithms. The characterization of parallel algorithms will help in the design and analysis of these algorithms. Some example algorithms are given. Techniques are shown to determine the performance of MIMD parallel algorithms.

8.4.1 Classification of Parallel Algorithms

Although extensive research has been performed on SIMD algorithms, there are few results available concerning the specification, design and analysis of MIMD multiprocessor algorithms. That is the basis for this section. A parallel algorithm for a multiprocessor is a set of k concurrent processes which may operate simultaneously and cooperatively to solve a given problem. If $k = 1$, it is called a *sequential algorithm*. To ensure that a parallel algorithm works correctly and effectively to solve a given problem, processes interact to synchronize and exchange data. Hence, in a task system, there may be some points where the processes communicate with other processes. These points are called *interaction points*. The interaction points divide a process into stages. Therefore, at the end of each stage, a process may communicate with some other processes before the next stage of the computation is initiated.

Because of the interactions between the processes, some processes may be blocked at certain times. The parallel algorithms in which some processes have to wait on other processes are called *synchronized algorithms*. Since the execution time of a process is variable, depending on the input data and system interruptions, all the processes that have to synchronize at a given point wait for the slowest among them. This worst case computation speed is a basic weakness of synchronized algorithms and may result in worse than expected speedup and processor utilization.

To remedy the problems encountered by synchronized parallel algorithms, *asynchronous parallel algorithms* exist for some set of problems. In an asynchronous algorithm, processes are not generally required to wait for each other and communication is achieved by reading dynamically updated global variables stored in shared memory. However, because of the concurrent memory accesses performed, conflicts may occur which will introduce some small delay in processes accessing common variables. For convenience, we shall often refer to synchronized and asynchronous parallel algorithms simply as synchronized and asynchronous algorithms, respectively.

Another alternative approach to constructing parallel algorithms is *macro-pipelining*, which is applicable if the computation can be divided into parts, called stages, so that the output of one or several collected parts is the input for another part. The program flow is illustrated in Figure 8.26. Because each computation part is realized as a separate process, communication costs may be high unless communication is achieved by address transmission. The question may arise as to whether to move the output data to the site of the next process in the pipeline or to move the next process, in particular its code, to the site of the data.

As an example, consider a simple pipeline compiler. Different processes are responsible for lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. Source input is lexically analyzed and the recognized lexemes are input to the syntax-analysis process, thus building input for the semantic analyzer that, in turn, produces a tree for the code generator. Generated code is adapted by the final optimization process before being archived as the final

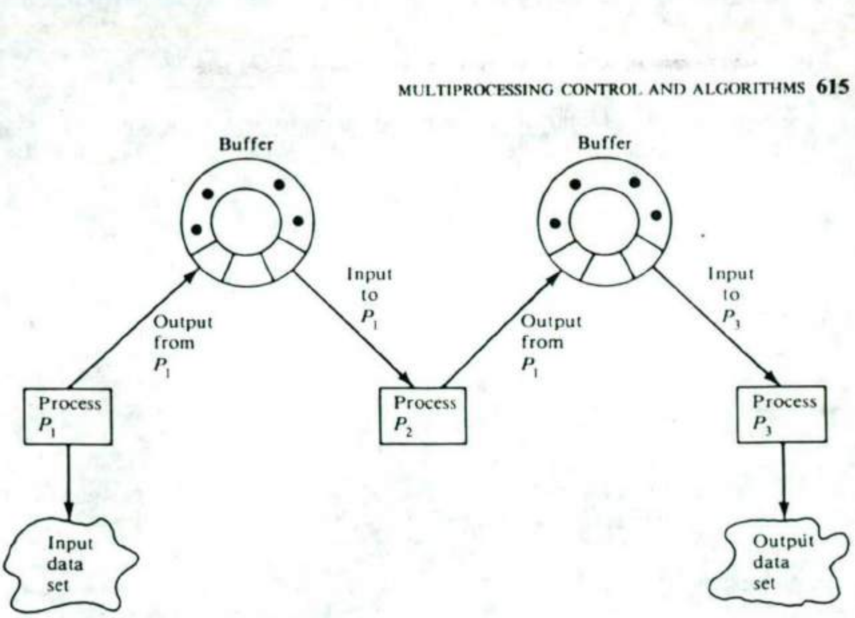


Figure 8.26 Program flow in macropipelines.

compiler output. Note that the processes that result from pipelining are heterogeneous, while those resulting from partitioning are homogeneous.

The time taken to execute a fixed stage of a process is a random variable satisfying some cumulative distribution function. The fluctuations may be due to the variability of the processor's speeds and the input to the stage. A process may be blocked at the end of a stage because it is waiting for inputs in a synchronized algorithm or for the entering of a critical section in an asynchronous algorithm. The blocking time of a process is the total time that the process is blocked. If the multiprocessor system is heterogeneous, the execution time of a process will be smallest if the process is assigned to run on a faster processor. As an illustration of the variability due to input, we recall that the number of comparisons needed to sort n elements by the Quicksort algorithm ranges from $0(n \log_2 n)$ to $0(n^2)$, depending on the ordering of the input elements. The fluctuations in execution time may also result from delays due to memory conflicts, system interrupts, page faults, cache misses and the system work load. A typical source of nonnegligible overhead is that due to the execution of synchronization primitives. Synchronization primitives are needed for synchronizing processes and implementing critical sections.

An algorithm which requires execution on a multiprocessor system must be decomposed into a set of processes to exploit the parallelism. Two methods of decomposition naturally arise: static decomposition and dynamic decomposition. In *static decomposition*, the set of processes and their precedence relations are known before execution. In *dynamic decomposition*, the set of processes changes during execution. Static decomposition algorithms offer the possibility of very low process communication, provided the number of processes are small; however, their adaptability is limited. Dynamic decomposition algorithms can adapt effectively to variations in execution time of the process graph, but only at the expense of high process communication and other design overheads.

8.4.2 Synchronized Parallel Algorithms

A synchronized parallel algorithm is a parallel algorithm consisting of processes with the property that there exist a process such that some stage of the process is not activated until another process has completed a certain stage of its program. The synchronization can be performed by using the various synchronization primitives discussed in Section 8.1. For example, suppose it is required to compute the matrix $\mathbf{Z} = \mathbf{A}\cdot\mathbf{B} + (\mathbf{C} + \mathbf{D})\cdot(\mathbf{I} + \mathbf{G})$ by maximum decomposition. A synchronized parallel algorithm may be constructed by creating three process P_1, P_2, and P_3, as shown in Figure 8.27. Processes P_1, P_2, and P_3 consist of two, one, and two stages, respectively, as shown below.

Example 8.9

```
var W, Y: shared real; var $S_w$, $S_y$: semaphore; initial $S_w$ = $C_y$ = 0;
cobegin
  Process $P_1$: begin
                 V ← A × B; // stage 1 of $P_1$ //
                 P($S_y$);
                 Z ← V + Y; // stage 2 of $P_2$ //
                 end
  Process $P_2$: begin
                 W ← C × D; // stage 1 of $P_2$ //
                 V($S_w$);
                 end
  Process $P_3$: begin
                 X ← I + G; // stage 1 of $P_3$ //
                 P($S_w$);
                 Y ← W + X; // stage 2 of $P_3$ //
                 V($S_y$);
                 end
coend
```

Clearly, the activation of the second stage of process P_3 is subject to the condition that process P_2 is completed. Si•ilarly, the second stage of P_1 cannot be initiated unless the second stage of P_3 is completed. Hence, the set of processes P_1, P_2, and P_3 is a synchronized parallel algorithm.

Since the time taken by a stage of a process is a random variable, synchronized algorithms have the drawback that some processes may be blocked at a given time, thereby degrading the performance of the algorithm. To illustrate the effect of the drawback, consider a synchronized algorithm with n processes. Assume

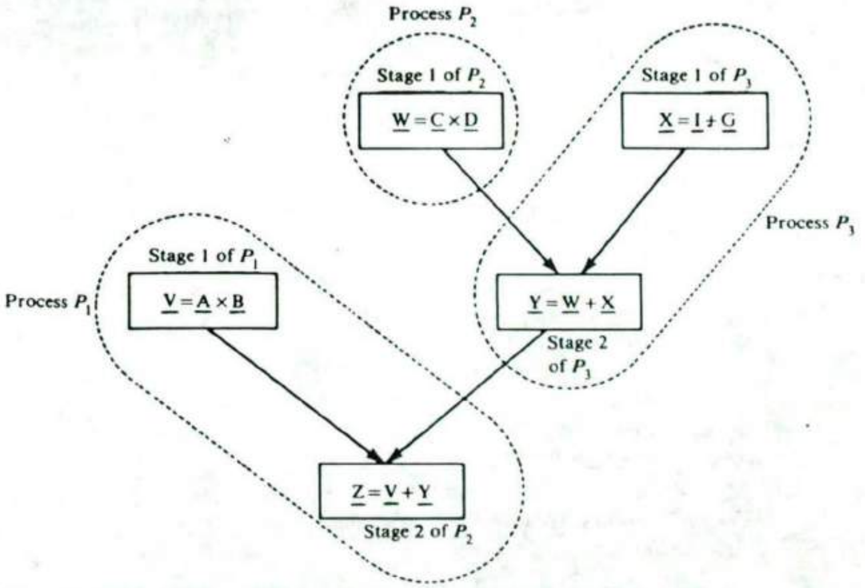

Process P_2
Stage 1 of P_2
$\underline{W} = \underline{C} \times \underline{D}$
Stage 1 of P_3
$\underline{X} = \underline{I} + \underline{G}$
Process P_3
Stage 1 of P_1
Process P_1
$\underline{V} = \underline{A} \times \underline{B}$
$\underline{Y} = \underline{W} + \underline{X}$
Stage 2 of P_3
$\underline{Z} = \underline{V} + \underline{Y}$
Stage 2 of P_2

Figure 8.27 Example of a synchronized algorithm with synchronizing stages.

that this algorithm is run on a homogeneous multiprocessor system with n processors and the algorithm takes T_n. During the execution of the algorithm, let t_i denote the total time that i processes are running; that is, $n - i$ processes are blocked. Hence, $T_n = \sum_{i=0}^{n} t_i$. Assume that the algorithm can be run on a uniprocessor system in a time $T_1 \leq \sum_{i=1}^{n} (i \cdot t_i)$. Therefore, the speedup S_n on an n-processor system is bounded by

$$S_n = \frac{T_1}{T_n} \leq \frac{\sum_{i=1}^{n} (i \cdot t_i)}{\sum_{i=1}^{n} t_i} \tag{8.24}$$

where $S_n \leq n$ is obvious.

