the system throughput—the number of tasks completing per time unit—without speeding up the execution of individual tasks. Second, a multiprocessor system can speed up the execution of a single task in the following way: if parallelism exists in a task, it can be divided into many subtasks and these subtasks can be executed in parallel on different processors.

Fault tolerance. A multiprocessor system exhibits graceful performance degradation to processor failures because of the availability of multiple processors.

16.3 BASIC MULTIPROCESSOR SYSTEM ARCHITECTURES

According to the classification of Flynn [6], in MIMD (multiple instruction multiple data) architectures, multiple instruction streams operate on different data streams. In the broadest sense, an MIMD architecture qualifies as a full-fledged multiprocessor system. Thus, a multiprocessor system consists of multiple processors, which execute different programs (or different segments of a program) concurrently. The main memory is typically shared by all the processors. Based on whether a memory location can be directly accessed by a processor or not, there are two types of multiprocessor systems: tightly coupled and loosely coupled [7].

16.3.1 Tightly Coupled vs. Loosely Coupled Systems

In *tightly coupled systems*, all processors share the same memory address space and all processors can directly access a global main memory. Examples of commercially available tightly coupled systems are Multimax of Encore Corporation, Flex/32 of Flexible Corporation, and FX of Sequent Computers.

In *loosely coupled systems*, not only is the main memory partitioned and attached to processors, but each processor has its own address space. Therefore, a processor cannot *directly* access the memory attached to other processors. One example of a loosely coupled system is Intel's Hypercube.

Tightly coupled systems can use the main memory for interprocessor communication and synchronization (see Chap.2). Loosely coupled systems, on the other hand, use only message passing for interprocessor communication and synchronization (see Chap. 4).

We limit our discussion to tightly coupled multiprocessor systems. Figure 16.1 illustrates the schematic diagram of a typical tightly coupled multiprocessor system. A number of processors are connected to the shared memory by an interconnection network. The shared memory is normally divided into several modules and multiple modules can be accessed concurrently by different processors. A *memory contention* occurs when two or more processors simultaneously try to access the same memory module. In case of a memory contention, the request of only one of the requesting processors can be met. The requests of other processors can be queued up for later processing.

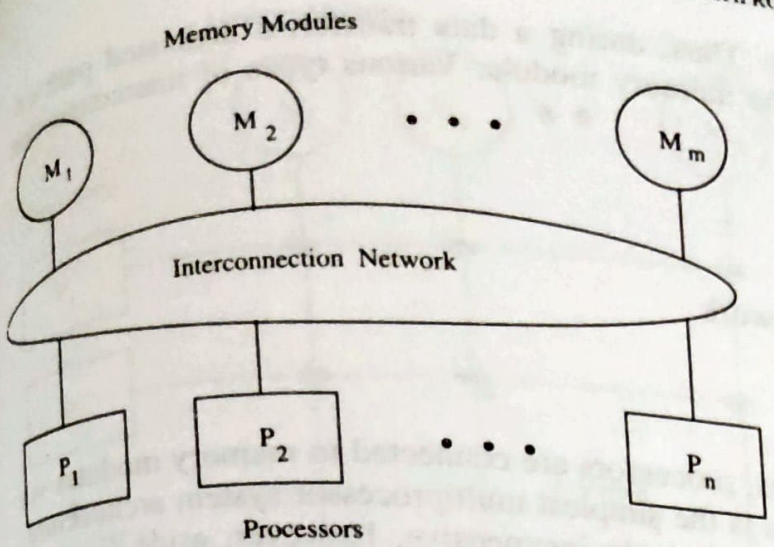

FIGURE 16.1
A tightly coupled multiprocessor system.

16.3.2 UMA vs. NUMA vs. NORMA Architectures

Based on the vicinity and accessibility of the main memory to the processors, there are three types of multiprocessor system architectures: UMA (uniform memory access), NUMA (nonuniform memory access), and NORMA (no remote memory access).

In UMA architectures, the main memory is located at a central location such that it is equidistant from all the processors in terms of access time (in the absence of conflicts). That is, all the processors have the same access time to the main memory. In addition to this centralized shared memory, processors may also have private memories, where they can cache data for higher performance. Some examples of UMA architectures are Multimax of Encore Corporation, Balance of Sequent, and VAX 8800 of Digital Equipment.

In NUMA architectures, main memory is physically partitioned and the partitions are attached to the processors. All the processors, however, share the same memory address space. A processor can directly access the memory attached to any other processor, but the time to access the memory attached to other processors is much higher than the time to access its own memory partition. Examples of NUMA architectures are Cm* of CMU and Butterfly machine of BBN Laboratories.

In NORMA architectures, main memory is physically partitioned and the partitions are attached to the processors. However, a processor cannot directly access the memory of any other processor. The processors must send messages over the interconnection network to exchange information. An example of NORMA architecture is Intel's Hypercube.

16.4 INTERCONNECTION NETWORKS FOR MULTIPROCESSOR SYSTEMS

The interconnection network in multiprocessor systems provides data transfer facility between processors and memory modules for memory access [5]. The design of the interconnection network is the most crucial hardware issue in the design of multiprocessor systems. Generally, circuit switching is used to establish a connection between

processors and memory modules. Thus, during a data transfer, a dedicated path exists between the processor and the memory module. Various types of interconnection networks include:

- Bus
- Cross-bar Switch
- Multistage Interconnection Network

16.4.1 Bus

In bus-based multiprocessor systems, processors are connected to memory modules via a bus (Fig. 16.2). Conceptually, this is the simplest multiprocessor system architecture. It is also easy to implement and is relatively inexpensive. However, aside from the shared memory, the bus is also a source of contention because the bus can support only one processor-memory communication at any time. Moreover, this architecture can support only a limited number of processors because of the limited bandwidth of the bus. These problems can be mitigated by using multiple buses to connect processors and memories. In a b bus system, up to b processor-memory data transfers can take place concurrently. CMU's Cm* and Encore Corporation's Multimax are examples of bus-based multiprocessor systems.

16.4.2 Cross-bar Switch

A cross-bar switch is a matrix (or grid structure) that has a switch at every cross-point. Figure 16.3 shows a multiprocessor system with n processors and m memory modules. A cross-bar is capable of providing an exclusive connection between any processor-memory pair. Thus, all n processors can concurrently access memory modules provided that each processor is accessing a different memory module (and $n \leq m$). A cross-bar switch does not face contention at the interconnection network level. A contention can occur only at the memory module level.

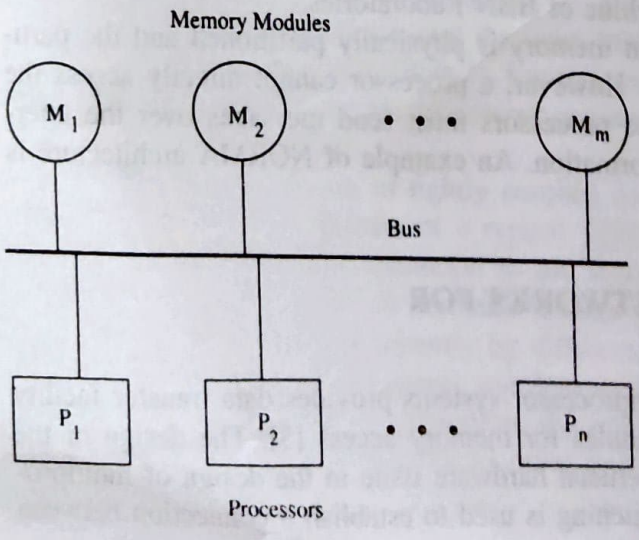

FIGURE 16.2
A multiprocessor system with a bus.

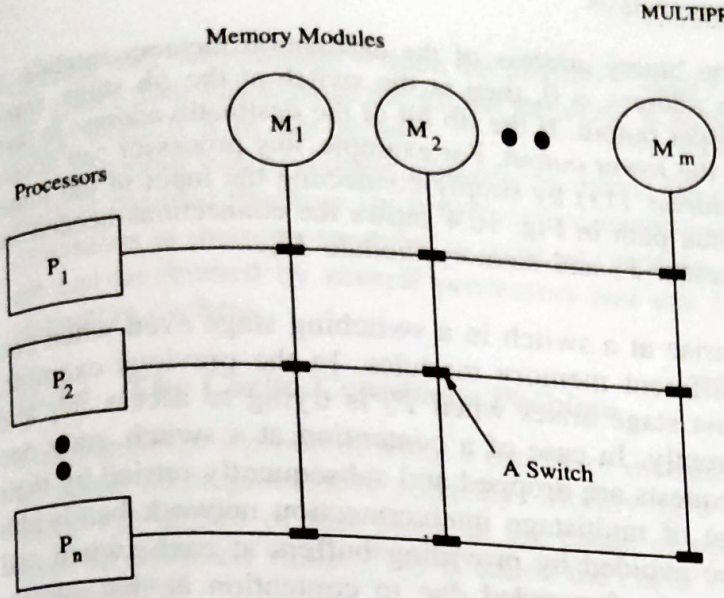

FIGURE 16.3
A multiprocessor system with a cross-bar.

