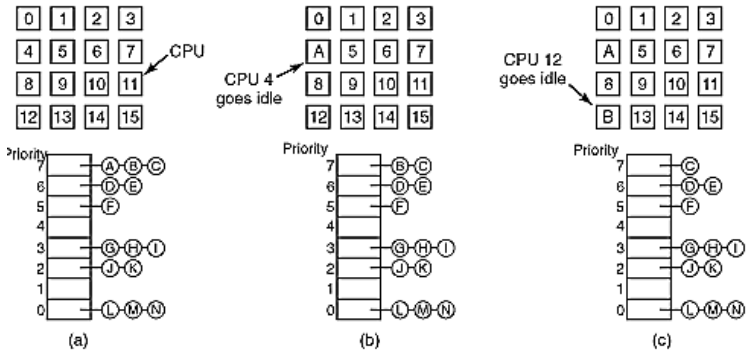
Multiprocessor Scheduling:-

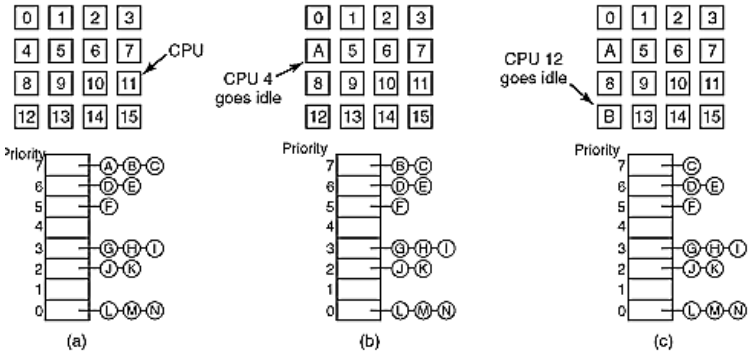
- On a uniprocessor, scheduling is one dimensional. The only question that must be answered (repeatedly) is: ''Which process should be run next?'' On a multiprocessor, scheduling is two dimensional. The scheduler has to decide which process to run and which CPU to run it on. This extra dimension greatly complicates scheduling on multiprocessors.
- Another complicating factor is that in some systems, all the processes are unrelated whereas in others they come in groups. An example of the former situation is a timesharing system in which independent users start up independent processes. The processes are unrelated and each one can be scheduled without regard to the other ones

Timesharing

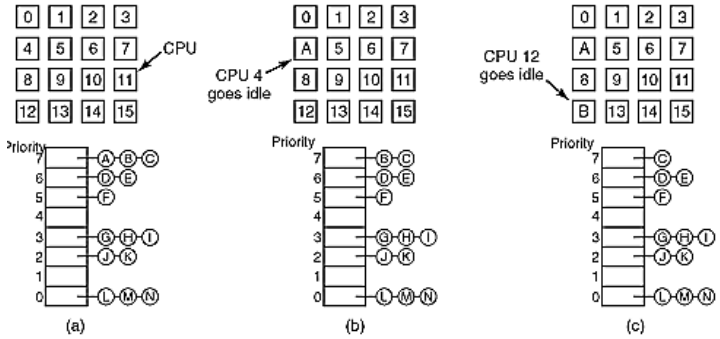
- The simplest scheduling algorithm for dealing with unrelated processes (or threads) is to have a single system wide data structure for ready processes, possibly just a list, but more likely a set of lists for processes at different priorities as depicted



- Here the 16 CPUs are all currently busy, and a prioritized set of 14 processes are waiting to run. The first CPU to finish its current work (or have its process block) is CPU 4, which then locks the scheduling queues and selects the highest priority process, *A*



- Next, CPU 12 goes idle and chooses process *B*, as illustrated in Fig. As long as the processes are completely unrelated, doing scheduling this way is a reasonable choice.



- Having a single scheduling data structure used by all CPUs timeshares the CPUs, much as they would be in a uniprocessor system. It also provides automatic load balancing because it can never happen that one CPU is idle while others are overloaded. Two disadvantages of this approach are the potential contention for the scheduling data structure as the numbers of CPUs grows and the usual overhead in doing a context switch when a process blocks for I/O.

- It is also possible that a context switch happens when a process' quantum expires.

Space Sharing

- The other general approach to multiprocessor scheduling can be used when threads are related to one another in some way.

- Scheduling multiple threads at the same time across multiple CPUs is called **space sharing**.

- At any instant of time, the set of CPUs is statically partitioned into some number of partitions, each one running the threads of one process. In Fig , we have partitions of sizes 4, 6, 8, and 12 CPUs, with 2 CPUs unassigned, for example. As time goes on, the number and size of the partitions will change as new threads are created and old ones finish and terminate.

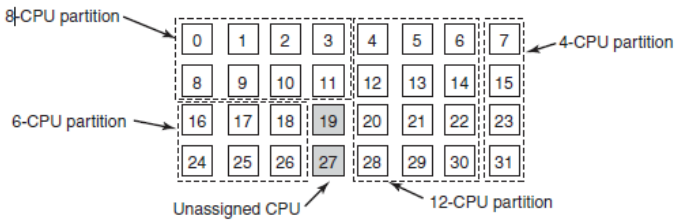


Figure 8-13. A set of 32 CPUs split into four partitions, with two CPUs available.

