CHAPTER 4

The Operating System Kernel: Implementing Processes and Threads

The process model is fundamental to operating system (OS) design, implementation, and use. Mechanisms for process creation, activation, and termination, and for synchronization, communication, and resource control form the lowest level or **kernel** of all OS and concurrent programming systems. Chapters 2 and 3 described these mechanisms abstractly from a user's or programmer's view, working at the application level of a higher level of an OS. However, these chapters did not provide details of the internal structure, representations, algorithms, or hardware interfaces used by these mechanisms.

In this chapter, we present a more complete picture, discussing, for example, how a process is blocked and unblocked. We start with an overview of possible kernel functions, objects, and organizations. The remaining sections are concerned with implementation aspects of the kernel.

First, we outline the various queue data structures that are pervasive throughout OSs. The next two sections elaborate on the most widely used adaptation of processes, namely, **threads**, and show how processes and threads are built. Internal representations and code for important interaction objects, including semaphores, locks, monitors, and messages are then discussed; a separate subsection on the topic of timers also appears. The last section presents the lowest-level kernel task, *interrupt handling*, and illustrates how this error-prone, difficult function can be made to fit naturally into the process model.

4.1 KERNEL DEFINITIONS AND OBJECTS

The OS kernel is a basic set of objects, primitive operations, data structures, and processes from which the remainder of the system may be constructed. In one appealing view, the purpose and primary function is to transform the computer hardware into an OS's "machine"—a computer that is convenient for constructing OSs. Thus, in a layered functional hierarchy (Chapter 1), the kernel is the lowest level of software, immediately above the hardware architectural level.

Compatible with this view is the notion that the kernel should define or provide *mechanisms* from which an OS designer can implement a variety of *policies* according to application or client desires. The mechanism-policy distinction is important. For example, semaphores and monitors are generally considered mechanisms, whereas resource-allocation schemes that use these mechanisms, such as storage or buffer allocators, are OS policies. A scheduler that allocates CPUs to processes according to given priorities is a mechanism; how and on what basis to select the process priorities are policy decisions. The difference between the two ideas is not always unambiguous. An example is the queue removal operation for a semaphore *mechanism*; it normally also incorporates a *policy* decision, such as either first-come/first-served or priority-based.

What constitutes a kernel is a matter of definition. In one view, the part of the OS that resides permanently in main memory is called the kernel. Because the OS must be protected from user programs, most of it, especially the kernel, runs under a form of hardware protection, such as supervisor mode. This leads to another pragmatic definition of the kernel as that part of the OS that runs in a protected mode. Yet another variation combines both of the above definitions. In this view, the kernel includes any part of the system that resides permanently in main memory and runs in a supervisory mode.

The set of possible functions and objects in a kernel can be divided into four classes:

1. *Process and thread management:* Process creation, destruction, and basic interprocess communication and synchronization.
2. *Interrupt and trap handling:* Responding to signals triggered by various system events; among these are the termination of a process, completion of an I/O operation, a timer signal indicating a timeout or clock tick, request for a service to be performed by the OS, an error caused by a program, or hardware malfunctioning.
3. *Resource management:* Primitives for maintaining, allocating, and releasing units of resources such as CPUs, timers, main memory, secondary storage, I/O devices, or files.
4. *Input and output:* Read, write, and control operations for initiating and supervising the transfer of data between I/O devices and main memory or registers.

OS kernels can be quite small, so-called **microkernels**, consisting of most of the first two classes and some limited resource management, usually processor scheduling and low-level virtual memory allocation. Windows NT (Solomon 1998) and Mach (Accetta et al. 1986) are examples of systems with such microkernels. At the other extreme, almost all of the functions listed in the four classes above can be incorporated into a very large kernel. Most versions of UNIX, as well as Linux, have monolithic kernels.

In this chapter, we are concerned primarily with the first two types of kernel operations, i.e., those for process and thread management, and interrupt handling; the next chapter covers CPU scheduling. The discussion of I/O and the management of other important resources is the subject of later chapters.