Cross-bar based multiprocessor systems are relatively expensive and have limited scalability because of the quadratic growth of the number of switches with the system size ($n \times n$ if there are n processors and n memory modules). Alliant FX/8 is an example of a commercially available cross-bar architecture.

16.4.3 Multistage Interconnection Network

A multistage interconnection network is a compromise between a bus and a cross-bar switch. A multistage interconnection network permits simultaneous connections between several processor-memory pairs and is more cost-effective than a cross-bar. A typical multistage interconnection network consists of several stages of switches. Each stage consists of an equal number of cross-bar switches of the same size (such as 2×2 or 4×4). The outputs of the switches in a stage is connected to the inputs of the switches in the next stage. These connections are made in such a way that any input to the network can be connected to any output of the network (by making the appropriate connections in the switches at each stage). Depending upon how output-input connections between adjacent stages are made, there are numerous types of interconnection networks [5]. The routing path between a processor and a memory module pair is given by a binary string that is derived from the binary addresses of the processor and the memory module. The ith bit of this binary string determines to which output the input should be connected at the switch at stage i.

An $N \times N$ multistage interconnection network can connect N processors to N memory modules. If $N = 2^k$, it will consist of k ($= \log_2 N$) stages of 2×2 switches with $N/2$ switches in each stage. Thus, an $N \times N$ multistage interconnection network requires only $(N/2) \times \log_2 N$ switches as compared to N^2 switches in an $N \times N$ cross-bar.

Example 16.1. Figure 16.4 shows an 8×8 Omega multistage interconnection network that is constructed from 2×2 cross-bar switches. Note that there exists a *unique* path between a processor-memory pair. In this case, the routing through various stages is

completely determined by the binary address of the destination memory module. If the ith bit of the destination address is 0, then at the switch at the ith stage, input should be connected to the upper output. If the ith bit of the destinatin address is one, input should be connected to the lower output. For example, any processor can access memory module M_8 (with address 111) by simply connecting the input to the lower output at every stage. The thick path in Fig. 16.4 shows the connections needed for communication between processor P_2 and memory module M_1.

Note that a contention can arise at a switch in a switching stage even when two processors are trying to access different memory modules. In the previous example, a contention at a switch in the first stage arises when P_2 is trying to access M_1 and P_6 is trying to access M_4 concurrently. In case of a contention at a switch, only one request succeeds and rest of the requests are dropped and subsequently retried by their respective processors. The wastage of multistage interconnection network bandwidth due to the retry of requests can be avoided by providing buffers at each switch and buffering the requests, which cannot be forwarded due to contention at that switch, for later transmission. Clearly such multistage interconnection networks fall under the category of store-and-forward networks. The BBN Butterfly machine is an example of a commercially available multiprocessor system that uses multistage interconnection network.

CHAPTER
17

MULTIPROCESSOR OPERATING SYSTEMS

17.1 INTRODUCTION

Multiprocessor operating systems are similar to multiprogrammed uniprocessor operating systems in many respects and they perform resource management and hide unpleasant idiosyncracies of the hardware to provide a high-level machine abstraction to the users. However, multiprocessor operating systems are more complex because multiple processors execute tasks concurrently (with *physical* as opposed to *virtual* concurrency in multiprogrammed uniprocessors.) Thus, a multiprocessor operating system must be able to support the concurrent execution of multiple tasks and must prudently exploit the power of multiple processors to increase performance.

17.2 STRUCTURES OF MULTIPROCESSOR OPERATING SYSTEMS

Based upon the nature of the control structure and its organization, there are three basic classes of multiprocessor operating systems: separate supervisor, master-slave, and symmetric [13].

THE SEPARATE SUPERVISOR CONFIGURATION. In the separate supervisor configuration, all processors have their own copy of the kernel, supervisor, and data structures. There are some common data structures for the interaction among processors,

the access to which is protected by using some synchronization mechanism (such as semaphores). Each processor has its own I/O devices and file system. There is very little coupling among processors and each processor acts as an autonomous, independent system. Therefore, it is difficult to perform parallel execution of a single task (that is, to break up a task and schedule the subtasks on multiple processors concurrently). Also, this configuration is inefficient because the supervisor/kernel/data structure code is replicated for each processor. This configuration, however, degrades gracefully in the face of processor failures because there is very little coupling among processors.

THE MASTER-SLAVE CONFIGURATION. In the master-slave configuration, one processor, called the *master*, monitors the status and assigns work to all other processors, the *slaves*. Slaves are treated as a schedulable pool of resources by the master. Such an operating system is simple because it runs only on the master processor. (The slave processors essentially execute application programs.) Since the operating system is executed by a single processor, it is efficient and its implementation (synchronization of access to shared variables, etc.) is easy. The master-slave configuration permits the parallel execution of a single task, where a task can be broken into several subtasks and the subtasks can be scheduled on multiple processors concurrently.

However, an operating system based on the master-slave configuration is highly susceptible to the failure of the master processor. Also, the master can become a bottleneck and will consequently fail to fully utilize slave processors. Examples of such operating systems are Cyber 170 and DEC-10.

THE SYMMETRIC CONFIGURATION. In the symmetric configuration, all processors are autonomous and are treated equally. There is one copy of the supervisor or kernel that can be executed by all processors concurrently. However, concurrent access to the shared data structures of the supervisor needs to be controlled in order to maintain their integrity. The simplest way to achieve this is to treat the entire operating system as a critical section and allow only one processor to execute the operating system at one time. This method is called the *floating master* method because it can be viewed as a master-slave configuration where the master "floats" from one processor to another. Note that the execution of the operating system by processors can become a bottleneck in this method. This problem can be mitigated by dividing the operating system into segments that normally have very little interaction (i.e., the sharing of variables, communication, etc.) such that the segments can be executed concurrently by the processors (although each segment is still executed serially). This method requires a serialization mechanism that controls concurrent access to the shared data structures of the supervisor.

The symmetric configuration is the most flexible and versatile of all the configurations. It permits the parallel execution of a single task. It degrades gracefully under failures and makes very efficient use of resources. However, it is the most difficult configuration to design and implement. Examples of such an operating system include Hydra on C.mmp.

17.3 OPERATING SYSTEM DESIGN ISSUES

A multiprocessor operating system encompasses all the functional capabilities of the operating system of a multiprogrammed uniprocessor system. However, the design of a multiprocessor operating system is complicated because it must fulfill the following requirements. A multiprocessor operating system must be able to support concurrent task execution, it should be able to exploit the power of multiple processors, it should fail gracefully, and it should work correctly despite physical concurrency in the execution of processes. The design of multiprocessor operating systems involves the following major issues:

Threads. The effectiveness of parallel computing depends greatly on the performance of the primitives that are used to express and control parallelism within an application. It has been recognized that traditional processes impose too much overhead for context switching. In light of this, threads have been widely utilized in recent systems to run applications concurrently on many processors.

Process Synchronization. In a multiprocessor operating system, disabling interrupts is not sufficient to synchronize concurrent access to shared data. A more elaborate mechanism that is based on shared variables is needed. Moreover, a synchronization mechanism must be carefully designed so that it is efficient, otherwise, it could result in significant performance penalty.

Processor Scheduling. To ensure the efficient use of its hardware, a multiprocessor operating system must be able to utilize the processors effectively in executing the tasks. A multiprocessor operating system, in cooperation with the compiler, should be able to detect and exploit the parallelism in the tasks being executed.

Memory Management. The design of virtual memory is complicated because the main memory is shared by many processors. The operating system must maintain a separate map table for each processor for address translation. When several processors share a page or segment, the operating system must enforce the consistency of their entries in respective map tables. Moreover, efficient page replacement becomes a complex issue.