Gang Scheduling:-

- A clear advantage of space sharing is the elimination of multiprogramming, which eliminates the context-switching overhead. However, an equally clear disadvantage is the time wasted when a CPU blocks and has nothing at all to do until it becomes ready again. Consequently, people have looked for algorithms that attempt to schedule in both time and space together, especially for threads that create multiple threads, which usually need to communicate with one another.
- To see the kind of problem that can occur when the threads of a process are independently scheduled, consider a system with threads *A*0 and *A*1 belonging to process *A* and threads *B*0 and *B*1 belonging to process *B*. Threads *A*0 and *B*0 are timeshared on CPU 0; threads *A*1 and *B*1 are timeshared on CPU 1. Threads *A*0 and *A*1 need to communicate often. The communication pattern is that *A*0 sends *A*1 a message, with *A*1 then sending back a reply to *A*0, followed by another such sequence, common in client-server situations.

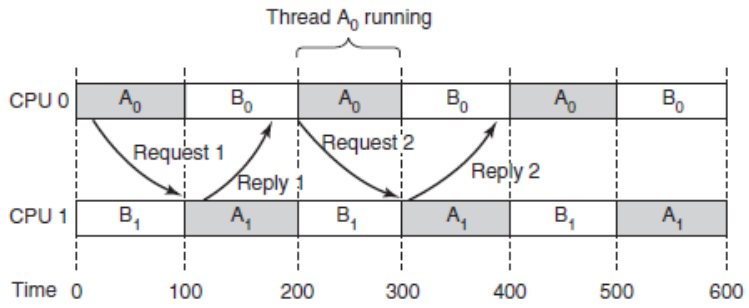


Figure 8-14. Communication between two threads belonging to thread *A* that are running out of phase.

The solution to this problem is **gang scheduling**, which is an outgrowth of **coscheduling** (Ousterhout, 1982). Gang scheduling has three parts:

- 1. Groups of related threads are scheduled as a unit, a gang.
- 2. All members of a gang run at once on different timeshared CPUs.
- 3. All gang members start and end their time slices together.

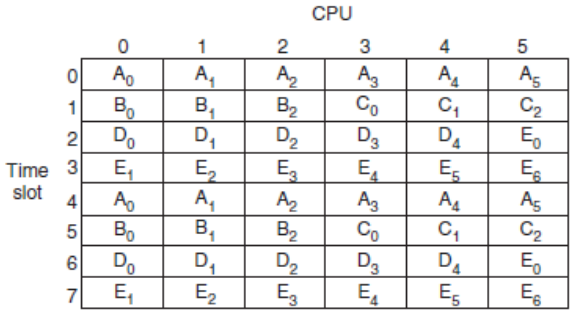


Figure 8-15. Gang scheduling.

MULTICOMPUTERS

- Multiprocessors are popular and attractive because they offer a simple communication model: all CPUs share a common memory. Processes can write messages to memory that can then be read by other processes. Synchronization can be done using mutexes, semaphores, monitors, and other well-established techniques.
- To get around these problems, much research has been done on **multicomputer**, which are tightly coupled CPUs that do not share memory. Each one has its own memory.
- These systems are also known by a variety of other names, including **cluster computers** and **COWS** (**Clusters Of Workstations**).
- Cloud computing services are always built on multicomputer because they need to be large.
- Multicomputer are easy to build because the basic component is just a stripped-down PC, without a keyboard, mouse, or monitor, but with a high-performance network interface card.

Multicomputer Hardware:-

- The basic node of a multicomputer consists of a CPU, memory, a network interface, and sometimes a hard disk. The node may be packaged in a standard PC case, but the monitor, keyboard, and mouse are nearly always absent. Sometimes this configuration is called a **headless workstation** because there is no user with a head in front of it. A workstation with a human user should logically be called a ‘‘headed workstation,’’ but for some reason it is not.

Interconnection Technology:-

Each node has a network interface card with one or two cables (or fibers) coming out of it. These cables connect either to other nodes or to switches. In a small system, there may be one switch to which all the nodes are connected in the star topology of Fig. Modern switched Ethernets use this topology.

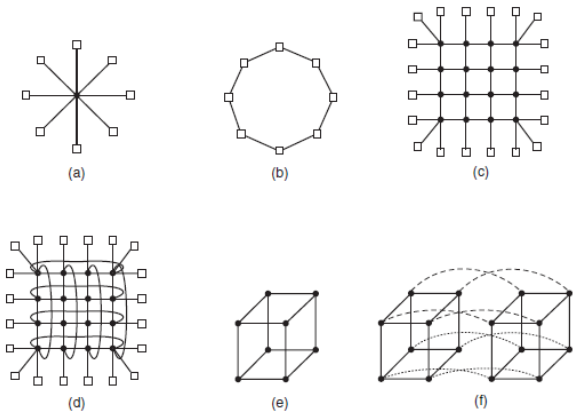


Figure 8-16. Various interconnect topologies. (a) A single switch. (b) A ring. (c) A grid. (d) A double torus. (e) A cube. (f) A 4D hypercube.

- As an alternative to the single-switch design, the nodes may form a ring, with two wires coming out the network interface card, one into the node on the left and one going into the node on the right, as shown in Fig. In this topology, no switches are needed and none are shown.
- The **grid** or **mesh** is a two-dimensional design that has been used in many commercial systems. It is highly regular and easy to scale up to large sizes. It has a **diameter**, which is the longest path between any two nodes, and which increases only as the square root of the number of nodes. A variant on the grid is the **double torus** of Fig. which is a grid with the edges connected.
- Not only is it more fault tolerant than the grid, but the diameter is also less because the opposite corners can now communicate in only two hops.
- The **cube** of Fig. is a regular three-dimensional topology. We have illustrated a 2 × 2 × 2 cube, but in the most general case it could be a $k \times k \times k$ cube.
- To go to six dimensions, we could replicate the block of four cubes and interconnect the corresponding nodes, and so on. An n-dimensional cube formed this way is called a **hypercube**.