The degradation of performance can be clarified further by considering the class of synchronized parallel algorithms where only identical stages of processes are synchronized. Synchronized parallel algorithms which are adapted from SIMD algorithms are generally of this type. Consider the execution of a set of processes in which it is required to synchronize n identical stages and the time taken by the ith stage is a random variable t_i. Since the stages are all identical, the t_i's are identically distributed random variables with a mean, say, of t. Synchronization of the n stages means a new stage of any process can not be activated until all n stages are complete. Hence, the expected time taken by the synchronized stage of any process is the mean $\overline{T}$ of the random variable $T = \max_{1 \leq i \leq n}\{t_i\}$, rather than

t. In general, T is larger than t. The ratio $T/t = \lambda_n$ is the *penalty factor* for synchronizing the n identical stages. If the penalty factor is large, the performance of the synchronized algorithm is largely degraded. The speedup bound S_n and the penalty factor λ_n give some indications of the performance of synchronized algorithms.

Synchronized iterative algorithms In practice, a large number of problems are solved by iterative methods. For example, zeros of function f may be computed by Newton's iteration method:

$$x_{i+1} = x_i - f'(x_i)^{-1}f(x_i) \tag{8.25}$$

where $f'(x)$ is the derivative of $f(x)$. The solution of a linear system of an equation by iteration is of the form

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i + \mathbf{b} \tag{8.26}$$

where the $\mathbf{x}_i$, $\mathbf{b}$ are vectors each of size n and $\mathbf{A}$ is an $n \times n$ matrix.

A common application of iterative methods is in solving elliptic differential equations with boundary conditions (boundary-value problems). A simple but important equation of physics is Poisson's equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial u^2}{\partial y^2} = f(x, y) \tag{8.27}$$

When $f(x, y) = 0$ for all x, y, this equation is known as the Laplace equation. The boundary-value problem consists of finding the function $u(x, y)$ that satisfies Eq. 8.27 within a closed region D and conditions prescribed on the boundary of D. Let D be a square domain in R^2. Also let the value of u be fixed on the boundary of D (denoted by $\overline{D}$). That is $u(x, y) = f(x, y)$ for all $x, y \in \overline{D}$. This is the Dirichlet problem. To solve this boundary-value problem on a digital computer, the domain D is sampled by superimposing a rectangular m by n grid or *mesh*. The distance between any two mesh points on any horizontal or vertical line is the *mesh width*, denoted by h. If h is small enough, we can approximate Eq. 8.27 by

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2} &\simeq \frac{u(x + h, y) + y(-h, y) - 2u(x, y)}{h^2}\\ \frac{\partial^2 u}{\partial y^2} &\simeq \frac{u(x, y + h) + y(x, y - h) - 2u(x, y)}{h^2}\end{aligned} \tag{8.28}$$

A discrete approximation to Eq. 8.27 is thus

$$4u(x, y) - y(x + h, y) - u(x - h, y) - u(x, h + h) - y(x, y - h) = -h^2 f(x, y) \tag{8.29}$$

Considering only the points in the mesh, we can rewrite Eq. 8.29 as

$$u_{i,j} = \tfrac{1}{4}(b_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) \tag{8.30}$$

for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$, where $u_{i,j} = u(ih, jh)$ and $b_{i,j} = -h^2 f(ih, jh)$. Equation 8.30 relates the update formula for any point (i, j) to only its nearest neighbors. A linear system of equations results in the $m \cdot n$ unknown values of the $\{u\}$ at all the points on the mesh inside D. Note that the dimension of the mesh is m by n.

The solution can be found iteratively by various methods. Two methods are discussed in this section. In the first method, we use a synchronized algorithm to solve the *partial differential equation* (PDE). In the next subsection, we will discuss an asynchronous algorithm to solve the PDE problem. However, in both cases, we assign a contiguous subset of the grid array $\{u_{i,j}\}$ to a process. One possible partitioning scheme is to make each subset consist of a number of consecutive rows of the grid. Thus, when the grid is stored in row-major format, a contiguous set of elements is allocated for each process. The synchronized algorithm consists of computing locally the values of the rth iterate $\{u_{i,j}^{(r)}\}$ from the values of the $(r-1)$th iterate $\{u_{i,j}^{(r-1)}\}$ as follows:

$$u_{i,j}^{(r)} = \tfrac{1}{4}(b_{i,j} + u_{i+1,j}^{(r-1)} + u_{i-1,j}^{(r-1)} + u_{i,j+1}^{(r-1)} + u_{i,j-1}^{(r-1)})$$

When all processes have completed the computation of their iterates locally, the variables are updated in all other processes. It is after this synchronization step that the $(r+1)$th iteration is initiated. This procedure is continued until the iterates converge to within an acceptable tolerance. The time in which a problem is solved depends both on the speedup and the convergence of the algorithm.

In general, an iterative algorithm can be described by an iterative function

$$\mathbf{x}_{i+1} = \phi(\mathbf{x}_i, \mathbf{x}_{i-1}, \ldots, \mathbf{x}_{i-d+1}) \tag{8.31}$$

where $\mathbf{x}_i \in R^n$.

The problems that can be formulated by an iteration function as described in Eq. 8.31 are numerous. They include relaxation methods for solving differential equations, solutions to linear and nonlinear systems of equations, and searches for extrema of functions. One approach to partitioning iterative algorithms for multiprocessors is to exploit the parallelism within the iteration function. Among all the possible partitions of ϕ, one tries to obtain a set of parallel tasks of the same size or, at least, of the same complexity so that the execution times of the tasks are *i*ndependent and *i*dentically *d*istributed (i.i.d.).

For example, in the case of Jacobi's iteration to solve a *l*inear *s*ystem of *e*quation (LSE), Eq. 8.31 reduces to Eq. 8.26. If the system has a dimension n, we can allocate p processors, each updating n/p unknown iterates, assuming that n is divisible by p. This decomposition strategy of an iteration function is called *vertical* decomposition. In this case, if the matrix $\mathbf{A}$ has no sparsity structure, all the tasks have identical stochastic properties. Since some instances of the iterates x or some elements of the matrix $\mathbf{B}$ may be null, the processing time of each task is a random variable. However, the **cdf** of the processing time is the same for all tasks. This definition assumes also that the processors are identical and under control of the same integrated operation system. For example, one may observe that in the iteration of Eq. 8.25, the evaluations of f and f' at $\mathbf{x}_i$ can be done in parallel, and in the matrix iteration of Eq. 8.26, all the components of the vector $\mathbf{x}_{i+1}$ can be computed simultaneously. Another strategy to implement Eq. 8.31 on a multiprocessor is to exploit the fluctuations in the speed of a process. In this case, the idea is to use more than one process to compute the same function in parallel and expect that the process which obtains the result first takes less than the average time.

In the following discussion, we give an example of a synchronized algorithm that locates the zero of a monotonically increasing continuous function $f(x)$. It is assumed that $f(x)$ has opposite signs at the endpoints l and u such that the interval is $|u - l|$, as shown in Figure 8.28. The process terminates when $|u - l| < \varepsilon$, a permissible error. It should be noted that the algorithm presented can be easily modified to deal with discrete f, and thus can be used to search for a desired item in an ordered list.

The zero-searching algorithm is iterative and consists of n slave processes and a master process. The master process divides the given interval $u - l$ into $n + 1$ subintervals, each of size $\Delta = |(u - l)/(n + 1)|$. Each slave process i evaluates the function at $x_i + l + i\Delta$ as a stage of the process. When all of these evaluations complete, the master process compares the computed function values for the sign

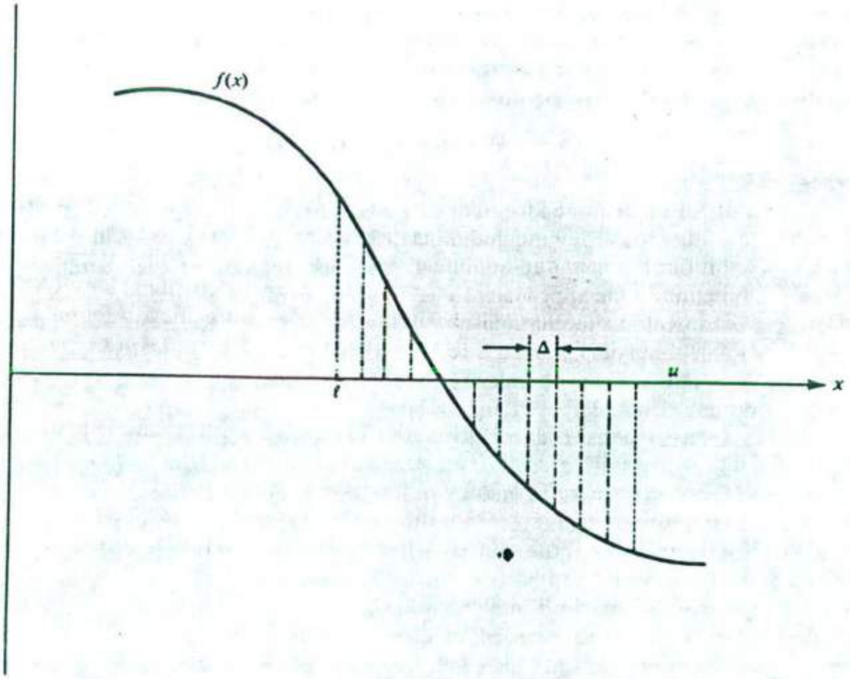


Figure 8.28 Finding the zero of a function $f(x)$ with 10 processors, where processor p_i evaluates $f(x)$ at $x = l + i\Delta$. $\Delta = |u - l|/(n + 1)$.

change that indicates the presence of a root in a particular subinterval. This becomes the new interval of uncertainty to be subdivided for the next iteration. This cycle continues until the size of the interval containing the root is sufficiently reduced. The following example explains the zero-searching algorithm.