Reliability and Fault Tolerance. The performance of a multiprocessor system must be able to degrade gracefully in the event of failures. Thus, a multiprocessor operating system must provide reconfiguration schemes to restructure the system in the face of failures to ensure graceful degradation.

Next, these issues in the design of multiprocessor operating systems are discussed in detail. Other issues include protection and interprocess communication. Protection deals with the design of mechanisms that prevent unauthorized access to resources. Interprocess communication in an operating system calls for a support of a variety of models for communication between processes.

17.4 THREADS

Traditionally, a process has a single address space and a single thread of control to execute a program within that address space. To execute a program, a process has to

initialize and maintain state information. The state information typically comprises page tables, swap images, file descriptors, outstanding I/O requests, saved register values, etc. This information is maintained on a per program basis, and thus, a per process basis. The volume of this state information makes it expensive to create and maintain processes as well as switch between them.

With the advent of shared memory multiprocessor machines, it became imperative to create and switch between processes to take advantage of the concurrency available in many programs. The effectiveness of parallel computing depends to a great extent on the performance of the primitives that are used to express and control the parallelism within the program [2]. In networking systems, servers provide various services to machines connected to the network. For instance, file servers provide file system services to the machines in the network (see Chap. 9). These servers (typically uniprocessor machines) cater to different requests from different users. The design of servers may be simplified if separate processes are maintained at the server to cater to each active user. To provide service to different users, it becomes necessary to switch between processes efficiently. To handle the situations where creating, maintaining, and switching between processes occur frequently, *threads* or *lightweight processes* have been proposed.

A thread separates the notion of execution from the rest of the definition of a process [3]. A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Each thread makes use of a separate program counter, a stack of activation records (which describe the state of the execution), and a control block. The control block contains the state information necessary for thread management such as putting a thread into a ready list and synchronizing with other threads. Most of the information that is part of a process is also common to all the threads executing within a single address space and hence can be maintained in common to all the threads. By sharing common information, the overhead incurred in creating and maintaining the information, and the amount of information that needs to be saved when switching between threads of the same program, is reduced significantly.

Threads can be supported either at the user-level or at the kernel-level. We next discuss the advantages, disadvantages, and performance implications of supporting threads at these levels.

17.4.1 User-Level Threads

In user-level threads, a run-time library package provides the routines necessary for thread management operations. These routines are linked at runtime to applications. Kernel intervention is not required for the management of threads. The libraries multiplex a potentially large number of user-defined threads on top of a single kernel-implemented process. Typically, the cost of a user-level thread operation is within an order of magnitude of the cost of a procedure call. Because of their low cost, user-level threads can provide excellent performance compared to kernel-level threads. In addition, user-level threads have the following advantages.

- No modifications in the existing operating system kernel are required to support user-level threads.

- They are flexible. They can be customized to suit the language or needs of the users and libraries can be used to implement different thread packages that are customized differently for various users. Thus, overhead due to providing all the capabilities or facilities in one package can be avoided. An example of customizing is where one set of library routines can provide preemptive priority scheduling, while another set can provide the simpler first-in-first-out scheduling.

While user-level threads have their advantages, they have the following disadvantages.

- The generally excellent performance of user-level threads may be limited to applications such as parallel programs that require little kernel involvement. User-level threads operate within the context of traditional processes. Thread systems treat a process as a virtual processor, and consider it a physical processor executing under its control. In reality, however, the physical processors are controlled by the operating system kernel. The kernel might assign a different physical processor to a virtual processor during each timeslice. In addition, other factors such as I/O, multiprogramming, and page faults can distort the equivalence between the virtual processor and the physical processor assumed by the thread system. In other words, there is a lack of coordination between scheduling and synchronization. For example, a thread executing in a critical section may be preempted by the kernel, making other threads wait longer. Another example is that of an application that assumes that all its runnable threads are served in a finite time. However, timeslicing across a fixed number of kernel threads by the kernel across many applications may make this assumption untrue. Note that when a thread blocks, the underlying kernel process also blocks. Eventually, the application may run out of kernel threads to serve its execution contexts, even when there are runnable threads. This situation may lead to a deadlock [2].
- User-level threads require that system calls be nonblocking. If a thread blocks because of a system call, it will prevent other runnable threads from executing. Note that many frequently performed system calls—file open, file close, read, and write—block under UNIX [16].

17.4.2 Kernel-Level Threads

In kernel-level threads, the kernel directly supports multiple threads per address space [7], [26], [27]. The kernel also provides the operations for thread management. The kernel-level threads have the following advantages.

- Coordination between the synchronization and scheduling of threads is easy to achieve, since the kernel has all the information concerning the status of all the threads.
- They are suitable for multithreaded applications, such as server processes, where interactions with the kernel are frequent due to IPC, page faults, exceptions, etc. [9].
- They incur less overhead compared to traditional processes.

The disadvantages of kernel-level threads are as follows.

- Thread management operations incur higher overhead relative to user-level threads. Every operation involves a kernel trap, even when the processor is multiplexed between the threads in the same address space. On every thread operation, there is overhead due to copying and checking of parameters being passed to the kernel to ensure safety [2].
- Since the kernel is the sole provider of thread managing operations, it has to provide any feature needed by any reasonable application. This generality means that even applications not using a particular feature still have to incur overhead due to unused features provided in the kernel.

In summary: (1) kernel-level threads are too costly to use, and (2) user-level threads can provide excellent performance, but problems such as a lack of coordination between synchronization and scheduling, and blocking system calls, pose serious obstacles to the realization of performance potential.

System developers have favored user-level threads, despite their disadvantages, because of their potential for excellent performance. The cause of the problems with user-level threads are traced to the following facts.

- User-level threads are not recognized or supported by the kernel [16].
- Kernel events, such as processor preemption and I/O blocking and resumption, are handled by the kernel in a manner invisible to the user-level [2].
- Kernel threads are obliviously scheduled with respect to the user-level thread state [2].

The above problems have been addressed in at least two different ways: (1) by granting user-level threads a first-class status so that they can be used as traditional processes, while leaving the details of the implementations to the user-level code [16], and (2) through the explicit vectoring of kernel events to the user-level threads scheduler [2]. We next describe two thread mechanisms based on the above approaches.

17.4.3 First-Class Threads

First-class threads [16] were developed as a part of the Psyche parallel operating system [21]. Kernel processes are used to implement the virtual processor that execute user-level threads. Creating many virtual processors in the same address space and assigning them to different physical processors provides parallelism.

In Psyche, a thread package creates and maintains the state of the threads in user-space. Most of the thread operations, such as creation, destruction, synchronization, and the context switching of threads, are handled by the thread package. However, coarse-grain resource allocation and protection (such as preemptive scheduling) is in the domain of the kernel.

Under first-class threads, to overcome the problems associated with the user-level threads, three mechanisms are provided to communicate (in both directions) between

the kernel and the thread package. These communications occur without any kernel traps. Descriptions of these mechanisms follow.

1. The kernel and the thread package share important data structures. The kernel managed data is made available to the thread package through read-only access. For example, thread package can obtain the current processor ID and process ID without making a system call. In the opposite direction, through the shared data structure, the thread package can communicate with the kernel. For instance, the thread package can specify what actions are to be taken when the kernel detects events such as a timer expiration.
2. The kernel provides the thread package with software interrupts (signals, upcalls) whenever a scheduling decision is required. Essentially, on interruption, a user-level interrupt handler is activated. The interrupt handler then takes care of the scheduling decision. Following are instances when the software interrupts are employed. When a thread blocks or resumes after blocking because of a system call, the kernel delivers an interrupt that allows the thread package to schedule an appropriate thread. Timer interrupts support the timeslicing of threads. Warnings prior to imminent preemption allow the thread package to coordinate synchronization with the kernel resource allocation. For example, the thread package may decide to postpone obtaining locks if it is faced with imminent preemption.
3. Scheduler interfaces are provided to enable the sharing of data abstractions between dissimilar thread packages. The interfacing occurs through the thread scheduling routines available in the thread package. These routines are listed in the thread data structure shared between the kernel and the thread package. The typical usage of these interfaces is to block and unblock threads at the user-level. Consider for example, the producer-consumer problem where producer and consumer threads are from different thread packages. When the consumer thread tries to read a buffer and finds it empty, the identity of the thread unblocking routines (available in the thread data structure) can be stored in the shared buffer before blocking the consumer. The producer, on storing data in the buffer, will find the address of the saved routines and can unblock the consumer thread.