A highest-level user process is usually established in response to a request expressed in the system's control or command language; a standard request of this sort is a "login" command. We will assume that such a highest-level supervisory or "envelope" process, say p_j, is created for each user j. In many systems, this is actually the case. Where it is not, it is still convenient conceptually to assume the existence of such a process,

even though its function may be performed centrally by a systems process. The process p_j has the responsibility for initiating, monitoring, and controlling the progress of j's work through the system as well as for maintaining a global accounting and resource data structure for the user. The latter may include static information such as user identification, maximum time and I/O requirements (for batch processing), priority, type (e.g., interactive, batch, real-time), and other resource needs. It also may include dynamic information related to resource use and other processes created as children.

In general, p_j will create a number of such child processes, each corresponding to a requested unit of work. These processes may in turn create additional ones, either in sequence or in parallel. Thus, as a computation progresses, a corresponding tree hierarchy of processes grows and shrinks. The processes are not totally independent, but interact with each other and parts of the OS such as resource managers, schedulers, and file system components, using synchronization and communication primitives provided by the kernel.

These ideas are illustrated in Figure 4-1, where an OS process p_s has created user processes $p_1, \ldots, p_n$, and the process p_j, in turn, has created the processes $q_1, \ldots, q_m$, and so on. All processes use the primitives provided by the underlying kernel. The remainder of

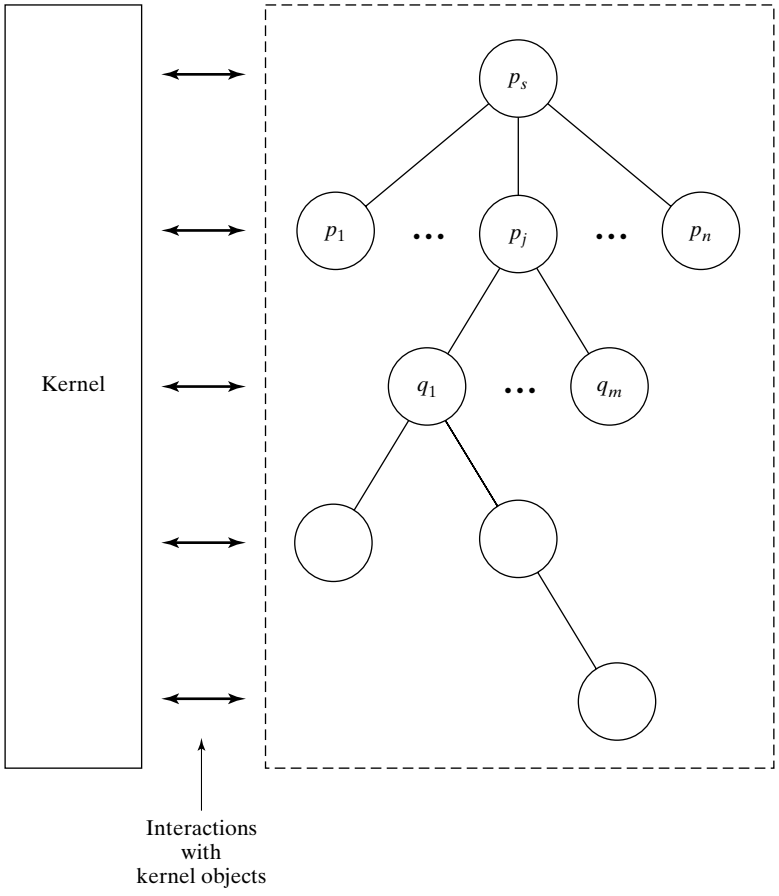

FIGURE 4-1. A process creation hierarchy.

this chapter will discuss the internal representation of processes and the implementation of operations on processes and resources.