Network Interfaces:-

- All the nodes in a multicomputer have a plug-in board containing the node's connection to the interconnection network that holds the multicomputer together.
- The way these boards are built and how they connect to the main CPU and RAM have substantial implications for the operating system.
- In virtually all multicomputers, the interface board contains substantial RAM for holding outgoing and incoming packets. Usually, an outgoing packet has to be copied to the interface board's RAM before it can be transmitted to the first switch.
- The reason for this design is that many interconnection networks are synchronous, so that once a packet transmission has started, the bits must continue flowing at a constant rate. If the packet is in the main RAM, this continuous flow out onto the network cannot be guaranteed due to other traffic on the memory bus. Using a dedicated RAM on the interface board eliminates this problem. This design is shown in Fig.

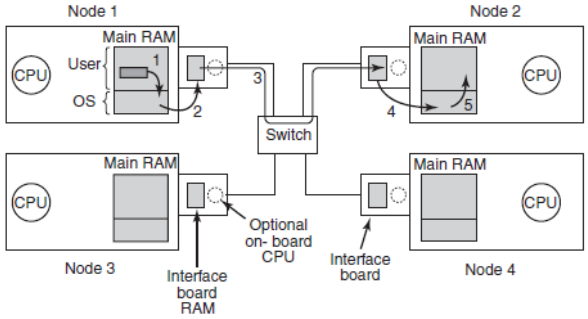


Figure 8-18. Position of the network interface boards in a multicomputer.

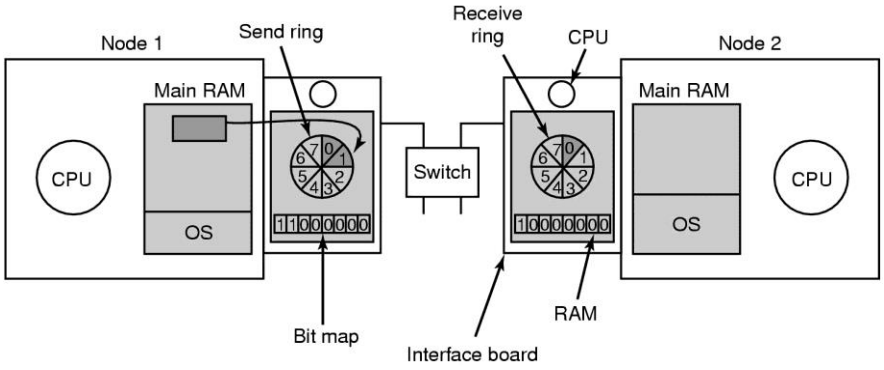
- The same problem occurs with incoming packets. The bits arrive from the network at a constant and often extremely high rate. If the network interface board cannot store them in real time as they arrive, data will be lost.

Low-Level Communication Software:-

- The enemy of high-performance communication in multicomputer systems is excess copying of packets. In the best case, there will be one copy from RAM to the interface board at the source node, one copy from the source interface board to the destination interface board (if no storing and forwarding along the path occurs), and one copy from there to the destination RAM, a total of three copies. However, in many systems it is even worse. In particular, if the interface board is mapped into kernel virtual address space and not user virtual address space, a user processcan send a packet only by issuing a system call that traps to the kernel. The kernels may have to copy the packets to their own memory both on output and on input, for example, to avoid page faults

while transmitting over the network. Also, the receiving kernel probably does not know where to put incoming packets until it has had a chance to examine them.

- While this approach definitely helps performance, it introduces two problems. First, what if several processes are running on the node and need network access to send packets? Which one gets the interface board in its address space? Having a system call to map the board in and out of a virtual address space is expensive, but if only one process gets the board, how do the other ones send packets? And what happens if the board is mapped into process A's virtual address space and a packet arrives for process B, especially if A and B have different owners, neither of whom wants to put in any effort to help the other? The second problem is that the kernel may well need access to the interconnection network itself, for example, to access the file system on a remote node.



Node-to-Network Interface Communication:-

- Another issue is how to get packets onto the interface board. The fastest way is to use the DMA chip on the board to just copy them in from RAM. The problem with this approach is that DMA may use physical rather than virtual addresses and runs independently of the CPU, unless an I/O MMU is present. To start with, although a user process certainly knows the virtual address of any packet it wants to send, it generally does not know the physical address. Making a system call to do the virtual-to-physical mapping is undesirable, since the point of putting the interface board in user space in the first place was to avoid having to make a system call for each packet to be sent.

Remote Direct Memory Access:-

- Remote Direct Memory Access (RDMA) is a technology that allows computers in a network to exchange data in main memory without involving the processor, cache, or operating system of either computer. Like locally-based Direct Memory Access (DMA), RDMA improves throughput and performance because it frees up resources. RDMA also facilitates a faster data transfer rate. RDMA implements a transport protocol in the network interface card (NIC) hardware and supports a feature called zero-copy networking. Zero-copy networking makes it possible to read data directly from the main

memory of one computer and write that data directly to the main memory of the other computer. RDMA has proven useful in applications that involve high-speed clusters and data center networks.