17.4.4 Scheduler Activations

A scheme based on scheduler activations to overcome the disadvantages of user-level threads has been developed at the University of Washington [2].

Under this scheme, communication between the kernel and a user-level thread package is structured in terms of scheduler activations. A scheduler activation has three roles. (1) It serves as an execution context for running user-level threads. (2) It notifies the user-level thread system of kernel events. (3) It provides space in the kernel for saving the processor context of the activation's current user-level thread when the thread is stopped by the kernel.

When a program starts, the kernel creates a scheduler activation, assigns it to a processor, and upcalls into the program's address space at a fixed entry point. Once the thread system receives the upcall, it uses the activation's context to initialize itself

and then runs a program's thread. This thread may create additional threads and request additional processors. For each processor request, the kernel will create a scheduler activation and assigns a processor to it, and then upcall into the thread system's user-space. Note that once an upcall is started, the activation is similar to a thread. It can be used to run a user-level thread, process an event, or make system calls (which can block).

NOTIFYING KERNEL-LEVEL EVENTS TO THE USER-LEVEL THREAD SYSTEM. To notify the thread system of kernel-level events, the kernel creates a new scheduler activation, assigns it to a processor, and then upcalls into the user-space. We next describe how several common kernel events are handled.

When a user-level thread blocks in the kernel space, the kernel creates a new scheduler activation to inform the thread system that the thread has blocked. The thread system then saves the state of the blocked thread, frees the activation used by the blocked thread, and informs the kernel that the activation is free for reuse. Then the thread system decides which thread to run next using the new activation. Note that the number of scheduler activations assigned to an application is always equal to the number of processors assigned to the application.

When a user-level thread that was stopped in the kernel resumes, it may have to continue in the kernel space. In such a case, the kernel resumes the thread temporarily until it reblocks or is at a point where it will exit the kernel space. In the latter case, the thread system is informed of the unblocking of the thread through an activation.

Sometimes, when the kernel wishes to inform the thread system of an event, a processor may not be available to assign to an activation. In such a case, the kernel stops a thread belonging to the application to which the event has to be informed, uses that processor to upcall into the thread system, and informs the thread system of the event and that a thread has been stopped. Now the thread system is free to handle the event and schedule an appropriate thread.

If the kernel decides to take a processor away from an application, the kernel stops two threads belonging to that application, thus freeing two processors. One processor is assigned to an activation meant for a different address space. The second processor is assigned to a new activation, using which the kernel informs the thread system that two threads of the application are stopped. Now the thread system is free to schedule any one of the threads that it deems appropriate.

Whenever the thread system learns that a thread is preempted, it checks to see whether the thread was executing a critical section. If so, the thread system assigns a processor to the thread through a user-level context switch. Once the thread is out of the critical section, the thread is put back into the ready queue.

It is important to note that under no circumstance does the kernel deal with the scheduling of threads. It is always the thread system that handles this.

NOTIFYING USER-LEVEL EVENTS TO THE KERNEL. The thread system notifies the kernel whenever the thread system enters a state wherein it has more processors than runnable threads or has more runnable threads than the number of assigned processors.

17.5 PROCESS SYNCHRONIZATION

The execution of a concurrent program on a multiprocessor system may require the processors to access shared data structures and thus may cause the processors to concurrently access a location in the shared memory. Clearly, a mechanism is needed to serialize this access to shared data structures to guarantee its correctness. This is the classic mutual exclusion problem. The mechanism should make accesses to a shared data structure appear atomic with respect to each other.

17.5.1 Issues in Process Synchronization

Although numerous solutions exist for process synchronization in uniprocessor systems, these solutions are not suitable for a multiprocessor system. This is because busy-waiting by processors can cause excessive traffic on the interconnection network, thereby degrading system performance. For example, software solutions to the mutual exclusion problem (such as Dekker's solution or, Peterson's method [22]) are impractical for multiprocessor systems because they do busy-waiting and are likely to consume substantial bandwidth of the interconnection network. To overcome this problem, multiprocessor systems provide instructions to atomically read and write a single memory location (in the main memory). If the operation on shared data is very elementary (such as an integer increment), it can be embedded in a single atomic machine language instruction. Thus, mutual exclusion can be implemented completely in hardware provided the operation on the shared data is elementary.

However, if an access to a shared data constitutes several instructions (which is, the critical section consists of several instructions), then primitives such as lock and unlock (or P and V operations) are needed to ensure mutual exclusion. In such cases, the acquisition of a lock itself entails performing an elementary operation on a shared variable (which indicates the status of the lock). Atomic machine language instructions can be used to implement the lock operation, which automatically serialize concurrent attempts to acquire a lock. Next, we discuss several such atomic hardware instructions and describe how they can be used to implement P and V operations [8], [10].

17.5.2 The Test-and-Set Instruction

The test-and-set instruction atomically reads and modifies the contents of a memory location in one memory cycle. It is defined as follows (variable m is a memory location):

```
function Test-and-Set(var m: boolean): boolean;
begin
    Test-and-Set:=m;
    m:=true
end;
```

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

P(S): while Test-and-Set(S) do nothing;
V(S): S:= false;

Initially, S is set to $false$. When a P(S) operation is executed for the first time, Test-and-Set(S) returns a $false$ value (and sets S to $true$) and the "while" loop of the P(S) operation terminates. All subsequent executions of P(S) keep looping because S is $true$ until a V(S) operation is executed.

17.5.3 The Swap Instruction

The swap instruction atomically exchanges the contents of two variables (e.g., memory locations). It is defined as follows (x and y are two variables):

```
procedure swap(var x, y: boolean);
var temp: boolean;
begin
    temp:= x;
    x:= y;
    y:=temp
end;
```

P and V operations can be implemented using the swap instruction in the following way (p is a variable private to the processor and S is a memory location):

```
P(S): p:=true;
      repeat swap(S, p) until p=false;
V(S): S:= false;
```

Clearly, the above two implementations of the P operation employ busy-waiting and therefore increase the traffic on the interconnection network. Another problem with test-and-set and swap instructions is that if n processors execute any of these operations on the same memory location, the main memory will perform n such operations on the location even though only one of these operations will succeed. Next, we discuss a *fetch-and-add* instruction that eliminates this overhead from the memory.

17.5.4 The Fetch-and-Add Instruction of the Ultracomputer

The fetch-and-add instruction of the NYU Ultracomputer [12] is a multiple operation memory access instruction that atomically adds a constant to a memory location and returns the previous contents of the memory location. This instruction is defined as follows (m is a memory location and c is the constant to be added).

```
Function Fetch-and-Add(m: integer; c: integer);
var temp: integer;
begin
    temp:= m;
    m:= m + c;
    return (temp)
end;
```

An interesting property of this instruction is that it is executed by the hardware placed in the interconnection network (not by the hardware present in the memory modules). When several processors concurrently execute a fetch-and-add instruction on the same memory location, these instructions are combined in the network and are executed by the network in the following way. A single increment, which is the sum of the increments of all these instructions, is added to the memory location. A single value is returned by the network to each of the processors, which is an arbitrary serialization of the execution of the individual instructions. If a number of processors simultaneously perform fetch-and-add instructions on the same memory location, the net result is as if these instructions were executed serially in some unpredictable order.

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

```
P(S): while (Fetch-and-Add(S, −1) < 0) do
      begin
            Fetch-and-Add(S, 1);
            while (S ≤ 0) do nothing;
      end;
```

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

```
V(S): Fetch-and-Add(S, 1)
```

17.5.5 SLIC Chip of the Sequent

The Sequent Balance/21000 multiprocessor system supports a low-level mutual exclusion in hardware using a technique that is totally different from the previously discussed techniques, which use atomic multi-operation machine language instructions. The main component of a Balance/21000 is a SLIC (system link and interrupt controller) chip that supports many other functions in addition to low-level mutual exclusion.

A SLIC chip contains 64 single-bit registers and supports the operations necessary for process synchronization. Each processor has a SLIC chip and all the SLIC chips are connected by a separate SLIC bus. Each bit in the SLIC chip, called a *gate*, acts as a separate lock and stores the status of the corresponding lock. Balance/21000 replicates these 64 status bits over all the processors instead of keeping them at a central place, e.g., the shared main memory (as in the previous techniques). Thus, this method substantially reduces traffic on the network that connects memory modules to the processors and it also expedites lock access time.