Example 10

```
real procedure rootf(f, l, u, n)
begin
function f;
var Δ, l, u, y[1:n]: shared real;
var i: shared integer;
while |u − l| > ε do
  begin
    Δ ← |u − l|/(n + 1); // compute subinterval //
    parfor i=1 until n do // create n slave process //
      begin
        y[i] ← f(l + iΔ); // evaluate function, f(x) //
      end
    l ← l + Δ; i ← 1; // obtain new interval of uncertainty //
    while sign (y[i] = sign (y[i + 1]) do
      begin
        l ← l + Δ; i ← i + 1;
      end
    u ← l + Δ;
  end
  z ← (l + u)/2; // zero of function, f(x), is z //
end procedure rootf
```

The key feature of this algorithm is the synchronization that occurs between the slave processes. Each slave process which completes its evaluation of the function is blocked. The two sequences of statements "$l \leftarrow l + \Delta; i \leftarrow i + 1;$" are not executed until all n slave processes have completed their evaluations of the function. Every slave process is awakened from the blocked state when the next iteration begins, and all the slave processes become eligible to resume execution simultaneously. The nature of the parallel solution demands this synchronization policy.

Let the time needed to evaluate f at a point in the interval be a random variable t with mean $\bar{t}$, and time needed to determine the new uncertainty interval and to check the stopping criteria be another random variable c with mean $\bar{c}$. For this example, we assume that $\bar{t} \gg \bar{c}$, so that c can be ignored in the analysis. It is also assumed that the execution time of the synchronization primitive can be ignored.

In evaluating the relative performance of the synchronized parallel zero-searching algorithm, we note that, on a uniprocessor, the binary search produces the best known search method and takes at most $\lceil \log_2 |u - l| \rceil$ function evaluations. Hence the expected running time is $\lceil \log_2 |u - l| \rceil \bar{t}$.

For the synchronized parallel algorithm, it is clear that every iteration reduces the length of the interval of uncertainty by a factor of $n + 1$, when n slave processors are used. Therefore, the algorithm uses $[\log_{n+1}|u - l|]$ iterations and is optimal. However, the expected time for each iteration is $\lambda_n t$ rather than t, where λ_n is the penalty factor of synchronizing n function evaluations. Therefore, the expected running time of the algorithm is $[\log_{n+1}|u - l|] \cdot \lambda_n t$. Since the speedup is of the order of $\log n$, it performs poorly for large n. Hence when n is large a different search scheme that is efficient must be devised. The synchronized parallel algorithm can be inefficient when λ_n is large also, which usually happens when n is large.

Example 8.11 This example is a synchronized parallel algorithm which is iterative. In this case, the iteration function is decomposed so that each iteration step is performed by more than one process, and the processes are synchronized at the end of each iteration. Consider the solution of a linear system of equations with two concurrent processes, using Eq. 8.32. The most natural technique is to decompose each vector $\mathbf{x}_i$ into two segments $\mathbf{x}_i^{(1)}$ and $\mathbf{x}_i^{(2)}$, each of size $n/2$ (assuming n is divisible by 2) and update them by two parallel process as follows:

$$\begin{bmatrix} \mathbf{x}_{i+1}^{(1)} \\ \mathbf{x}_{i+1}^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_i^{(1)} \\ \mathbf{x}_i^{(2)} \end{bmatrix} + \begin{bmatrix} \mathbf{b}^{(1)} \\ \mathbf{b}^{(2)} \end{bmatrix} \tag{8.32}$$

where $\mathbf{x}_{i+1}^{(1)} = \mathbf{A}_{11}\mathbf{x}_i^{(1)} + \mathbf{A}_{12}\mathbf{x}_i^{(2)} + \mathbf{b}^{(1)}$ and $\mathbf{x}_{i+1}^{(2)} = \mathbf{A}_{21}\mathbf{x}_i^{(1)} + \mathbf{A}_{22}\mathbf{x}_i^{(2)} + \mathbf{b}^{(2)}$.

That is, at an iteration step, each process updates half of the components and starts the next iteration only after both processes have completed updating their iterates.

8.4.3 Asynchronous Parallel Algorithms

In each asynchronous parallel algorithm, there is a set of global variables accessible to all processes. When a stage of a process is completed, the process reads some global variables. Based on the values of the variables read together with the results just obtained from the last stage, the process modifies a subset of the global variables and then activates the next stage or terminates itself. In many cases, in order to ensure logical correctness, the operations on global variables are programmed as critical sections.

Therefore, in an asynchronous algorithm, the communications between processes are achieved through the global variables or shared data and there is no explicit dependency between processes as in synchronized parallel algorithms. The main characteristic of an asynchronous parallel algorithm is that its processes never wait for inputs at any time but continue execution or terminate according to whatever information is currently contained in the global variables. However, it should be noted that processes may be blocked from entering critical sections which are needed in many algorithms.

For illustrative purposes, we show an asynchronous iterative algorithm corresponding to the familiar Newton's iteration for finding the zeros of a function $f(x)$. In this case, we conveniently create three global variables v_1, v_2, and v_3 to contain the current values of $f(x)$, $f'(x)$, and x, respectively. For example, after the $(i + 1)$th iteration of Eq. 8.25, $f(x_{i-1})$, $f'(x_{i-1})$, and x_i are updated as $f(x_i)$, $f'(x_i)$, and x_{i+1}, respectively. Suppose that the evaluation of $f'(x)$ is computationally more expensive than that of $f(x)$, then a reasonable asynchronous iterative algorithm consisting of two processes P_1 and P_2 can be defined as follows. Let process P_1 update variables v_1 and v_3, while process P_2 updates v_2. The program below shows a sketch of processes P_1 and P_2.

Example 8.12

```
function f, f';
var v_1, v_2, v_3: shared real;
  cobegin
Process P_1: begin
             while ⟨termination criteria S not satisfied⟩ do
               begin
               $v_1 \leftarrow f(v_3)$;    // step 1 of P_1 //
               $v_3 \leftarrow v_3 - v_2^{-1} v_1$; // step 2 of P_1 //
                 end
               end P_1
Process P_2: begin
             while ⟨termination criteria S not satisfied⟩ do
               $v_2 \leftarrow f'(v_3)$; // step 1 of P_2 //
             end P_2
  coend
```

From the program it could be seen that, as soon as a process completes updating a global variable, it proceeds to the next updating by using the current values of the relevant variables without any delay. Suppose that the iterates are labeled in the order they are computed by step 2 of process P_1. Then, in general, the iterates generated do not satisfy the recurrence relation of Eq. 8.25. For example, if the initial values of the variables are $v_1 = f(x_0)$, $v_2 = f'(x_0)$ and $v_3 = x_1$, then the sequence and time period of step completions for each iteration within each process may be illustrated by a timing diagram, as shown in Figure 8.29. The number i on a demarcation on the timing diagram indicates the point where the ith iteration starts for that process. Then, for this illustration,

$$x_2 = x_1 - f'(x_0)^{-1}f(x_0)$$
$$x_3 = x_2 - f'(x_1)^{-1}f(x_2)$$
$$x_4 = x_3 - f'(x_2)^{-1}f(x_3)$$

From the concurrent program given above for P_1 and P_2, the recurrence relation that is generally followed by the execution of the processes is

$$x_{i+1} = x_i - f'(x_j)^{-1}f(x_i) \tag{8.33}$$

where $j \leq i$. Therefore, the iterates generated by the asynchronous iterative algorithm are different from those generated by the sequential algorithm or synchronized iterative algorithms. It is difficult to derive any general theory for

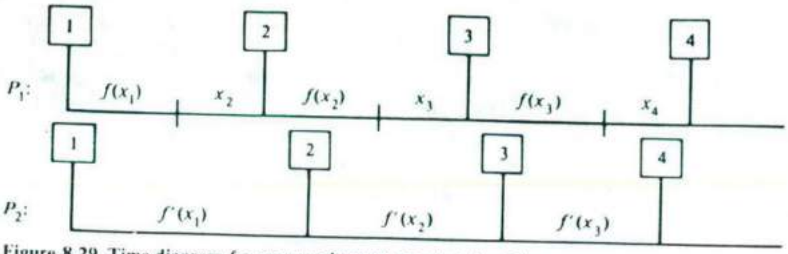


Figure 8.29 Time diagram for an asynchronous parallel algorithm.

the properties of the sequence $\{x_i\}$ because of the fluctuations of the speed of a process. Moreover, since the iterates generated by an asynchronous iterative algorithm in general do not satisfy any deterministic recurrence relation such as Eq. 8.25, it is difficult to obtain a general theory concerning conditions for convergence or the speed of convergence.