User-Level Communication Software:-

- Processes on different CPUs on a multicomputer communicate by sending messages to one another. In the simplest form, this message passing is exposed to the user processes. In other words, the operating system provides a way to send and receive messages, and library procedures make these underlying calls available to user processes. In a more sophisticated form, the actual message passing is hidden from users by making remote communication look like a procedure call.

Send and Receive:-

- At the barest minimum, the communication services provided can be reduced to two (library) calls, one for sending messages and one for receiving them. The call for sending a message might be

send(dest, &mptr);

and the call for receiving a message might be

receive(addr, &mptr);

- The former sends the message pointed to by mptr to a process identified by dest and causes the called to be blocked until the message has been sent. The latter causes the called to be blocked until a message arrives. When one does, the message is copied to the buffer pointed to by mptr and the called is unblocked. The addr parameter specifies the address to which the receiver is listening. Many variants of these two procedures and their parameters are possible.

Blocking versus Nonblocking Calls:-

- The calls described above are blocking calls (sometimes called synchronous calls). When a process calls send, it specifies a destination and a buffer to send to that destination. While the message is being sent, the sending process is blocked (i.e., suspended). The instruction following the call to send is not executed until the message has been completely sent, as shown in Fig. Similarly, a call to receive does not return control until a message has actually been received and put in the message buffer pointed to by the parameter. The process remains suspended in receive until a message arrives, even if it takes hours. In some systems, the receiver can specify from whom it wishes to receive, in which case it remains blocked until a message from that sender arrives.

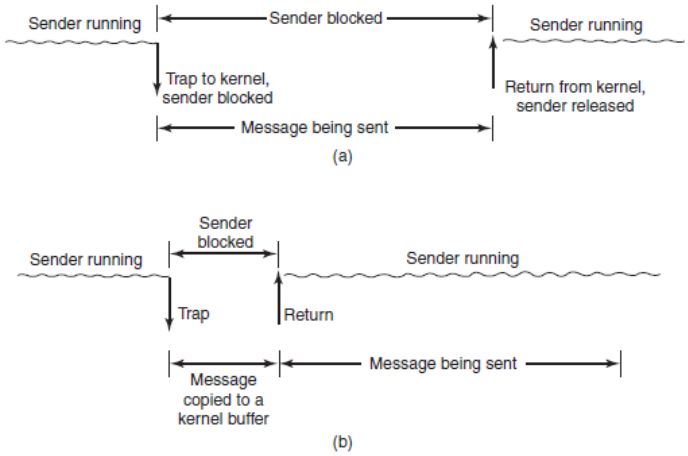


Figure 8-19. (a) A blocking send call. (b) A nonblocking send call.

- An alternative to blocking calls is the use of nonblocking calls (sometimes called asynchronous calls). If send is nonblocking, it returns control to the called immediately, before the message is sent. The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle (assuming no other process is runnable).

Remote Procedure Call:-

- Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. (A procedure call is also sometimes known as a function call or a subroutine call.) RPC uses the client/server model. The requesting program is a client and the service-providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned. However, the use of lightweight processes or threads that share the same address space allows multiple RPCs to be performed concurrently.
- When program statements that use RPC are compiled into an executable program, a stub is included in the compiled code that acts as the representative of the remote procedure code. When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client runtime program in the local computer. The client runtime program has the knowledge of how to address the remote computer and server application and sends the message across the network that requests the remote procedure. Similarly, the server includes a runtime program and stub that interface with the remote procedure itself. Results are returned the same way.
- The actual steps in making an RPC are shown in Fig. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making

a system call to send the message. Packing the parameters is called marshalling. Step 3 is the kernel sending the message from the client machine to the server machine. Step 4 is the kernel passing the incoming packet to the server stub (which would normally have called receive earlier). Finally, step 5 is the server stub calling the server procedure. The reply traces the same path in the other direction.

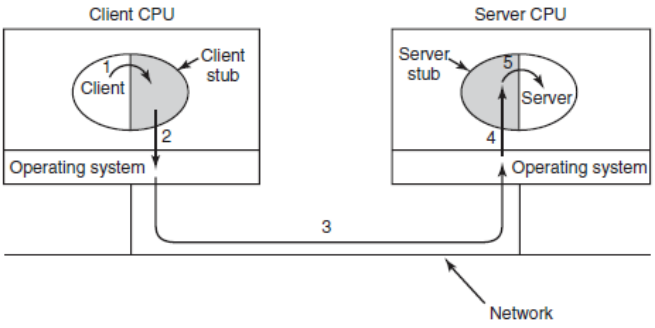


Figure 8-20. Steps in making a remote procedure call. The stubs are shaded gray.

Implementation Issues:-

- Cannot pass pointers
  - call by reference becomes copy-restore (but might fail)
- Weakly typed languages
  - client stub cannot determine size
- Not always possible to determine parameter types
- Cannot use global variables
  - may get moved to remote machine

Distributed Shared Memory:-

Distributed shared memory (DSM) is a form of memory architecture where the (physically separate) memories can be addressed as one (logically shared) address space. Here, the term "shared" does not mean that there is a single centralized memory but "shared" essentially means that the address space is shared (same physical address on two processors refers to the same location in memory).