To lock a gate in the SLIC chip, a processor executes a lock-gate instruction. A lock-gate instruction is executed in the following manner. If the local copy (i.e., the bit in its SLIC chip) indicates that the gate is closed, the instruction fails. Otherwise, the local SLIC of the processor attempts to close the gate by sending messages to other

SLIC chips over the SLIC bus. If multiple SLIC chips attempt to close the same gate concurrently, only one of them succeeds and the rest of them fail. When the status of a gate changes because of the successful execution of a lock-gate or an unlock-gate instruction, an appropriate message is sent over the SLIC bus, which causes every SLIC chip to update its copy of the gate.

The following code implements P and V operations on a semaphore S:

P(S): while (lock-gate(S) = failed) do nothing;
V(S): unlock-gate(S);

Since busy-waiting is performed by checking the local SLIC, the SLIC bus is not overloaded due to busy-waiting. However, processors still waste CPU cycles because they continuously check the status of their SLIC chips.

17.5.6 Implementation of Process Wait

In all the implementations of a P operation discussed thus far, several processors may *wait* for the semaphore to open by executing the respective atomic machine language instructions concurrently. This wait can be implemented in three ways:

Busy Waiting. In busy-waiting, processors continuously execute the atomic instruction to check for the status of the shared variable. Busy-waiting (also called *spin lock*) wastes processor cycles and consumes the bandwidth of the network connecting memory modules to the processors. Increased traffic on the network due to busy-waiting can interfere with the normal memory accesses and degrade the system performance due to the increased memory access time.

Sleep-Lock. In sleep-lock, instead of continuously spinning the lock, a process is suspended when it fails to obtain the lock and a suspended process relies on interrupts to become reactivated when the lock is freed. When a process fails to obtain a lock, it is suspended. In this suspended state, a process does not relinquish its processor and all interrupts except interprocessor interrupts are disabled. When a process frees the lock, it sends interprocessor interrupts to all the suspended processors. This method substantially reduces network traffic due to busy-waiting, but it still wastes processor cycles.

Queueing. In queueing, a process waiting for a semaphore to open is placed in a global queue. A waiting process is dequeued and activated by a V operation on the semaphore. Although queueing eliminates network traffic and the wastage of processor cycles due to busy-waiting, it introduces other processing overhead because the enqueue and the dequeue operations require the execution of several instructions. Also, the queue forms a shared data structure and must be protected against concurrent access.

17.5.7 The Compare-and-Swap Instruction

The compare-and-swap instruction of IBM 370 is used in the optimistic synchronization of concurrent updates to a memory location. This instruction is defined as follows ($r1$ and $r2$ are two registers of a processor and m is a memory location):

```
Compare-and-Swap(var r1, r2, m: integer);
var temp: integer;
begin
    temp:= m;
    if temp = r1 then {m:= r2; z:=1}
    else {r1:= temp; z:= 0}
end;
```

If the contents of $r1$ and m are identical, this instruction assigns the contents of $r2$ to m and sets z to 1. Otherwise, it assigns the contents of m to $r1$ and sets z to 0. Variable z is a flag that indicates the success of the execution of the instruction. An execution of the instruction is successful if $z = 1$ after the execution. The intuitive meaning of "successful" should become clear from the example in the next paragraph.

The compare-and-swap instruction can be used to synchronize concurrent access to a shared variable, say m, in the following manner. A processor first reads the value of m into a register $r1$. It then computes a new value, which is x plus the original value, to be stored in m and stores it in register $r2$. The processor then performs a compare-and-swap($r1$, $r2$, m) operation (see Fig. 17.1). If $z = 1$ after this instruction has been executed, this means no other process has modified location m since it was read by this processor. Thus, mutually exclusive access to m is maintained. If $z = 0$, then some other processor has modified m since this processor read it. In this case, the new value of m is automatically stored in $r1$ by the compare-and-swap instruction so that this processor can retry its update in a loop.

17.6 PROCESSOR SCHEDULING

A parallel program is a task force consisting of several tasks. In processor scheduling, ready tasks are assigned to the processors so that performance is maximized. These tasks may belong to a single program or they may come from different programs. Since tasks often cooperate and communicate through shared variables or message passing, processor scheduling in multiprocessor systems is a difficult problem. Processor scheduling is very critical to the performance of multiprocessor systems because a naive scheduler can degrade performance substantially [28].

17.6.1 Issues in Processor Scheduling

The following are three major causes of performance degradation in multiprocessor systems [28]. These should be given consideration during the design of a processor scheduling scheme.

Preemption inside Spinlock-controlled Critical Sections. This situation occurs when a task is preempted inside a critical section when there are other tasks spinning

```
        r1:= m
label:  r2:= r1+x
        compare-and-swap(r1, r2, m)
        if z=0 then go to label
```

FIGURE 17.1
An illustration of the compare-and-swap instruction.

the lock to enter the same critical section. These tasks waste CPU cycles because they continue to spin locks until the preempted task is rescheduled and completes the execution of the critical section. Although the probability that a task is preempted while it is inside a critical section is very small (as critical sections are normally small), the time a task waits for a preempted process to be rescheduled is likely to be very long. Thus, the expected wait can be significant. This problem can be serious when a few large critical sections are entered frequently.

Cache Corruption. If tasks executed successively by a processor come from different applications, it is very likely that on every task switch, a big chunk of data needed by the previous tasks must be purged from the cache and new data must be brought into the cache. Initially, this will manifest itself as a very high miss ratio whenever a processor switches to another task. (Tasks from different applications are likely to have different working sets.) This problem, called *cache corruption*, can seriously degrade performance as overhead to handle cache misses can be significant.

Context Switching Overheads. Context switching entails the execution of a large number of instructions to save and store the registers, to initialize the registers, to switch address space, etc. This is a pure overhead as it does not contribute toward the progress of application tasks. (In addition, note that a context switch causes the problem of cache corruption.)

Next, several multiprocessor scheduling strategies that address the above issues in various ways are discussed.

17.6.2 Co-Scheduling of the Medusa OS

Co-scheduling was proposed by Ousterhout [19] for the Medusa operating system for Cm*. In co-scheduling, all runnable tasks of an application are scheduled on the processors simultaneously. Whenever a task of an application needs to be preempted, *all* the tasks of that application are preempted. Effectively, co-scheduling does context switching between applications rather than between tasks of several different applications. That is, all the tasks in an application are run for a timeslice, then all the tasks in another application are run for a timeslice, and so on.

Co-scheduling alleviates the problem of tasks wasting resources in lock-spinning while they wait for a preempted task to release the critical section. However, it does not alleviate the overhead due to context switching nor performance degradation due to cache corruption. The cache corruption problem may even be aggravated by co-scheduling; by the time an application is rescheduled, it is very likely that its working set has been flushed out of all the caches. Note that the designers of the Medusa operating system did not face this problem because the Cm* multiprocessor did not employ caches.

17.6.3 Smart Scheduling

Smart scheduling was proposed by Zahorjan et al. [29]. The smart scheduler has two nice features. First, it avoids preempting a task when the task is inside its critical section.

Second, it avoids the rescheduling of tasks that were busy-waiting at the time of their preemption until the task that is executing the corresponding critical section releases it. When a task enters a critical section, it sets a flag. The scheduler does not preempt a task if its flag is set. On exit from a critical section, a task resets the flag.

The smart scheduler eliminates the resource waste due to a processor spinning a lock that is held by a task preempted inside its critical section. However, it does not make any attempt to reduce the overhead due to context switching nor to reduce the performance degradation due to cache corruption.

17.6.4 Scheduling in the NYU Ultracomputer

Scheduling in the NYU Ultracomputer was proposed by Edler et al. [11] and it combines the strategies of the previous two scheduling techniques. In this technique, tasks can be formed into groups and the tasks in a group can be scheduled in *any* of the following ways:

- A task can be scheduled or preempted in the normal manner.
- All the tasks in a group are scheduled or preempted simultaneously (as in co-scheduling).
- Tasks in a group are never preempted.

In addition, a task can prevent its preemption irrespective of the scheduling policy (one of the above three) of its group. This provision can be used to efficiently implement a spin-lock (as in the smart scheduler).

This scheduling technique is flexible because it allows the selection of a variety of scheduling policies and a different scheduling technique can be used for different task groups. However, this scheduling technique does not reduce the overhead due to context switching nor the performance degradation due to cache corruption.