The design of an asynchronous iterative algorithm for a general iterative process (Eq. 8.31) involves the identification of some set of global variables $\{v[1], v[2], \ldots, v[n]\}$ such that each iterative step can be regarded as computing the new values of the v_i's from their old values. In general, it is desirable to choose the v_i's so that the updating of each v_i constitutes a significant portion of the work involved in one iteration. For example, consider the matrix iteration of Eq. 8.26. In this case, $v[i]$'s may be chosen as segments of equal size of the components in a vector iterate. After the $v[i]$'s have been chosen, concurrent processes which update the $v[i]$'s asynchronously can be defined as follows. Suppose there are n elements each in the vector $\mathbf{x}_i$ and $\mathbf{b}$. The set of global variables $\{v[1], v[2], \ldots, v[n]\}$ can be partitioned into p subsets, each of size $n/p = s$ (assuming that p divides n). The kth process updates the subset $\{v[(k-1)s+1], \ldots, v[ks]\}$, where $v[j]$ represents the current value of the jth component of the vector $\mathbf{x}_i$. Below is a p-process (Eq. 8.31) involves the identification of some set of global variables algorithms to solve the linear system of equations represented by Eq. 8.26.

Example 8.13

```
var v[1:n]: shared real;
parfor k = 1 until p do
  for i = (k - 1)s + 1 until ks do
    begin
    var acc: real;
    var A[(k - 1)s + 1: ks, 1: n] : real;
    var b[(k - 1)s + 1: ks] : real;
      acc ← 0.0;
      for j = 1 until n do
      acc ← acc + A[i,j] · v[j];
      v[i] ← acc + b[i];
    end
```

The above asynchronous iterative algorithms require parallelism inside the iteration function ϕ. It is possible to construct an asynchronous parallel algorithm to speed up the iterative process (Eq. 8.31) and not use any parallelism inside the iteration function ϕ. These algorithms, called *simple asynchronous* iterative algorithms, always generate the same sequence of iterates as the sequential algorithm. In general, these algorithms do not achieve speedup by sharing work; instead, the speedup is achieved by taking advantage of the fluctuations in the evaluation time. Below is an example of a simple asynchronous iterative algorithm which consists of k identical processes $P_1, \ldots, P_k$, each of which evaluates the iteration function ϕ by using the most recent iterates available at the time the evaluation is performed. In the concurrent program given below, i and x_i are global variables while j is local to the process. The value of variable i is the index of the iterate which was most recently computed; hence, the "if" statement is executed as a critical section.

```
var s: semaphore; initial s = 1;
var i, x[1:n] : shared : real;
parfor i ← 1 until k do
  begin
    while termination condition S not satisfied do
      begin
        j ← i + 1;
        x[j] ← $\phi$ (x[j − 1],x[j − 2], . . . ,x[j − d]);
        P(s);
        if i < j then i ← j;
        V(s);
      end
    end
```

The main advantage of simple asynchronous iterative algorithms is their general applicability. The algorithms are not restricted to numerical iterative processes only. In fact, they can be employed to speed up any sequence of processes. These algorithms are particularly attractive when decomposition of the processes is difficult. There are, however, some disadvantages. First, note that critical sections are needed in the algorithms. Second, it seems that unless fluctuations in computation time due to the system are large and coefficient of variation of the random variable t (time needed for one evaluation of the iteration ϕ by one process) is large, the speedup of the algorithm will be quite limited.

Concurrent quicksort algorithm In order to implement a parallel algorithm to sort an array of numbers, we discuss an example using the quicksort technique. The parallel algorithm consists of a variable number of processes which are created dynamically and assigned to processing elements (PE). The processes share the array of elements and a stack. The stack contains descriptors for continuous subsets of the array that have not yet been sorted. The stack must be accessed in a mutually exclusive manner. The stack is initialized to contain a single descriptor which describes the whole array.

In each pass, a process tries to pop a descriptor for a new subset from the shared stack. If successful, the process partitions the subset into two smaller ones consisting, respectively, of all elements less than and greater than some estimated median value. This median is computed as the mean of the first, last and middle elements of the subset. After this partitioning, a descriptor for the shorter of the new subsets is pushed onto the stack, and the longer subset is further partitioned in the same way. This procedure is continued until the number of elements in each subset is no more than a preselected threshold value. This subset may then be sorted in a single process using the simple insertion sort method.

The above parallel algorithm was implemented with a 20,480-element array and run on the Cm*. A brief description of the Cm* architecture was given in Section 7.1. Figure 8.30 shows the speedup of the parallel algorithm as a function of the number of processes for three operating environments. The speedup is relative to the speed of a uniprocess with the threshold value set to 10. The graphs are shown for a threshold value of 10. The three operating system environments consist of Smap, Medusa, and Staros. Smap is just a basic monitor with no operating

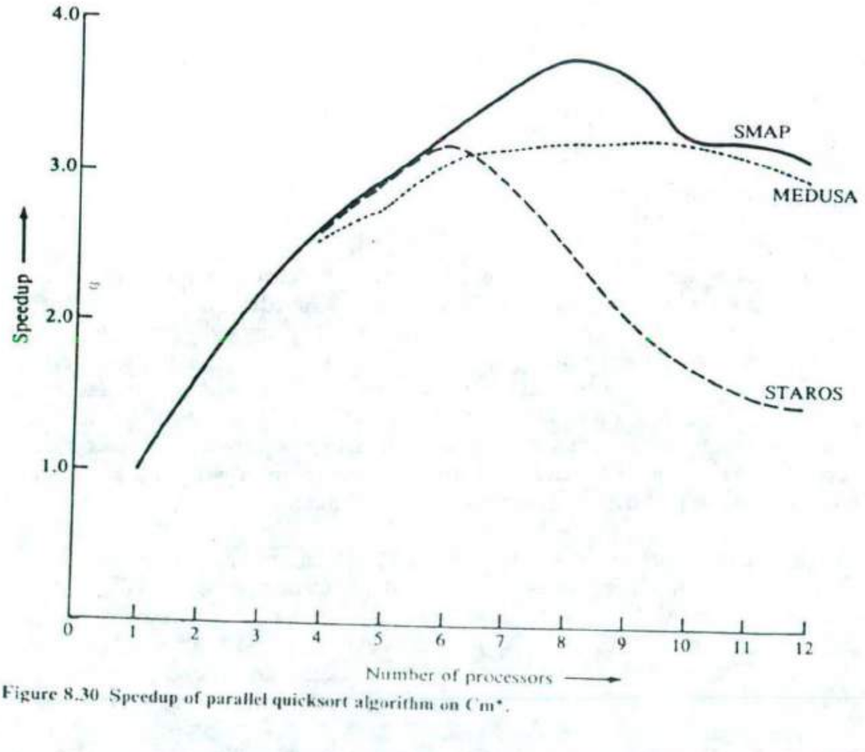


Figure 8.30 Speedup of parallel quicksort algorithm on Cm*.

system features. Medusa and Staros are two different operating systems designed for the Cm*. Notice the effect of the operating system on the performance of the algorithm. This is due to the overhead incurred by the invocation of the operating system functions for process scheduling and other chores. The performance peaks for a degree of decomposition between 6 and 8. This is due to the heavy contention of references to all the shared data located in one of the computer modules.

Asynchronous PDE A purely asynchronous method can be constructed to solve the PDE problem. In this method, each process updates the value of each point using the current values of immediate neighbors directly from the shared array so that it is available for other processes. This reduces the working space because no buffers are needed (as in the synchronized algorithm) and also assumes that the newest and probably the best approximation is used as soon as it is computed. The only variable that has to be locked against simultaneous use is the one that records the number of processes that have not finished their computations yet. This variable is accessed once per iteration. A counter is kept by each process to denote the number of performed iterations. Since each process has the same amount of work to do during each iteration, the value of its counter is a good measure of the relative speed of the process.

Experiments were performed on the Cm* to measure the performance of the asynchronous PDE algorithm using a 150×150 grid and fixed-point arithmetic. Figure 8.31 illustrates the speedup obtained with various degrees of decomposition

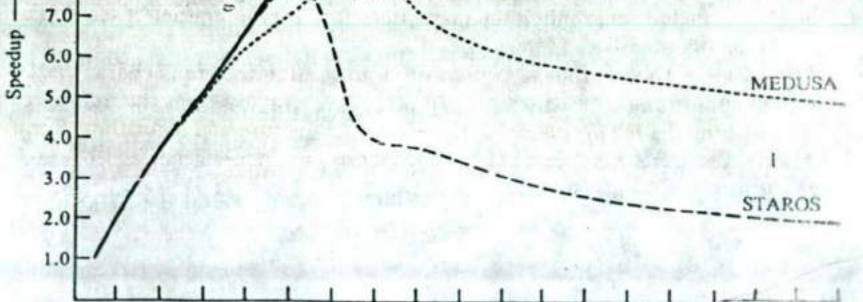


Figure 8.31 Speedup of purely asynchronous PDE algorithm on the Cm*.

and operating system environments. As long as most of the processes run in the cluster contain the global data, the speedup is almost linear. Otherwise, the speedup resembles that of the slowest process. In general, the distribution of the global data among the clusters will affect the performance. Also, the convergence depends on the relative speed of the various processes evaluating different parts of the grid. In general, asynchronous algorithms, if available, are preferred to synchronized algorithms provided the computations converge to the desired result.

8.4.4 Performance of Parallel Algorithms

Techniques developed in Section 8.3 do not account for the communication overhead caused by the interconnection topology. In this section, we study a methodology for evaluating the performance of a parallel algorithm which interacts with an architecture. Synchronized iterative algorithms are used as an example with loosely coupled multiprocessor systems. Finally, the effect of the degree of decomposition of a parallel algorithm on the performance is discussed.

Performance measures The effectiveness of a multiprocessor for a synchronized iterative algorithm depends on the performance features of the algorithm. In tightly coupled systems, performance is affected by memory interference. Conversely, the cost of interprocessor communication is the dominant factor for loosely coupled systems. In this section, we develop simple and approximate analytic models to estimate the relative effects of synchronization and interprocessor communication on the performance of a synchronized iterative algorithm. We recall from Section 8.4.2 that, in synchronized iterative algorithms, a cycle of operations is repeated until the result or a satisfying approximation to the result is obtained.