The difference between actual shared memory and DSM is illustrated in Fig. 8-21. In Fig. 8-21(a), we see a true multiprocessor with physical shared memory implemented by the hardware. In Fig. 8-21(b), we see DSM, implemented by the operating system. In Fig. 8-21(c), we see yet another form of shared memory,
implemented by yet higher levels of software.

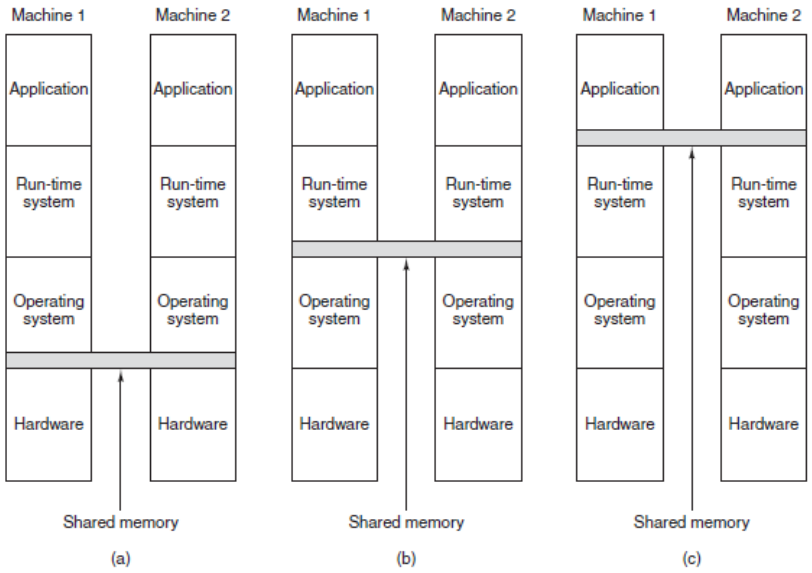


Figure 8-21. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) User-level software.

Let us now look in some detail at how DSM works. In a DSM system, the address space is divided up into pages, with the pages being spread over all the nodes in the system. When a CPU references an address that is not local, a trap occurs, and the DSM software fetches the page containing the address and restarts the faulting instruction, which now completes successfully. This concept is illustrated in Fig. 8-22(a) for an address space with 16 pages and four nodes, each capable of holding six pages.

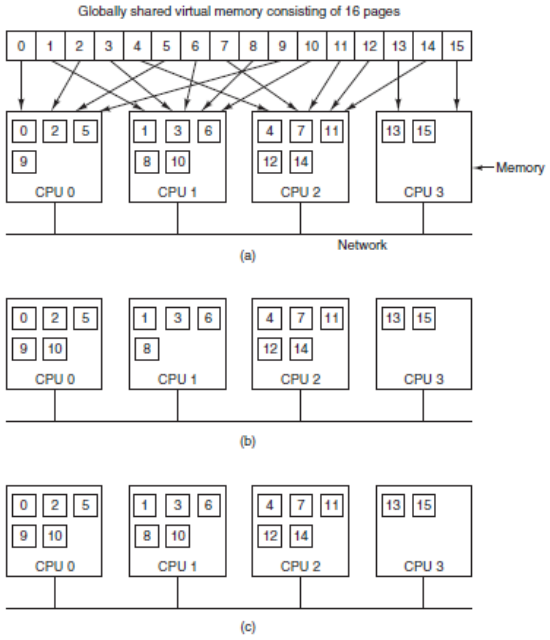


Figure 8-22. (a) Pages of the address space distributed among four machines. (b) Situation after CPU 0 references page 10 and the page is moved there. (c) Situation if page 10 is read only and replication is used.

Replication

- One improvement to the basic system that can improve performance considerably is to replicate pages that are read only, for example, program text, read-only constants, or other read-only data structures.
- Another possibility is to replicate not only read-only pages, but also all pages. As long as reads are being done, there is effectively no difference between replicating a read-only page and replicating a read-write page.

False Sharing

- False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line. Write contention on cache lines is the single most limiting factor on achieving scalability for parallel threads of execution in an SMP system.
- On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processes. Also, too large an effective page size introduces a new problem, called false sharing, illustrated in Fig. 8-23. Here we have a page containing two unrelated shared variables, A and B. Processor 1makes heavy use of A, reading and writing it. Similarly, process 2 uses B frequently. Under these circumstances, the page containing both variables will constantly be traveling back and forth between the two machines.

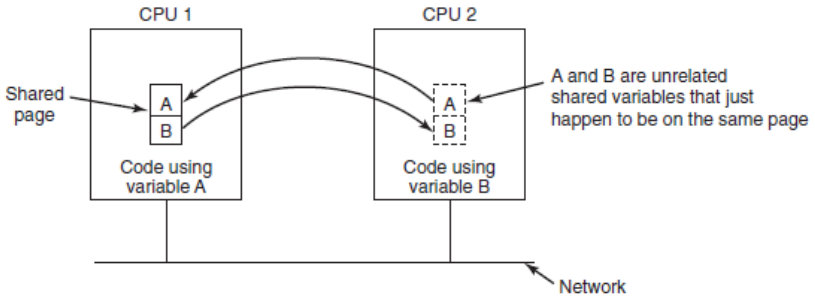


Figure 8-23. False sharing of a page containing two unrelated variables.

Achieving Sequential Consistency:-

If writable pages are not replicated, achieving consistency is not an issue. There is exactly one copy of each writable page, and it is moved back and forth dynamicallyas needed. Since it is not always possible to see in advance which pages are writable, in many DSM systems, when a process tries to read a remote page, local copy is made and both the local and remote copies are set up in their respective MMus as read only. As long as all references are reads, everything is fine.