17.6.5 Affinity Based Scheduling

Affinity based scheduling, proposed by Lazowska and Squillante [15], is the first scheduling policy to address the problem of cache corruption. In this policy, a task is scheduled on the processor where it last executed. This policy alleviates the problem of cache corruption because it is likely that a significant portion of the working set of that task is present in the cache of that processor when the task is rescheduled. Lazowska and Squillante show that in affinity based scheduling, a task can save a significant amount of time that is normally spent in reloading its working set in the cache of its processor. Affinity based scheduling also decreases bus traffic due to cache reloading.

Affinity based scheduling, however, restricts load balancing among processors because a task cannot be scheduled on any processor. (Tasks are tied to specific processors.) Since tasks are always executed on processors for which they have an affinity, the system suffers from load imbalance because a task may wait at a busy processor

while other processors are idle. Squillante proposes and mathematically analyzes several threshold-based scheduling policies for task migration for load balancing in systems with affinity based task scheduling [23].

17.6.6 Scheduling in the Mach Operating System

In the Mach operating system, an application or a task consists of several threads. A thread is the smallest independent unit of execution and scheduling in Mach. In the Mach operating system, all the processors of a multiprocessor are grouped in disjoint sets, called *processors sets*. The processors in a processor set are assigned a subset of threads for execution. These processors use priority scheduling to execute the threads assigned to their processor set. Threads can have priority ranging from 0 to 31, where 0 and 31 are the highest and the lowest priorities, respectively. Each processor set has an array of 32 ready queues—one queue to store the ready threads of each priority. When a thread with priority i becomes ready, it is appended to the ith queue. In addition, every processor has a local ready queue that consists of the threads that must be executed only by that processor. Clearly, it is two-level priority scheduling: all the threads in a local queue have priority over all the threads in the global queue and there are also priorities inside each of these two queues.

When a processor becomes idle, it selects a thread for execution in the following manner. If the local ready queue of the processor is nonempty, it selects the highest priority thread for execution. Otherwise, it selects the highest priority thread from the global ready queues for execution. If both the queues (local and global) are empty, the processor executes a special *idle* thread until a thread becomes ready. When a thread runs out of its timeslice at a processor, it is preempted only if an equal or higher priority ready thread is present. Otherwise, the thread receives another timeslice at the processor. The length of the timeslice is variable and depends upon the number of ready threads. The higher the number of ready threads, the shorter the timeslice.

HINTS IN THE MACH OPERATING SYSTEM. The scheduler in the Mach operating system uses the concept of a *hint* to effectively schedule tasks that are believed to communicate with each other [6]. A user may have application-specific information that may help the operating system make intelligent scheduling decisions. A hint is the information in coded form, which is supplied by the user at the time of a task submission to the system. Hints essentially help modulate (elevate as well as suppress) priority and determine the timing of the execution of threads such that communication and synchronization are efficiently made between the threads. Scheduling information specific to an application (such as the senders and receivers of messages, processes synchronizing through a rendezvous, etc.) can be advantageously used to effectively carry out communication and synchronization among threads. Sometimes a hint can be a mere guess and sometimes it can be known accurately, depending on the deterministic nature of the application.

The Mach operating system supports the following two classes of hints:

Discouragement Hints. A discouragement hint allows the scheduler to delay execution of a task. It indicates that the current thread should not be run at present.

Discouragement hints can be mild, strong, or absolute. A mild hint suggests that the thread should relinquish the processor to some other thread if possible. A strong hint suggests that the thread should not only relinquish the processor, but that it should also suppress its priority temporarily. An absolute hint blocks a thread for a specific period.

Discouragement hints can be effectively used to schedule threads in an application. For example, discouragement hints can be used to optimize the performance of applications that perform synchronization through shared variables. When one thread holds the lock on a shared variable, other threads that are competing for the same lock can reduce the wastage of resources by delaying their execution using discouragement hints.

Handsoff Hints. Handsoff scheduling indicates that a specific thread should be run instead of the currently executing thread. A handsoff hint "hands off" the processor to the specified thread, bypassing the scheduler. Handsoff scheduling may designate a thread within the same task or within a different task (on the same host) that should run next.

One excellent application of handsoff hints is the *priority inversion* problem, where a low priority thread holds a resource that is needed by high priority threads. In such situations, a high priority thread can hand the processor off to the low priority thread. For example, a thread that is waiting for a semaphore to open should hand off the processor to the thread that holds the semaphore.

17.7 MEMORY MANAGEMENT: THE MACH OPERATING SYSTEM

In this section, we explain memory management in multiprocessor operating systems by studying the virtual memory management of the Mach operating system, developed at Carnegie Mellon University. We discuss issues in the design of memory management and describe how the Mach operating system addresses these issues. The discussion of the Mach virtual memory system in this section is based on the work of Tevanian [25].

17.7.1 Design Issues

Portability. Portability implies the ability of an operating system to run on several machines with different architectures. The virtual memory system is a component of the operating system that heavily relies on the idiosyncracies of the underlying architecture and can thus be an impediment to the portability of the operating system. For wide spread applicability of an operating system, architecture-independence should be an important consideration in the design of a virtual memory system.

Data Sharing. In multiprocessor systems, an application is typically executed as a collection of processes that run on different processors. These processes generally share data for communication and synchronization. A virtual memory system must provide a facility for flexible data sharing to support the execution of parallel programs.

Protection. When memory is shared among several processes, memory protection becomes an important requirement. The operating system must support mechanisms that

a virtual memory system can employ to protect memory objects against unauthorized access.

Efficiency. A virtual memory system can become a bottleneck and limit the performance of the multiprocessor operating system. A virtual memory system must be efficient in performing address translations, page table lookups, page replacements, etc. Moreover, it should run in parallel to take advantage of multiple processors.

The Mach operating system is designed for parallel and distributed environments. It can run on multiprocessor systems and support the execution of parallel applications. In fact, the Mach operating system itself is designed to run in parallel—all algorithms are designed to run in parallel and all the data structures are designed to allow highly parallel access. The implementation of the virtual memory system is fully parallel in Mach to exploit the parallelism in multiprocessor systems. The Mach virtual memory system provides flexible data sharing and protection primitives to support high performance parallel applications.

17.7.2 The Mach Kernel

A key component of the Mach operating system is the Mach kernel, which provides only the basic primitives necessary for building parallel and distributed applications. It provides primitives for process management, memory management, interprocess communication, and I/O services. Other operating system services, which are useful to developers or end users, are built on top of the Mach kernel (Fig. 17.2). Since the Mach kernel provides only a small number of simple services and because only a few decisions are made within the Mach kernel, it is readily adaptable and portable to a wide array of architectures. A number of operating systems can be built on the Mach kernel as user programs.

The Mach kernel supports five abstractions: threads, tasks, ports, messages, and memory objects.

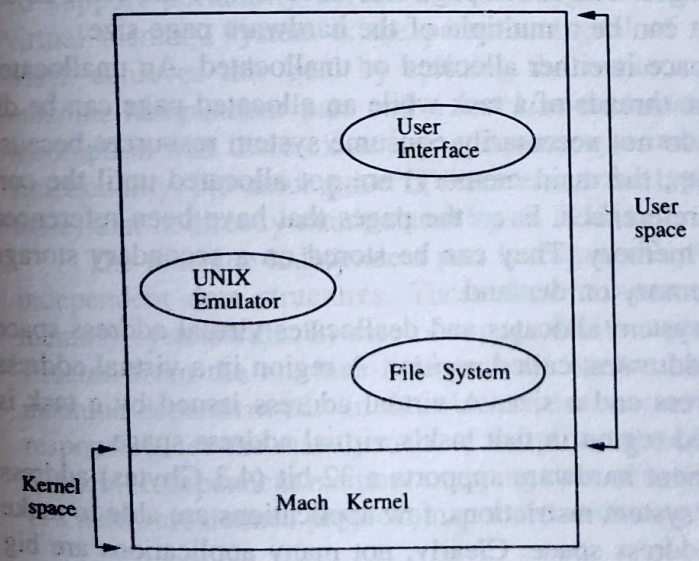

FIGURE 17.2
Mach operating system.

Tasks and Threads. A thread is the smallest independent unit of execution in Mach. A thread has a program counter and a set of registers. A task is an execution environment that may consist of many threads. A task includes a paged virtual address space and protected access to the system resources. A task is the basic unit of resource allocation.

Messages and Ports. A message is a typed collection of data used by threads for communication. Messages may be of an arbitrary size and can contain pointers and capabilities. A port is a unidirectional channel associated with an object (e.g., task, thread) that queues up messages for that object. A port can be viewed as a queue of messages. Tasks and threads communicate with other tasks and threads by performing send and receive operations on their ports. A port is protected in the kernel to ensure that only authorized tasks or threads can read or write to a port.