The model discussed here is for vectorial decomposition of iterative algorithms into i.i.d. tasks. The general case is very complex. Only the concurrent phase of the algorithm is modeled. Within each iteration, there may be a purely sequential phase which may include checking the convergence criterion after all processors are synchronized or when initiating a new iteration. Including such a sequential phase is a simple extension of the model.

An implementation of an algorithm on a given architecture is characterized by a set of performance features, $\{f_1, f_2, \ldots, f_N\}$, extracted from the analytical model. Let F be the *feature space* for the given architecture and algorithm. F can be seen as the product space of the one-dimensional spaces generated by each feature:

$$F = \{f_1\} \times \{f_2\} \times \cdots \times \{f_N\} \tag{8.33}$$

The topology of the space F is complex. The feature values may be real along some coordinate axes, and discrete along some others. A *performance index* for a given architecture is a real function defined on F by the analytical model. Local maxima of the index indicate operating points in F where the architecture and the algorithm implementation are particularly well matched with respect to the index.

The power of analytical models resides in the estimation of the impact on the performance of a given feature or subset of features in isolation.

A symmetric multiprocessor system is made up of a set of identical processors. Generally the computation on each processor consists of a random number of instruction executions. The randomness in this number results, for example, from the evaluation of a function whose definition varies over its domain or of a function computed by a series expansion. In a minicomputer or a microprocessor, an instruction cycle consists of a variable number of machine cycles. Typical machine cycles are instruction fetch, operand fetch, and execution cycle. We will thus distinguish between memory access cycles and execution cycles. When a request for a memory word is rejected because of conflicts, the processor automatically retries by initiating a new memory access cycle. *Memory access time fluctuations* result. The execution part of an instruction cycle also has a random duration. Its duration may depend on the operand values. The number of machine cycles in the execution of an instruction is thus random, resulting in *execution time fluctuations*.

Since we are strictly concerned with the modeling of synchronized iterative processes, we concentrate on the efficient implementation of one iteration of any iterative algorithm with a given structure and size on an MIMD machine. In this framework, we consider the input data set for an algorithm described by Eq. 8.31 as including both the iterates x_k and the parameters defining Φ. In the case of an LSE, for example, these parameters are the system coefficients. Another cause of processing time fluctuations is the occurrence of external interrupts and page faults in the local or shared memories. To effectively isolate the performance of the algorithm on the architecture, we assume that each processor is uniprogrammed and that the memories are large enough to accommodate the address space of each process so that page faults do not occur. Moreover, external interrupts are disabled.

A loosely coupled multiprocessor system is one in which the processors access their instructions and data in their local memory. Thus, no memory access time fluctuation exists in this system. To communicate, the processors can initiate a data block transfer through their direct memory access (DMA) gate and high-speed bus (HSB) with broadcasting capability. The DMA gate has a fast communication memory (CM) which can be accessed also by the processor. To send a message to other processors, a processor stores the message in the CM, then initiates the transfer. The DMA controller of the sender monitors the bus. When it is free, a connection is established on the bus with the DMA controllers of the receivers, in time t_{Ab}. The message is transmitted on the bus and is simultaneously read by the receivers in time t_{Fb}. The total time the bus is busy for one message transfer is thus

$$t_{Cb} = t_{Ab} + t_{Fb} \tag{8.34}$$

The communication memory speeds up the transfers by reducing the overhead in each transfer. Moreover, the overlap between processing in the local memory and transfer between the communication memories is conflict free. If the CM is double buffered, the processor and the DMA controller can access it concurrently without conflicts. Under these conditions, t_{Cb} from Eq. 8.34 is deterministic for a given transfer.

To implement a synchronized algorithm on a loosely coupled system by vectorial decomposition, each of the P processors updates its subset of the iterates, then sends the values to the $(P - 1)$ other processors through the HSB. When a processor has received the updates from all the other processors, it can proceed to the next iteration.

Assume that the iterative algorithm is decomposed vectorially into P i.i.d. tasks. The P processors iterate through cycles in which they compute their subset of the iterates (processing phase), eventually communicate, and synchronize, as illustrated in Figure 8.32. The performance index is the efficiency factor E, defined as the fraction of time a processor is doing useful work. In a loosely coupled system, useful work is done during the entire processing phase only. This is not so in tightly coupled systems, where the cycles wasted in memory conflicts have to be deducted. This definition of the efficiency isolates the effect of the architecture on the performance. To compare the performance of the parallel algorithm with its corresponding sequential version, we would multiply the efficiency as defined here by a factor taking into account software restructuring or added software overhead in each iteration and by the ratio of the number of iterations required in both cases.

The techniques used to model the effect of synchronizations are drawn from order statistics. Let $T_{j:P}$ be the processing time of the jth processor to terminate (in the chronological order) when P processors are used. The estimation of the mean of $T_{j:P}$ is equivalent to the estimation of the mean of the jth order statistic among P independent samples drawn from the processing time distribution. For many distributions of interest, and for i.i.d. processes, the mean of $T_{j:P}$ is given by

$$m_{T_{j:P}} = m + 0_{j:P} \cdot \sigma \tag{8.35}$$

where m and σ are the mean and variance of the processing time, and $0_{j:P}$ is the mean of the jth order statistics among P samples drawn from the processing time distribution with mean 0 and variance 1. For example, for a *uniform* distribution,

$$0_{j:P} = \sqrt{2}\,\frac{2j - P - 1}{P + 1}$$

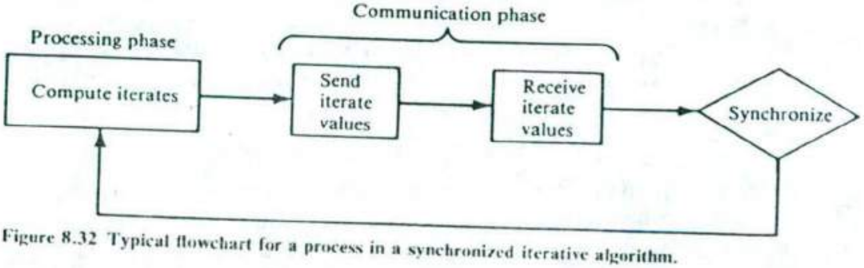


Figure 8.32 Typical flowchart for a process in a synchronized iterative algorithm.

For an *offset exponential* (an exponential plus a constant),

$$0_{j:P} = H_P - H_{P-j} - 1$$

where H_i is the ith harmonic number as defined by $\sum_{1 \le k \le i} 1/k$, for $i \ge 0$. For a *Gaussian*, no analytical formula has yet been found. The $0_{j:P}$ can, however, in this case be easily tabulated by using tables and recurrence relations.

The mean iteration time is then simply taken as the mean Pth order statistic among P independent samples:

$$m_I = m + 0_{P:P}\sigma \tag{8.36}$$

where m_I is the mean iteration time, and $0_{P:P}$ depends uniquely on the distribution of the normalized processing time.

For these examples, a Gaussian processing-time distribution has been assumed. The choice is justified by the computation model defined earlier. A computation is seen as a random number of instruction cycles, each taking a random time. By an extension to the central limit theorem, the computation-time distribution tends to a Gaussian when the average number of instructions increases, provided the fluctuations of the number of instructions are small relative to the sum of the fluctuations of each individual instruction. The Gaussian was shown to predict quite accurately the speedup of iterative algorithms on C.mmp.

We present a model for evaluating the performance of iterative algorithms on a loosely coupled system. Let us denote the two buffers of the communication memory of a processor by CM[1] and CM[2]. In each cycle of a vectorially decomposed synchronized algorithm, each processor goes through the following phases:

1. Read the iterate updates received during the previous iteration in CM[1].
2. Update a subset of the iterates.
3. Write iterate updates in CM[1].
4. Initiate transfer of the iterate updates to the $(P - 1)$ other processors.
5. Wait for reception of all other iterate updates in CM[2].
6. Switch CM[1] and CM[2] and go to 1.

The first three phases comprise the processing time.

A high-speed bus has simultaneous read-write capability and takes advantage of the randomness of the processing times according to the scheme described. The time taken by a block transfer is given by Eq. 8.34. It is assumed to be the same for all processors. For the high-speed bus, we derive a lower bound on the mean iteration time m_I for any processing-time distribution with mean order statistics given by Eq. 8.35.

The first processor in chronological order to complete its processing phase (at time $T_{1:P}$) always finds the bus free. The total iteration time is at least $I \ge T_{1:P} + Pt_{Cb}$. We also have $I \ge T_{P:P} + t_{Cb}$, since at the termination of the last processor at $T_{P:P}$, one transfer at least has yet to take place. In general, when the jth processor is finished, there remain at least $(P - j + 1)$ transfers on the bus [it is effectively $(P - j + 1)$ when the processor finds the bus free], so that

$$I = \underset{j=1,\ldots,P}{\text{Max}} \left[T_{j:P} + (P - j + 1)t_{Cb}\right] \tag{8.37}$$

Assuming equal block transfer times, Eq. 8.37 is exact and can be used in a simulation when the processing time distribution is known.

In taking the average of Eq. 8.37, we note that, in general, if $x_1, \ldots, x_n$ are n random variables:

$$E\left(\max_{1 \le i \le n} \{x_i\}\right) \ge \max_{1 \le i \le n} \{E(x_i)\}$$

where $E(\max)$ is the average of the maxima and $E(x_i)$ is the average of x_i. According to this result, we write

$$m_I \ge \underset{j=1,\ldots,P}{\text{Max}} \left[m_0 + 0_{j:P} \cdot \sigma_0 + (P - j + 1)t_{Cb}\right] \tag{8.38}$$

Taking the lower bound for m_I, we can derive the efficiency as:

$$E = \frac{m_0}{m_I} \simeq \frac{m_0}{\underset{j=1,\ldots,P}{\text{Max}} \left[m_0 + 0_{j:P}\sigma_0 + (P - j + 1)t_{Cb}\right]} = \frac{1}{1 + \Delta} \tag{8.39}$$

with

$$\Delta(P, C_0, t'_{Cb}) \simeq \underset{j=1,\ldots,P}{\text{Max}} \left[0_{j:P} \cdot C_0 + (P - j + 1) \cdot t'_{Cb}\right]$$

The set of performance features are P, C_0, $t'_{Cb} = t_{Cb}/m_0$ and $C_0 = \sigma_0/m_0$.