Multicomputer Scheduling:-

On a multiprocessor, all processes reside in the same memory. When a CPU finishes its current task, it picks a process and runs it. In principle, all processes are potential candidates. On a multicomputer the situation is quite different. Each node has its own memory and its own set of processes. CPU 1 cannot suddenly decide to run a process located on node 4 without first doing a fair amount of work to go get it. This difference means that scheduling on multicomputers is easier but allocation of processes to nodes is more important. Below we will study these issues.

Load Balancing:-

There is relatively little to say about multicomputer scheduling because once a process has been assigned to a node, any local scheduling algorithm will do, unless gang scheduling is being used. However, precisely because there is so little control once a process has been assigned to a node, the decision about which process should go on which node is important. This is in contrast to multiprocessor systems, in which all processes live in the same memory and can be scheduled on any CPU at will. Consequently, it is worth looking at how processes can be assigned to nodes in an effective way. The algorithms and heuristics for doing this assignment are known as processor allocation algorithms.

A Graph-Theoretic Deterministic Algorithm

The system can be represented as a weighted graph, with each vertex being a process and each arc representing the flow of messages between two processes. Mathematically, the problem then reduces to finding a way to partition (i.e., cut) the graph into k disjoint subgraphs, subject to certain constraints (e.g., total CPU and memory requirements below some limits for each subgraph). For each solution that meets the constraints, arcs that are entirely within a single subgraph represent intra machine communication and can be ignored. Arcs that go from one subgraph to another represent network traffic. The goal is then to find the partitioning that minimizes the network traffic while meeting all the constraints. As an example, Fig. 8-24 shows a system of nine processes, A through I , with each arc labeled with the mean communication load between those two processes (e.g., in Mbps).

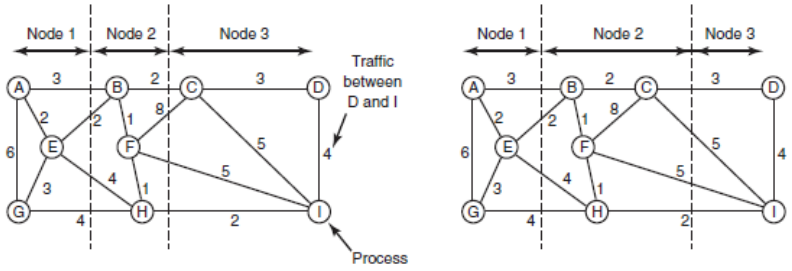


Figure 8-24. Two ways of allocating nine processes to three nodes.

A Sender-Initiated Distributed Heuristic Algorithm

One algorithm says that when a process is created, it runs on the node that created it unless that node is overloaded. The metric for overloaded might involve too many processes, too big a total working set, or some other metric. If it is overloaded, the node selects another node at random and asks it what its load is (using the same metric).

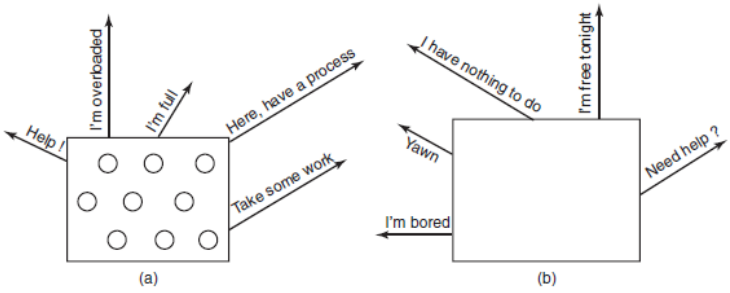


Figure 8-25. (a) An overloaded node looking for a lightly loaded node to hand off processes to. (b) An empty node looking for work to do.

A Receiver-Initiated Distributed Heuristic Algorithm

A complementary algorithm to the one discussed above, which is initiated by an overloaded sender, is one initiated by an under loaded receiver, as shown in Fig. 8-25(b). With this algorithm, whenever a process finishes, the system checks to see if it has enough work. If not, it picks some machine at random and asks it for work. If that machine has nothing to offer, a second, and then a third machine is asked. If no work is found with *N* probes, the node temporarily stops asking, does any work it has queued up, and tries again when the next process finishes. If no work is available, the machine goes idle.

DISTRIBUTED SYSTEMS

These systems are similar to multicomputers in that each node has its own private memory, with no shared physical memory in the system. However, distributed systems are even more loosely coupled than multicomputers. To start with, each node of a multicomputer generally has a CPU, RAM, a network interface, and possibly a disk for paging. In contrast, each node in a distributed system is a complete computer, with a full complement of peripherals. Next, the nodes of a multicomputer are normally in a single room, so they can communicate by a dedicated high-speed network, whereas the nodes of a distributed system may be spread around the world. Finally, all the nodes of a multicomputer run the same operating system, share a single file system, and are under a common administration, whereas the nodes of a distributed system may each run a different operating system, each of which has its own file system, and be under a different administration.

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

Network Hardware

Distributed systems are built on top of computer networks, so a brief introduction to the subject is in order. Networks come in two major varieties, **LANs** (**Local Area Networks**), which cover a building or a campus, and **WANs** (**Wide Area Networks**), which can be citywide, countrywide, or worldwide. The most important kind of LAN is Ethernet, so we will examine that as an example LAN. As our example WAN, we will look at the Internet, even though technically the Internet is not one network, but a federation of thousands of separate networks.