Memory Objects. A *memory object* is a contiguous repository of data, indexed by byte, upon which various operations, such as read and write, can be performed. Memory objects act as a secondary storage in the Mach operating system. Mach allows several primitives to map a virtual memory object into an address space of a task. Data in a memory object become available for direct access in an address space after the object or its part has been mapped into the address space. In Mach, every task has a separate address space. An address space consists of a collection of memory objects that are mapped into it. The kernel acts as a cache manager for memory objects where the physical memory is treated as the cache memory. A reference to data in a memory object that is not present in the physical memory causes a page fault and is translated into a page request.

17.7.3 Task Address Space

In the Mach operating system, each task is assigned a single paged-address space. The size of the address space is limited by the addressing capabilities of the underlying hardware (e.g., the size of *memory address register* of the processor). Mach treats an address space as a sequence of pages. The Mach page size need not be the same as the underlying hardware page size; it can be a multiple of the hardware page size.

A page in a task address space is either allocated or unallocated. An unallocated page may not be addressed by the threads of a task while an allocated page can be directly accessed. Allocated pages do not necessarily consume system resources because pages in the physical memory (i.e., the main memory) are not allocated until the corresponding virtual addresses are referenced. Even the pages that have been referenced need not be present in the main memory. They can be stored on a secondary storage and are brought into the main memory on demand.

The Mach virtual memory system allocates and deallocates virtual address space in contiguous chunks of virtual addresses, called *regions*. A region in a virtual address space is specified by a base address and a size. A virtual address issued by a task is valid only if it falls in an allocated region in that task's virtual address space.

A typical memory management hardware supports a 32-bit (4.3 Gbytes) address space. However, due to operating system restrictions, few applications are able to make use of the entire 4.3 Gbytes of address space. Clearly, not many applications are big

enough to use the entire address space. Nonetheless, there are applications that benefit from using a large address space sparsely; that is, they have a large address space at their disposal but use only a small fraction of it. For example, the extensive use of a mapped file may fragment the address space, creating large holes when a file is deleted. The Mach virtual memory system supports such large, sparse address spaces.

The Mach virtual memory system supports several operations that are often needed in advanced applications. For example, a thread can normally access only the address space of the task in which it executes. However, it is sometimes necessary for a task to read or write the address space of other tasks. For example, a debugger needs to examine and modify the address space of the task being debugged. The Mach virtual memory system provides primitives to perform these operations. In addition, it provides several other primitives, such as primitives to efficiently copy a region within an address space, primitives to query current virtual memory statistics maintained by the kernel, etc.

17.7.4 Memory Protection

Virtual memory protection is enforced at the page level. Each allocated page has the following two protection codes associated with it. (1) The *current* protection code, which corresponds to the protection associated with a page for memory references and (2) the *maximum* protection code, which limits the value of the current protection. A page's protection consists of a combination of read, write, and execute permissions. Each type of permission is mutually exclusive. Mach provides primitives that set the current or maximum protection. The current protection can only include the permissions specified in the maximum protection. The maximum protection can only be lowered. That is, permissions specified in the maximum protection can be deleted, but new permissions cannot be added.

17.7.5 Machine Independence

To support portability across a wide range of architectures, a machine-independent virtual memory system is the paramount goal of the Mach virtual memory system. Mach achieves this goal by splitting the implementation into two parts, namely, a *machine-independent* part and a *machine-dependent* part. This split is based on the assumption that there exists a paged *memory management unit* (MMU) with minimal functionality. No assumption is made about the type of data structure (such as a page table) that is directly manipulated by an MMU.

The machine-independent part is responsible for maintaining high level machine-independent data structures. These data structures represent the state of the virtual memory systemwide. In case of a page fault, entire mapping information can be constructed from the machine-independent data structures. The *pmap module* is the only machine-dependent part in the Mach virtual memory system implementation, and it is responsible for the management of the physical address space. This module consists of a machine-dependent memory mapping data structure, called the *pmap structure*, which is a hardware defined physical map that translates a virtual address to a physical address.

A pmap structure corresponds to a page table. All machine-dependent mapping is performed in the pmap module. The pmap module executes in the kernel and implements page level operations on pmap structures. It ensures that the correct hardware map is in place whenever a context switch takes place. It is reponsible for managing the MMU and setting up hardware page tables. Clearly, the pmap module depends upon MMU architecture and must be recoded for a new multiprocessor system architecture.

The interface of the pmap module assumes the existence of a simple, paged MMU architecture and it has been designed to support a wide variety of MMUs. The pmap module also deals with any discrepancy between operating system page size and the underlying hardware page size. The implementation of pmap module need not know any details of the machine-independent implementation and data structures. The pmap module provides an interface (i.e., a set of primitives) to the machine-independent part that are used by the machine-independent part to notify the machine-dependent part of any changes in the mapping, creation and destruction of address spaces, etc. This information is used by the pmap module to appropriately set up hardware page table registers and other machine specific hardware registers that are related to memory management.

In addition, the Mach virtual memory system provides two types of independence to higher layers: operating system independence and paging-store independence.

OS Independence. The Mach virtual memory system is implemented such that it is almost completely decoupled from the rest of the system. It makes few assumptions about other kernel functions and is easily adaptable to different systems. Also, the virtual memory system implementation has clean interfaces to the rest of the system.

Paging-Store Independence. The Mach virtual memory system assumes no knowledge of secondary storage systems for paging purposes. Instead, the Mach virtual memory implementation defines a simple *pager interface* to which any client may conform. An external pager is responsible for managing the secondary storage. It keeps track of which pages in the virtual address space are in the main memory and where the pages in the virtual address space are located on the secondary storage.

17.7.6 Memory Sharing

The ability to share memory among several tasks is very important for the efficient execution of parallel applications. These applications can use shared memory for efficient process synchronization and interprocess communication. Without these facilities, parallel applications must use expensive synchronization primitives that have high overhead.

In Mach, all the threads of a task automatically share all the memory objects that reside in the address space of the task. Different tasks can share a page (or a memory object) by installing that page in their virtual address spaces and by initializing entries in their page tables so that all references to a virtual address in the shared page are correctly translated into a reference to a physical page. Although a shared page may be mapped at different locations in the virtual address space of the tasks, only one copy of the page is present in the main memory.

The Mach virtual memory system allows the sharing of memory via the *inheritance* mechanism. In Mach, a new address space is created when a task is created. The new address space can either be empty or it can be based on an existing address space. When a new address space (child) is based on an existing address space (parent), a page in the new address space is based on the value of the inheritance attribute of the corresponding page in the existing address space. The inheritance attribute of a page can take three values: *none*, *copy*, and *share*. If a page is in the none inheritance mode, the child task does not inherit that page. If a page is in the copy mode, the child receives a copy of the page and subsequent modifications to that page only affect the task making the modifications. If a page is in the share mode, the same copy of the page is shared between the parent and the child tasks. Consequently, all subsequent modifications to that page are seen by both the tasks.

In addition to supporting shared memory via the inheritance mechanism, if interfaces provided by a host export primitives that permit more unrestricted memory sharing, Mach will allow this form of memory sharing. Thus, unrestricted memory sharing can be supported within a Mach host.

17.7.7 Efficiency Considerations

The Mach virtual memory system uses the following techniques to increase efficiency:

Parallel Implementation. The implementation of the Mach virtual memory system is fully parallel to exploit the parallelism in multiprocessor systems—all algorithms are designed to run in parallel and all data structures are designed to allow highly parallel access.

Simplicity. The underlying Mach philosophy is to use simple algorithms and data structures because complex algorithms and data structures are likely to waste CPU cycles without much performance improvement.

Lazy Evaluation. In lazy evaluation, the evaluation of a function is postponed as long as possible in the hope that the evaluation will never be needed. The Mach virtual memory system makes extensive use of lazy evaluation to increase time and space efficiency. For example, virtual to physical mapping of a page is postponed until it is actually needed (i.e., a page reference occurs). The Mach virtual memory system does not even allocate space to page tables until they are needed. Thus, Mach postpones the creation of page tables and the lookup of disk addresses until needed.