The efficiency factors for the loosely coupled system are displayed in Figure 8.33 as a function of the features. From these figures, it is clear that the high sensitivity of the performance index to the communication to computation times ratio t'_{Cb} limits the applicability of loosely coupled architectures for synchronized iterative algorithms. Such architectures are well matched for algorithms with a low communication to computation times ratio. However, when this ratio increases beyond a few percent, the efficiency decreases rapidly. This property reflects the power of a high-speed bus with broadcasting capability.

The above methodology can be used to study the degree of matching between an algorithm and a multiprocessor architecture. This methodology is based on extracting performance features for a class of algorithms and an architecture from an approximate analytical model. The features define a multidimensional space. A performance index is then a mapping from the feature space on the real line. Given the architecture, algorithms pertaining to the class defined by the hypothesis of the model can be partially ordered according to the value of the performance index. This ordering allows us to locate regions in the feature space where the architecture is well matched to algorithms in the class.

The difficulty of this approach is in striking a proper balance between the simplicity and tractability of the analytical model and its accuracy. As modeling

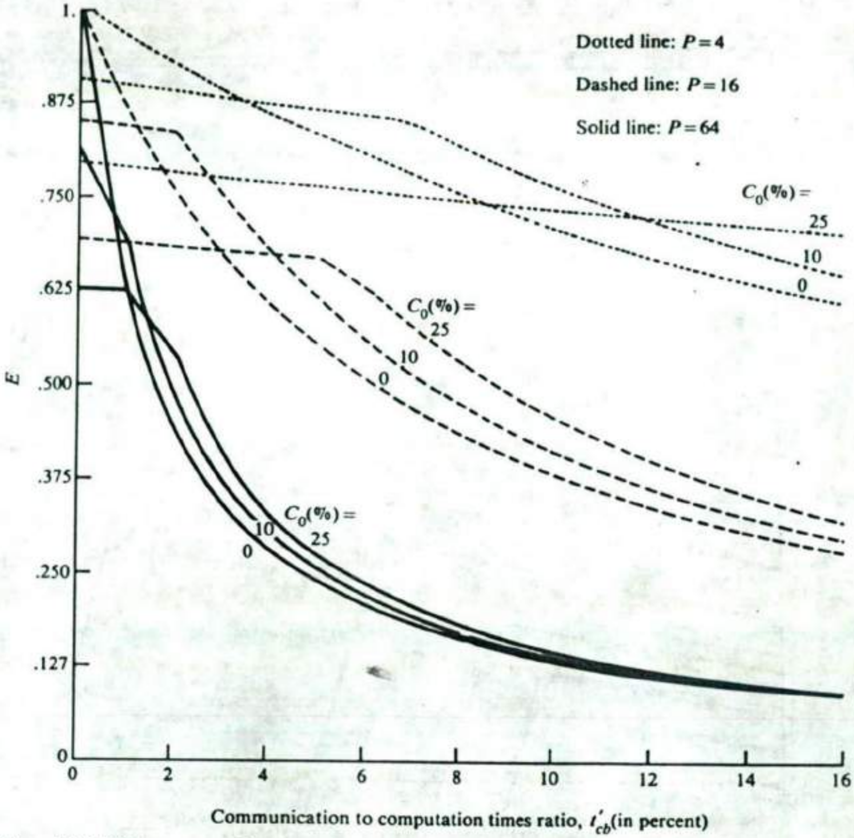


Figure 8.33 Efficiency versus feature t_{cb} for a loosely coupled system with a high speed. (Courtesy *IEEE Trans. Software Engg.*, Dubois and Briggs, July 1982.)

tools improve, the analytical model may be refined. Even if approximate, the feature space approach is much more realistic than complexity studies.

In Figure 8.34, cuts through the feature space are displayed. These cuts are two planes for each case. The index function E is represented by contours of equal index value. Loosely coupled systems are effective for processing-intensive computations (low values of feature t'_{Cb}). The regions with a high-efficiency factor shrink as the number of processes increases. Visualizing the feature space by cuts such as in Figure 8.34 is of great help in understanding the interaction between the architecture and the class of algorithms. Other architectures could be studied using the methodologies discussed.

The estimation of E is very important to the software designer for MIMD systems, since it is the proportionality factor between m_0 and m_I, the average iteration time (see Eq. 8.38 for example). As a result of the analysis, a given imple-

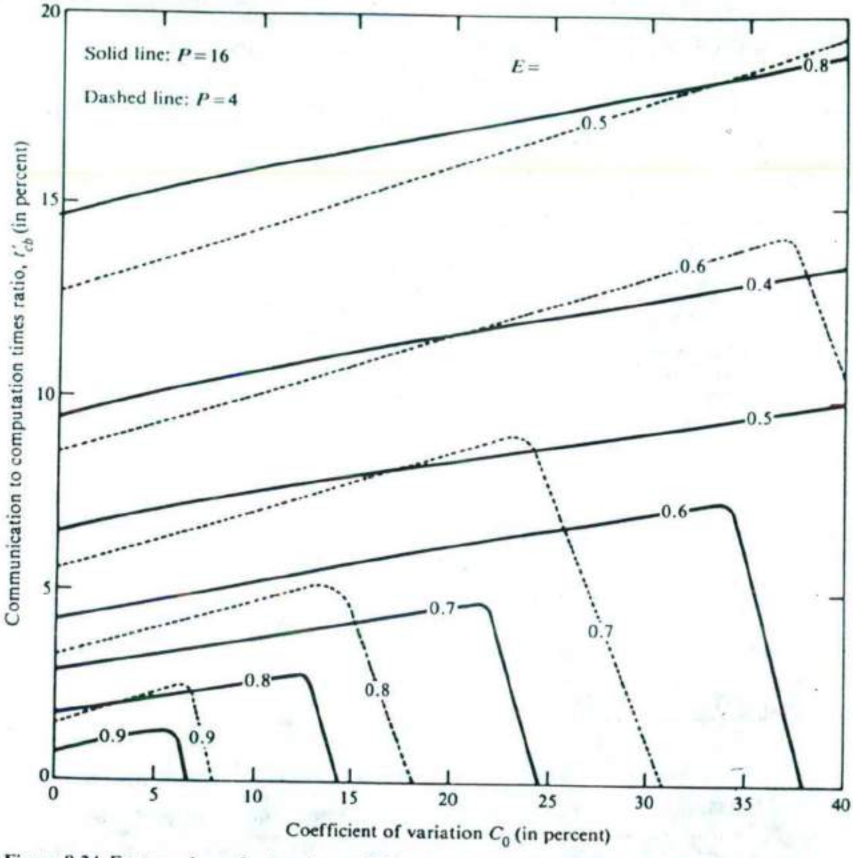


Figure 8.34 Feature planes for loosely coupled system with high-speed bus. (Courtesy of *IEEE Trans. Software Engg.*, Dubois and Briggs, July 1982.)

mentation may be revealed as inefficient for the architecture and may have to be restructured.

Effect of decomposition of performance Another interesting problem is to estimate the effect of the degree of decomposition of a given algorithm on the speedup in order to evaluate the optimum decomposition. For such a study, the assumptions on the computation to partition must be stronger. We use the equations derived in the previous section on the performance of synchronized algorithms.

We define a homogeneous computation as follows. Let SUM be a computation. SUM is seen as a random number of instruction cycles. If SUM can be partitioned into computation units with the same stochastic properties ("i.i.d units"), then it is said to be homogeneous with respect to the unit. Moreover, we assume a Gaussian distribution for the time taken by a computation unit for the same reasons as mentioned previously. This property is important so that the underlying processing-time distribution is preserved for all partitions of the computation. To obtain a decomposition of a homogeneous computation in i.i.d. tasks, we partition the computation into sets containing the same number of units. Each set defines a process. The number of such processes in the partition is the *degree of decomposition*, denoted by P since one processor is devoted to each process. The *maximum degree of decomposition* (PMAX) is obtained when each process contains only one computation unit.

A simple model for the mean and variance of the number of active cycles in a process of a homogeneous computation as a function of the decomposition is

$$m_0 = m_A + \frac{m_B}{P} \qquad \text{and} \qquad \sigma_0^2 = \sigma_A^2 + \frac{\sigma_B^2}{P} \tag{8.40}$$

m_A and σ_A^2 are the mean and variance of the fixed overhead independent of the decomposition, while m_B and σ_B^2 correspond to the mean and variance of the partitionable part of the computation.

As the degree of decomposition increases, the number of iterates to be computed by each process reduces proportionally (accounting for the second term in Eq. 8.40). The overhead term might include the initiation of a transfer through the high-speed bus. In most implementations, making a private copy of the iterate set will be required at the beginning of each iteration and should be accounted for in m_A and σ_A^2.

For the loosely coupled system, the block transfer time is modeled by

$$t_{Cb} = t_{Ab} + \frac{t_{Bb}}{P} \tag{8.41}$$

As in Eq. 8.35, the transfer time is deterministic and assumed identical for all the processes. The first term of Eq. 8.41 represents the transfer overhead (time between the reservation of the bus or memory module and the actual beginning of a transfer) plus the time taken possibly by the transfer of a fixed amount of information independent of the decomposition. Since the number of iterates computed by each process decreases proportionally to the decomposition, the transfer time decreases accordingly. Equation 8.41 means that the time to initiate a transfer t_{Ab} and the speed of the transfer are independent of the decomposition, an assumption not always verified for the bus system but nonetheless simple and realistic in most cases. The speedup, denoted by S_P, is defined as the ratio of the times taken by the algorithm on a uniprocessor and on a multiprocessor system when the degree of decomposition is P (with maximum PMAX). It is computed as follows for the case when $m_A \simeq \sigma_A^2 \simeq 0$.