4.2 QUEUE STRUCTURES

Every kernel and higher-level object in an OS is normally represented by a data structure containing its basic state, identification, accounting, and other information. Thus, there are data structures for each active object, such as a process or a thread, for each hardware and software resource, and for each synchronization element. These data structures are referred to as system **control blocks** or **descriptors**. Collectively, they represent the state of the OS and, as such, are accessed and maintained by systems routines. Later sections and chapters discuss the possible organizations and contents of particular object descriptors. This section focuses on queues—their central role at the kernel and higher OS level, and their implementations.

4.2.1 Resource Queues in an Operating System

A popular practical view considers an OS to be a queue processor. The active objects of the system move from one resource queue to another, triggered by interrupts and requests for service. A user entity might originate in a long-term queue, eventually being served by a long-term scheduler that activates or swaps in the object, sending it to a short-term scheduling queue. The latter is typically a **ready list** of processes waiting for allocation of a CPU by a scheduler.

During execution, an object will request other resource elements, and enqueue itself on servers or allocators for that resource. Thus, there are queues associated with managers of hardware such as main memory and secondary storage, I/O devices, and clocks. There are also queues for shared software components, such as files, buffers, and messages, and queues for low-level synchronization resources such as semaphores, locks, and monitors. In fact, it is no exaggeration to say that queues are the primary generic data structures of OSs. Many of these queues appear at the kernel level and are the descriptor representations for the kernel objects.

A variety of schemes or servicing policies are needed. The simplest, most efficient, and most common is *first-come/first-served* (FIFO); requests are considered and satisfied in the order of arrival. The bounded-buffer example developed throughout Chapter 3 uses a queue with exactly this policy. At the other extreme are *priority-based* schemes. Typically, a numerical priority is attached to each object in a queue, and the queue is serviced in priority order. For example, processes may be given different priorities according to their importance: Processes that are CPU-bound often have a lower priority for the CPU than processes that are I/O-bound. Processes with short time deadlines typically have higher CPU priorities than those with longer deadlines. Objects waiting on CSs, semaphores, or monitors might require priority waits, such as in the elevator algorithm described in Chapter 3. Priority-based queues are a very general and versatile mechanism, and subsume many other simpler schemes as special cases. For example, FIFO queues are also priority-based, where the priority depends directly on arrival time. But for pragmatic reasons—ease of implementation or areas of applicability—they are typically viewed as distinct structures.

Specific examples of OS queues are presented in the remainder of the book. The next section describes basic queue implementations usable by kernels and higher-level OS objects.

4.2.2 Implementations of Queues

We consider the two basic types of queues mentioned above: FIFO and priority based. To maintain these queues, at least two operations are needed, one to append a new element x to a queue Q and one to remove the first element from a given queue. Common forms for these operations are:

```
insert(queue Q, element x)
and
remove(queue Q, element x).
```

These are similar to the *deposit* and *remove* commands defined for the bounded buffer. (An alternative for the *remove* defines it as a function that returns an element, with the side effect of deleting the element from the queue; functions with side effects are generally not considered good programming practice.)

It is also convenient to provide a test for emptiness of a given queue; its form is:

```
int empty(queue Q)
```

This returns 1 if the queue is empty and 0 otherwise. Sometimes, it is also useful to define operations that insert an element into or remove an element from an arbitrary place in a queue structure, violating the basic queue properties. This is a practical compromise for those situations where a data structure is a queue most of the time, but not always. For simplicity, we will use the same notation *insert/remove* for such operations.

In addition to the queue elements themselves, most queues also contain a **header** or more global descriptive data. This may include some generic information about the queue, such as its name, length, and history of use, as well as application- or resource-specific data. Examples of the latter might be maximum and minimum message lengths in a message queue, the number of free and allocated blocks of memory for a memory queue, a list of allowed users of the queue, and the maximum amount of time allocated to all processes waiting in a ready list (queue) for a CPU.