THE COPY-ON-WRITE OPERATION. Copy-on-write is a succint example of lazy evaluation in Mach that optimizes memory space and CPU cycles. The copy-on-write operation postpones the actual copying of a data page until the copied page is written. When two tasks, A and B want to share a page, the system allows them to share the same copy of the physical page, but each process has read-only access to the page (See Fig. 17.3). When task B attempts to write into the page, a protection fault is generated and the page is copied into a new physical page and a new virtual mapping is set up for the newly created page. Now B has a separate physical copy of the page.

Copy-on-write optimization improves efficiency in a variety of ways. It reduces memory overhead because several pages that are copied may never be written. No CPU

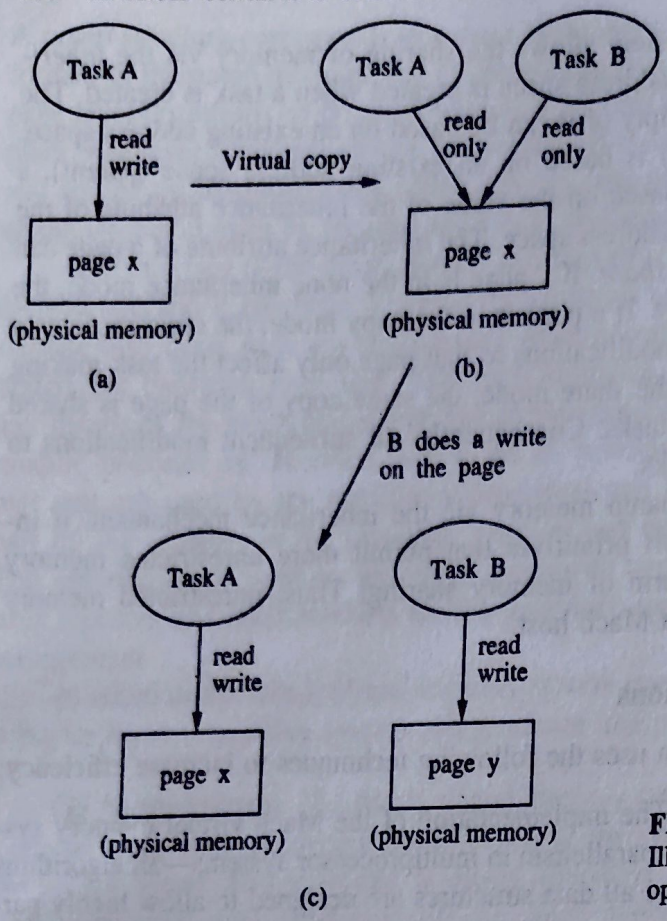

FIGURE 17.3
Illustration of the copy-on-write operation.

time is wasted in copying pages that are never written. If copied pages are never accessed, these pages do not incur mapping overhead because mapping is lazily evaluated.

17.7.8 Implementation: Data Structures and Algorithms

Data Structures

The Mach virtual memory system uses four basic data structures: memory objects, pmap structures, resident page tables, and address maps. Recall that a memory object is a repository of data that can be mapped into the address space of a task and a pmap structure is a hardware defined physical map that translates a virtual address to a physical address. Below we discuss resident page tables and address maps.

Resident Page Tables. The Mach operating system treats the physical memory as a cache for virtual memory objects. Information about physical pages (e.g., whether they are modified, referenced, etc.) is maintained in a page table (called a *resident page table*) whose entries are indexed by physical page number. A page entry in the page table may be linked into the following lists. (1) *Memory object list*: all page entries associated with an object are linked together as a memory object to speed up object deallocation and virtual copy operations. (2) *Memory allocation queues*: queues are maintained for free, reclaimable, and allocated pages and are used by the Mach pager

to determine a free page in the case of page fault. (3) *Object/offset hash bucket*: Fast lookup of a physical page associated with an object/offset, at the time of a page fault, is done using a bucket hash table (keyed by the memory object and byte offset). Byte offsets in memory objects are used to avoid binding the implementation to a particular physical page size.

Address Maps. An address map is a data structure that maps contiguous chunks of virtual addresses (i.e., memory regions) in the address space of a task to memory objects. An address map is a doubly linked list of *address map entries*, each of which maps a contiguous range of virtual addresses in the address space of a task onto a contiguous area of a memory object. Each address map entry contains byte offsets of the beginning and end of the region represented by it. The linked list is sorted in the ascending order of virtual addresses. Each address map entry contains information about the inheritance and protection attributes of the memory region it defines. Thus, all addresses (pages) within a memory region mapped by a map entry have the same attributes. The address map data structure permits the efficient implementation of the most frequently performed operations on the address space of a task, namely, page fault lookups, copy/protection operations on a memory region, and the allocation and deallocation of memory regions. An address map allows us to perform operations on memory regions simply and quickly. Also, an address map allows for the efficient maintenance of sparse address spaces.

Algorithms

The Page Replacement Algorithm. A page replacement algorithm decides which page in the physical memory to replace in the event of a page fault. Mach philosophy is to use a simple page replacement algorithm because a complex algorithm is likely to waste CPU cycles without vastly improving performance. The replacement algorithm in Mach is a modified-FIFO algorithm that keeps all the physical memory pages in one of the following three FIFO queues:

The free list. This contains pages that are free to use. These pages are not currently allocated to any task and can be allocated to any task.

The active list. This contains all pages that are *actively* in use by tasks. When a page is allocated, it is removed from the free list and placed at the end of the active list.

The inactive list. This contains pages that are not in use in any address space, but were recently in use. These are the pages that will be freed if they are not referenced soon.

A special kernel thread called a *pageout daemon* performs page replacement and management of these lists. In the event of a page fault, the daemon performs page replacement by taking a page from the inactive list and placing it in the free list. (The same action is taken when the page count is low in the free list.) The pageout daemon always maintains a small number of pages in the inactive list by moving pages from

the active list to the inactive list (and then removing mapping to those pages). The page replacement algorithm is a FIFO algorithm except that a page in the inactive list is activated if a task makes a reference to it. Thus, the inactive list serves as a second chance for pages targeted for replacement.

The Page Fault Handler. The page fault handler is invoked when a page is referenced for which there is either an invalid mapping or a protection violation. The page fault handler has the following responsibilities. (1) *Validity and protection:* it determines if the faulting thread has the desired access to the address by performing a lookup in its task's address space. (2) *Page lookup:* it attempts to find an entry for a cached page in the virtual to physical hash table. If the page is not present, the kernel requests the data from the pager. (3) *Hardware validation:* It informs the hardware physical map (i.e., the pmap module) of the new virtual to physical mapping.

Locking Protocols. All algorithms and data structures used in virtual memory implementation are designed to run in a multiprocessor environment and are thus fully parallel. The synchronization of accesses to shared data structures is achieved by the following locks. (1) *Map locks:* map locks provide exclusive access to address map data structures. (2) *Object locks:* object locks guarantee exclusive access to physical memory resources cached within an object. (3) *Hash table bucket locks:* these locks provide proper access to the object/resident page table hash table on a per bucket basis. (4) *Busy page locks:* these locks are used to indicate that some operation is pending on a given physical page. To prevent deadlocks, all algorithms acquire locks in the same order, i.e., map locks, object locks, and then either bucket or busy page locks.

17.7.9 Sharing of Memory Objects

Mach supports copy-on-write operations, which allow the sharing of the same copy of a memory object by several tasks as long as all the tasks only read the memory object. When a task performs a copy-on-write operation on a memory object, its address map starts pointing at the original copy of the memory object; that is, it shares the same copy with the original owner of the memory object.

If one of the tasks writes data in a copied memory object using the copy-on-write operation, a new page for that data is allocated, which is accessible only to the writing task. This new page contains the modifications by the writing task. Note that a separate copy is created only for the pages of a memory object that have been modified. Mach maintains special objects, called *shadow objects*, to hold pages of a memory object that have been modified. A shadow object collects and remembers all the modified pages of a memory object copied/shared using the copy-on-write operation. A shadow object typically does not contain all the pages of the region it defines. It relies on the original object for unmodified pages. A shadow object can itself be shadowed on subsequent copy-on-write operations, thus creating a chain of shadows.

Note that if the address maps of all the tasks that share a memory object using copy-on-write directly point to the shared memory object, then the mapping and remapping of the shared memory objects will require the manipulation of all the address

maps. Yet, many tasks may share the same region of memory in read/write mode and may simultaneously share the same region with other tasks in copy-on-write mode. To circumvent these problems, a level of indirection is introduced when accessing shared memory objects. An address map, called a *sharing map*, points to a shared object, which in turn is pointed to by the address map entries of all the tasks sharing that memory object.