For the loosely coupled architecture with a high-speed bus, we have

$$S_P \simeq \frac{P}{1 + \Delta_P} \tag{8.42}$$

with

$$\Delta_P = \max_{j=1,\ldots,P} \left[0_{j:P} \cdot C_B \cdot \sqrt{P} + (P - j + 1) \cdot (P \cdot t'_{Ab} + t'_{Bb}) \right]$$

$$t'_{Ab} = \frac{t_{Ab}}{m_B}, \; t'_{Bb} = \frac{t_{Bb}}{m_B} \quad \text{and} \quad C_B = \frac{\sigma_B}{m_B}$$

Note that, contrary to the general study discussed earlier, the parameters are normalized in this section with respect to the total computation m_B.

The speedup curves are displayed in Figure 8.35. C_B (the coefficient of variation of the total computation) is shown to limit the optimum decomposition in most cases. Loosely coupled systems have a very good speedup when t'_{Bb} is small (0.1 percent) and $t'_A \simeq 0$. However, a nonnull constant term (Figure 8.36) in each

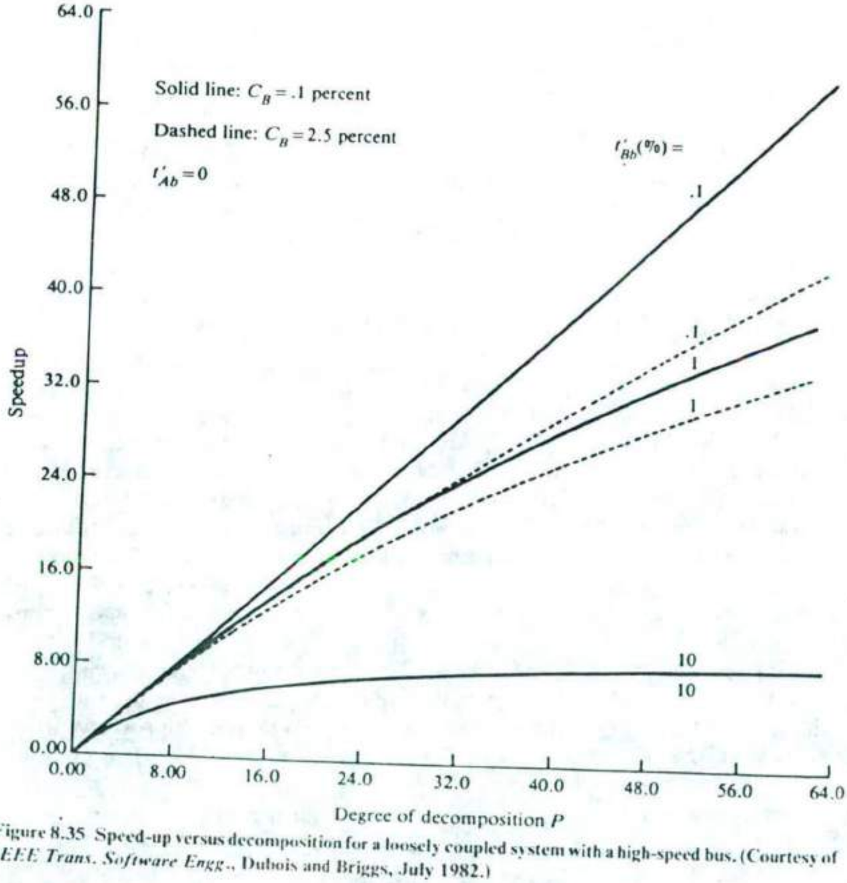


Figure 8.35 Speed-up versus decomposition for a loosely coupled system with a high-speed bus. (Courtesy of *IEEE Trans. Software Engg.*, Dubois and Briggs, July 1982.)

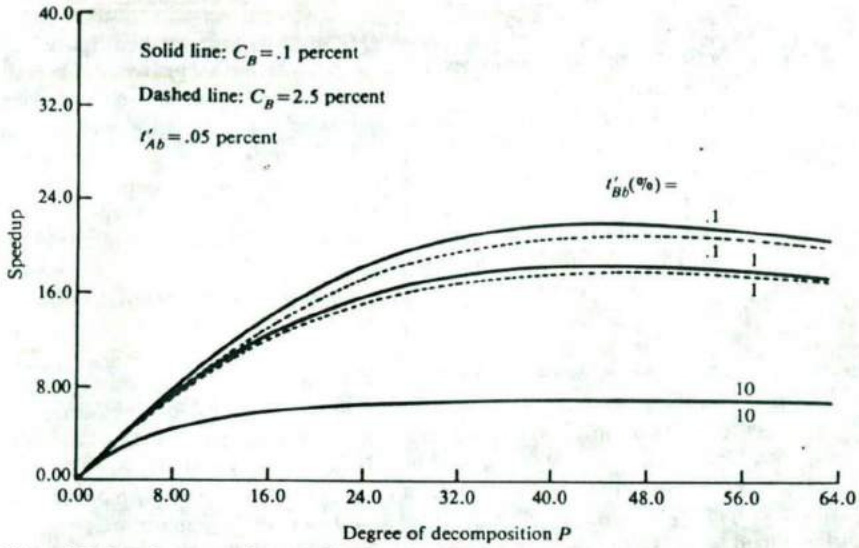


Figure 8.36 Speed-up versus decomposition for a loosely coupled system with a high-speed bus. (Courtesy of *IEEE Trans. Software Engg.*, Dubois and Briggs, July 1982.)

transfer causes the speedup to peak. In this latter case, the optimum decomposition is limited to a degree of 16 to 40 for the examples considered.

The speedup conceivably peaks out or saturates when the decomposition increases. This intuitive reasoning is confirmed quantitatively by the curves of Figure 8.34. These results can be used as a guideline by the compiler or the user for an effective decomposition of an MIMD iterative problem into tasks of similar statistical properties.

8.5 BIBLIOGRAPHIC NOTES AND PROBLEMS

Parallel algorithms for multiprocessors are analyzed in Kung (1976), including both synchronized and asynchronous algorithms. Other discussions including macropipelines can be found in Jones and Schwarz (1979). Baudet (1976) studied the performance of asynchronous parallel algorithms. Mathematical analysis of simple task graphs was developed in Robinson (1979). Iterative techniques for solving linear systems of equations are given in Conrad and Wallach (1977). Numerical methods are given in Dahlquist and Björch (1974). Performance of quicksort and PDE algorithms on Cm* is given in Deminet (1982). The performance of synchronized iterative algorithms was presented in Dubois and Briggs (1982a). Parallel tree search algorithms for multiprocessors can be found in Hwang and Yao (1977).

Some good sources of operating system design concepts are given in Coffman and Denning (1973), Habermann (1976), and Holt et al. (1978). A general overview of protection of information in computer systems was presented in Saltzer and Schroeder (1975). One of the earlier sources of information on protection mechanisms was given in Lampson (1974). A basic source of capability-based addressing schemes is in Fabry (1974). Application of capability-based addressing schemes to existing machines is presented in Lampson and Sturgis (1976) and Levy (1983). Another example of the implementation of protection schemes is in the Multics project Saltzer (1974). A complete overview of the deadlock problem is presented in Isloor and Marsland (1980). Some specific research in the solution to the deadlock problem are given in Coffman et al. (1971) and Holt (1972). Another comprehensive treatment of prevention, detection, and recovery of system deadlocks is given in Shoshani and Coffman (1970). An extension of such techniques to multiprocessor systems is given in Fontao (1971).

A general overview of process synchronization mechanisms is given in Madnick and Donovan (1974). The concepts of wakeup and block primitives are discussed in Habermann (1976). The wait and signal primitives are elaborated on in Stone (1980). The concept of pipes in the UNIX operating system can be found in Ritchie (1973). An application of synchronization primitives to the implementation of such pipes is given in Holt et al. (1978). An extension of the semaphore primitives developed by Dijkstra (1968) is presented in Agerwala (1977). The implementation of conditional critical regions and the construction of monitors are given in Schmid (1976). A general treatment of monitors is covered in Hoare (1974). Conditional critical regions were proposed by Hoare (1972) and Hansen (1972). Recently, concurrent programming was studied in Andrews and Schneider (1983).

A general overview of deterministic schedules is presented in Gonzalez (1977). Preemptive deterministic schedules have been researched in Muutz and Coffman (1969). Some nonpreemptive schedules are discussed in Coffman and Graham (1972) and Hu (1961). The multiprocessor anomalies are demonstrated in Graham (1972). The processor bounds are developed in Ramamoorthy et al. (1972) and Liu and Liu (1974). Probabilistic scheduling algorithms are presented in Coffman and Denning (1973). Load balancing in multiple processors is also studied in Ni and Hwang (1983). Traditional books on queueing theory such as Kleinrock (1975) are sources of information on probabilistic schedules. The group scheduling concept was discussed in Jones and Schwarz (1979).

Problems

8.1 Describe the following terminologies associated with multiprocessor operating systems and MIMD algorithms:

(*a*) Mutual exclusion
(*b*) The TEST-AND-SET instruction
(*c*) The ENQUEUE and DEQUEUE operations
(*d*) The **P** and **V** operators
(*e*) Conditional critical sections
(*f*) Deadlock prevention and avoidance
(*g*) Deadlock detection and recovery

(*h*) Protection versus security mechanisms
(*i*) Deterministic versus stochastic scheduling
(*j*) Multiprocessor anomalies
(*k*) Synchronized versus asynchronous parallel algorithms
(*l*) Degree of decomposition of a parallel algorithm

8.2 The following is a proposal to handle the "empty stack problem" of a stack that is used to structure a shared storage (or page) pool. GETSPACE is a procedure which returns a pointer to a free page. Before calling GETSPACE, a process should enter a critical section for stack inspection, for which purpose a global (shared variable) count STACKLENGTH records the current number of free pages. A process should not call GETSPACE until STACKLENGTH > 0. Similarly, there is a RELEASE (*i*) procedure, which releases page *i* to the page pool. Write two procedures: FREEPAGE to get and return a pointer to a free page, and RETURN-PAGE (*i*) to release page *i* using P and V primitives. (*Hint*: A binary semaphore mutex should be used to control access to the stack. Mutex is initialized to 1. Also use "STACKLENGTH" as a semaphore to control access to the variable STACKLENGTH for update).