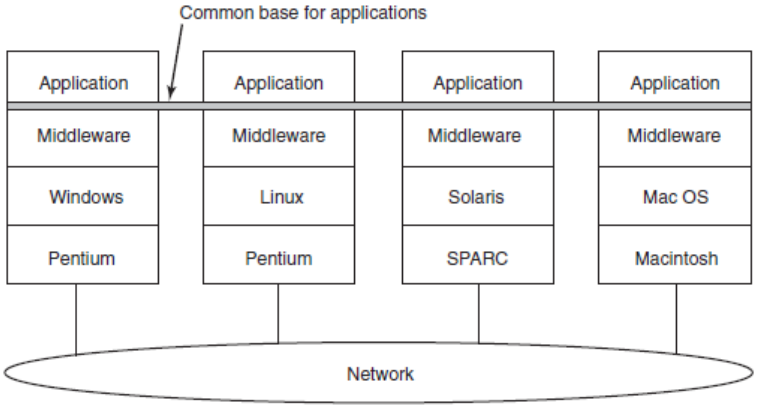


Figure 8-27. Positioning of middleware in a distributed system.

Ethernet

In the very first version of Ethernet, a computer was attached to the cable by literally drilling a hole halfway through the cable and screwing in a wire leading to the computer. This was called a **vampire tap**, and is illustrated symbolically in Fig. 8-28(a). The taps were hard to get right, so before long, proper connectors were used. Nevertheless, electrically, all the computers were connected as if the cables on their network interface cards were soldered together.

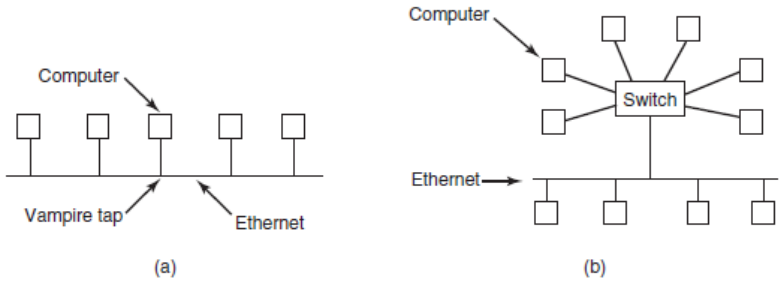


Figure 8-28. (a) Classic Ethernet. (b) Switched Ethernet.

The Internet

The Internet consists of two kinds of computers, hosts and routers. **Hosts** are PCs, notebooks, handhelds, servers, mainframes, and other computers owned by individuals or companies that want to connect to the Internet. **Routers** are specialized switching computers that accept incoming packets on one of many incoming lines and send them on their way along one of many outgoing lines. A router is similar to the switch of Fig. 8-28(b), but also differs from it in ways that will not concern us here. Routers are connected together in large networks, with each router having wires or fibers to many other routers and hosts. Large national or worldwide router networks are operated by telephone companies and ISPs (Internet Service Providers) for their customers.

Figure 8-29 shows a portion of the Internet. At the top we have one of the backbones, normally operated by a backbone operator. It consists of a number of routers connected by high-bandwidth fiber optics, with connections to backbones operated by other (competing) telephone companies. Usually, no hosts connect directly to the backbone, other than maintenance and test machines run by the telephone company.
Attached to the backbone routers by medium-speed fiber optic connections are regional networks and routers at ISPs. In turn, corporate Ethernets each have a router on them and these are connected to regional network routers. Routers at ISPs are connected to modem banks used by the ISP's customers. In this way, eve very host on the Internet has at least one path, and often many paths, to every other host.

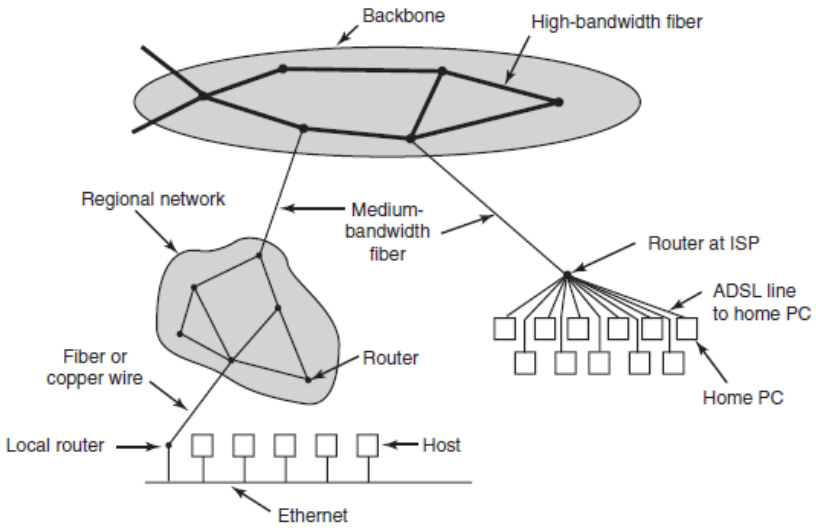


Figure 8-29. A portion of the Internet.

Network Services and Protocols

All computer networks provide certain services to their users (hosts and processes), which they implement using certain rules about legal message exchanges. Below we will give a brief introduction to these topics.