8.3 Consider the following program:

```
For i = 1 to n do
   cobegin
      S_1[i]; S_2[i];...S [i];
coend
```

k processors are devoted to the above computation. Each computation $S_j[i]$, for $1 \le j \le k$ takes an unpredictable and random time to execute. Complete the program between the **begin** and **end** statements for process *j* using critical section (csect) on the shared variable, pcount, and P and V operations on the binary semaphores *s*. The primitives implement the synchronization of these *k* processors. *Hint*: The last processor to finish one iteration for one value of *i* updates pcount and "awakens" the other blocked processors through *s*.

```
var pcount: shared; initial pcount = k - 1
var s = {s_1, . . . ,s } : binary semaphore; initial s = {1, 1, . . . ,1};
{Process 1}:
   :
{Process j}:
  For ← 1 to n do
     begin
        . . .
        ⟨S_j(i)⟩
        . . .
     end
{Process j + 1};
   :
```

8.4 Assume that f_1 and f_2 are pointers to two sorted lists, each arranged as a circular doubly linked list with headnodes f_1 and f_2, respectively. f_1 and f_2 are sorted in ascending order and each node has three fields, namely, LLINK (left link), DATA, and RLINK (right link) (Figure 8.37). Write an asynchronous MIMD algorithm for two processes P_1 and P_2 to merge the files f_1 and f_2 into a sorted list f, also arranged as a circular list. Process P_1 retrieves from the fronts of f_1 and f_2 to produce a sorted sublist d_1, while P_2 retrieves from the rears of f_1 and f_2 to produce the sorted sublist d_2 until either f_1 or f_2 becomes empty. Attach the nonempty list to d_1 and d_2 to produce f. Return any unused node x to the storage pool with the operation POOL (x). Note that sublists d_1 and d_2 are initially empty.

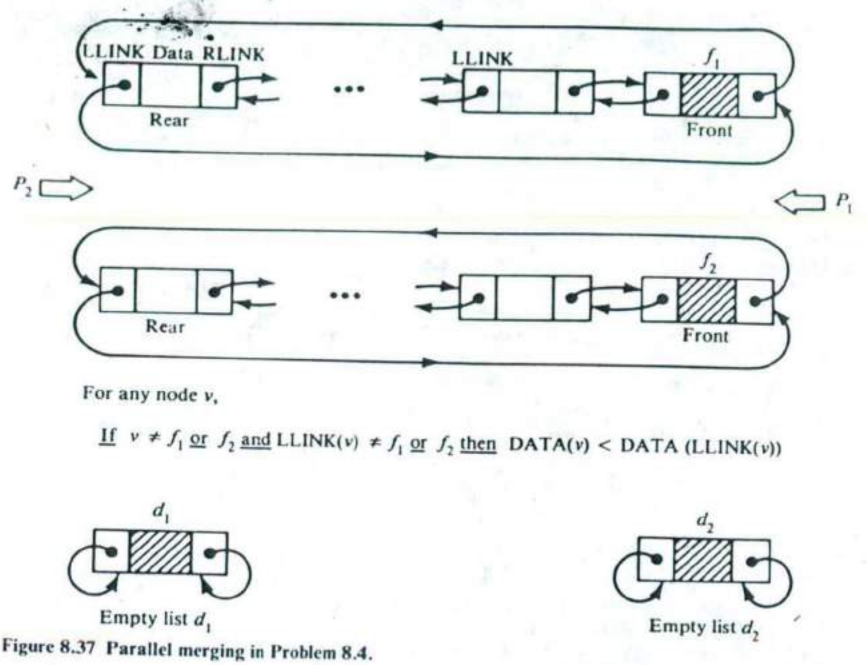


Figure 8.37 Parallel merging in Problem 8.4.

8.5 The following is a synchronized n-process MIMD algorithm to compute vector:

```
   y = A · x + b, where y is n × 1, A is n × n, b is n × 1
PROCESS:
{Parfor i ← until n − 1 do
   {y (i) ← 0;
      For j ← 0 until n − 1 do
         (i) ← y (i) + A (i, j) · x(j);
      y (i) ← y (i) + b (i);
   }
}
```

Assume that each assignment of the variable i in the **Parfor** statement takes c seconds. The time it takes to start or spawn a new process for a given i is a sum of independent random variables $\sum_{j=0}^{i} x_j - ic$ where x_j is exponentially distributed with mean time $1/(j+1)\lambda$. What is the speedup S_n of the parallel process if multiplication and addition takes t_m and t_a seconds, respectively, on each processor and n concurrent processes are used? Also assume that $nc < 1/\lambda$. Then plot S_n versus n for values of $1/\lambda T =$ 0.1, 0.5, 0.8, 1, 2, 5, and 10.

8.6 Dijkstra's problem of dining philosophers (slightly generalized) is: There are n philosophers whose lives consist of alternately thinking and eating. The philosophers eat at a large circular table with a preassigned plate for each. Between two plates is a fork, which may be used by either adjacent philosopher. In order to eat, a philosopher must have two forks (one on his left and the other one on his right). Devise a control program of the general form: Note that the *n* philosophers may not necessarily follow the same control program to claim and release the forks.

```
Philosopher (i): begin
                 Think;
                 ...(a)...
                 Eat;
                 ...(b)...
                 end
```

where (a) and (b) are code sections which claim and release the two forks, respectively. You should specify these two code sections such that the following properties are met:

(*a*) Use P-V for communication and synchronization.

(*b*) Allow a fork to be held by only one philosopher at a time.

(*c*) Use strictly local information, e.g., LEFT.FORK and RIGHT.FORK to indicate the two forks (resources) at both sides of each philosopher (process).

(*d*) Guarantee that no philosopher will starve, that is when the *n* processes enter a deadlock situation.

8.7 Solve the dining philosophers problem using (*a*) generalized *P* and *V*, (*b*) conditional critical regions.

8.8 Using *P-V* operations, write the synchronizing program for the task graph shown in Figure 8.38, where T_i' is the statement that controls the execution of task T_i. Solution should be of the following form:

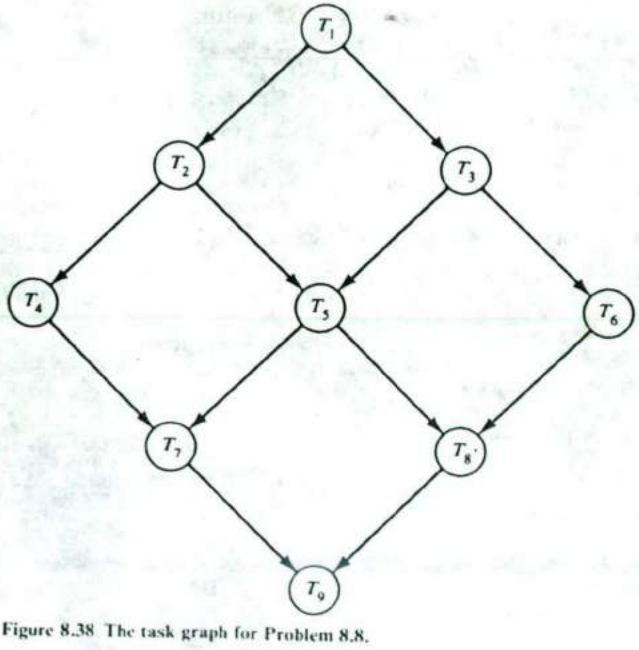


Figure 8.38 The task graph for Problem 8.8.

```
Define semaphore;
Initialization of semaphores;
cobegin
   $T'_1$:
   $T'_2$:
    ⋮
   $T'_7$:
   $T'_8$:
   $T'_9$:
coend
```

8.9 (*a*) Which combination of processes cause deadlock in the five process code segment programmed below? The processes are A, B, C, D, and E.

```
Begin
    shared record
      begin
        var $S_1$, $S_2$, $S_3$, $S_4$: semaphore;
        var blocked, unblocked: integer;
      end
    initial blocked = 0, unblocked = 1;
      initial $S_1 = S_2 = S_3$ = unblocked, $S_4$ = blocked;
  cobegin
    A: begin P($S_1$); V($S_1$); P($S_2$); V($S_2$) end;
    B: begin P($S_1$), P($S_2$); V($S_4$); V($S_2$); V($S_1$) end;
    C: begin P($S_2$); P($S_3$); V($S_2$); V($S_3$) end
    D: begin P($S_4$); P($S_2$); P($S_1$); V($S_1$); V($S_2$) end
    E: begin P($S_3$); P($S_2$); V($S_2$); V($S_3$) end
  coend
End
```

(*b*) Besides the combination of processes mentioned in your answer to part (*a*), which additional process(es) could be indefinitely blocked because of this deadlock?

(*c*) For the processes given, is deadlock inevitable, or does it depend on race conditions? Justify your answer.

(*d*) Assume that the skeletal code segment programmed above is an abbreviated version of a more complex program and that only the details of the semaphore-related code is shown here. Guarantee that all five processes in the real program complete by making a minor change to one of the skeletal processes.

8.10 Write a parallel algorithm to implement the concurrent Quicksort algorithm described on pages 625–626.

8.11 Show that the use of PE and VE could result in a deadlock or possibly starvation in a system of resources.