Network Services

Computer networks provide services to the hosts and processes using them. **Connection-oriented service** is modeled after the telephone system. To talk to someone, you pick up the phone, dial the number, talk, and then hang up. Similarly, to use a connection-oriented network service, the service user first establishes a connection, uses the connection, and then releases the connection. The essential aspect of a connection is that it acts like a tube: the sender pushes objects (bits) in at one end, and the receiver takes them out in the same order at the other end. In contrast, **connectionless service** is modeled after the postal system. Each message (letter) carries the full destination address, and each one is routed through the system independent of all the others. Normally, when two messages are sent to the same destination, the first one sent will be the first one to arrive. Howe err, it is possible that the first one sent can be delayed so that the second one arrives first. With a connection-oriented service this is impossible.

	Service	Example
Connection-oriented	Reliable message stream	Sequence of pages of a book
	Reliable byte stream	Remote login
	Unreliable connection	Digitized voice
Connectionless	Unreliable datagram	Network test packets
	Acknowledged datagram	Registered mail
	Request-reply	Database query

Figure 8-30. Six different types of network service.

Network Protocols

The set of rules by which particular computers communicate is called a **protocol**. Since most distributed systems use the Internet as a base, the key protocols these systems use are the two major Internet protocols: IP and TCP. **IP** (**Internet Protocol**) is a datagram protocol in which a sender injects a datagram of up to 64 KB into the network and hopes that it arrives. No guarantees are given. The datagram may be fragmented into smaller packets as it passes through the Internet. These packets travel independently, possibly along different routes.

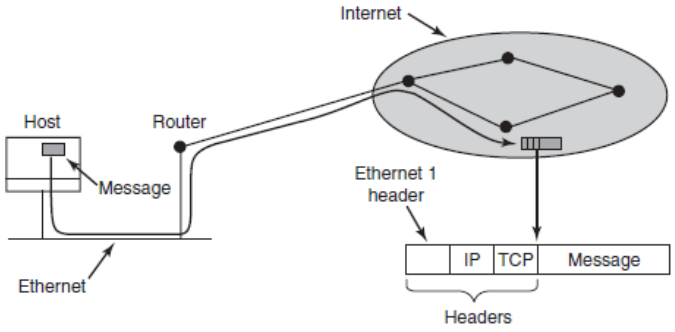


Figure 8-31. Accumulation of packet headers.

DNS names are commonly known because Internet email addresses are of the form *user-name@DNS-host-name*. This naming system allows the mail program on the sending host to look up the destination host's IP address in the DNS database, establish a TCP connection to the mail daemon process there, and send the message as a file. The *user-name* is sent along to identify which mailbox to put the message in.

Document-Based Middleware

The original paradigm behind the Web was quite simple: every computer can hold one or more documents, called **Web pages**. Each Web page contains text, images, icons, sounds, movies, and the like, as well as **hyperlinks** (pointers) to other Web pages. When a user requests a Web page using a program called a **Web browser**, the page is displayed on the screen. Clicking on a

link causes the current page to be replaced on the screen by the page pointed to. Although many bells and whistles have recently been grafted onto the Web, the underlying paradigm is still clearly present: the Web is a great big directed graph of documents that can point to other documents, as shown in Fig. 8-32.

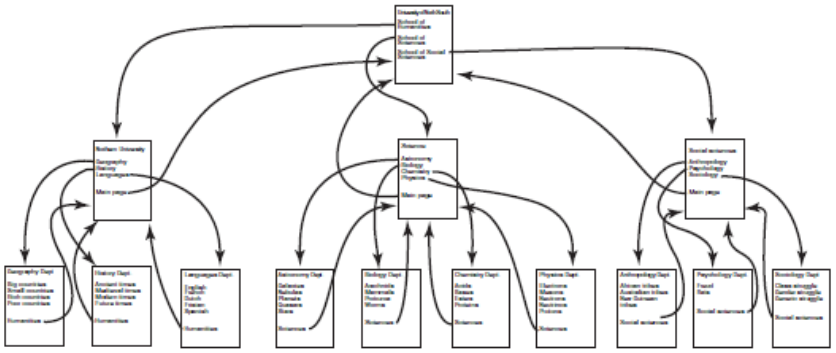


Figure 8-32. The Web is a big directed graph of documents.

Each Web page has a unique address, called a **URL** (**Uniform Resource Locator**), of the form *protocol://DNS-name/file-name*. The protocol is most commonly *http* (Hypertext Transfer Protocol), but *ftp* and others also exist. Then comes the DNS name of the host containing the file. Finally, there is a local file name telling which file is needed. Thus a URL uniquely specifies a single file worldwide

As a simple example, suppose the URL provided is *http://www.minix3.org/getting-started/index.html*. The browser then takes the following steps to get the page.

1. The browser asks DNS for the IP address of *www.minix3.org*.
2. DNS replies with 66.147.238.215.
3. The browser makes a TCP connection to port 80 on 66.147.238.215.
4. It then sends a request asking for the file *getting-started/index.html*.
5. The *www.minix3.org* server sends the file *getting-started/index.html*.
6. The browser displays all the text in *getting-started/index.html*.
7. Meanwhile, the browser fetches and displays all images on the page.
8. The TCP connection is released.

File-System-Based Middleware

The basic idea behind the Web is to make a distributed system look like a giant collection of hyperlinked documents. A second approach is to make a distributed system look like a great big file system. In this section we will look at some of the issues involved in designing a worldwide file system.