Single-Level Queues

A bounded array is often the most convenient and efficient way to construct a FIFO queue, especially at the kernel level. A diagram of a typical implementation appears in Figure 4-2a. The queue elements are stored in a "circular" array of size n, with two moving pointers, *front* which points to the head of the queue (the first element to be removed) and *rear* (the last element that was deposited). Both pointers are advanced to the right, modulo n, which implements the circular nature of the queue. This is essentially the data structure assumed for the bounded buffer in Chapter 3.

Operations on the array implementation are fast and deterministic. However, there are two negative features. The first is the boundedness. We must know, or check at every *deposit* call, that the queue does not overflow beyond its capacity of n. Similarly, we must check at every *remove* that the queue does not underflow below zero. An additional problem with arrays is that it is an awkward form if we must insert or delete an element at an arbitrary place in the structure.

A linked list implementation is more flexible but not as efficient. A singly linked version is sketched in Figure 4-2b. Such a list is, in principle, unbounded in length,

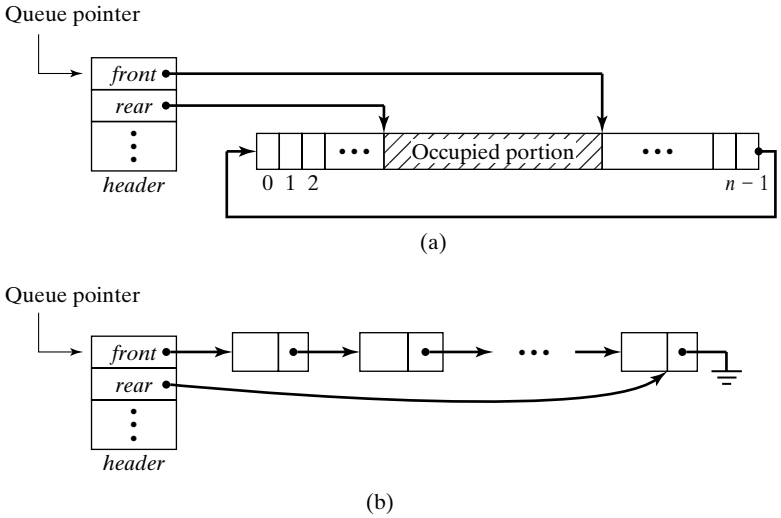

FIGURE 4-2. FIFO queues: (a) circular array implementation; and (b) linked list implementation.

and thus, able to accommodate long bursts of insertions. However, length checking may be necessary, since kernel space is usually limited and node storage must be obtained dynamically. Insertions and deletions at arbitrary spots are also easy to handle, especially if each node is doubly linked.

Priority Queues

A priority queue combines several simple queues into one common data structure. In addition to its application data, each queue element contains a priority associated with the corresponding object. Typically, the priorities are positive integers with lower numbers treated as higher or lower priorities, depending on the implementation. The *remove* operation is obligated to delete and return the highest-priority queue element.

A common case is when there are a *fixed* and relatively small number of priorities, say n, numbered from 0 to $n - 1$. One particularly convenient organization starts with an **array of** n **headers** of FIFO queues, one header for each priority. As shown in Figure 4-3a, each header points to a linked list of elements queued at that priority; alternatively, and at greater storage cost, each queue could be represented as a circular array. In either case, inserting an element with a given priority can be accomplished quickly in constant time. Similarly, a removal is implemented by just scanning the header array from the top down for the first nonempty queue. By keeping the number of the highest-priority, nonempty queue in the global header, a removal can be performed more quickly. With the linked-list version, it is also relatively easy to insert or delete an element at an arbitrary location in the queue, or to change the priority of some entry.

A data structure that works well when priorities are not fixed but can range over a large domain of values is the **binary heap** (e.g., [Weiss 1993]). Examples of such priorities are those that are directly based on time, such as deadlines, or on space, such as storage addresses (e.g., disk cylinder) or numbers of memory blocks. A binary heap

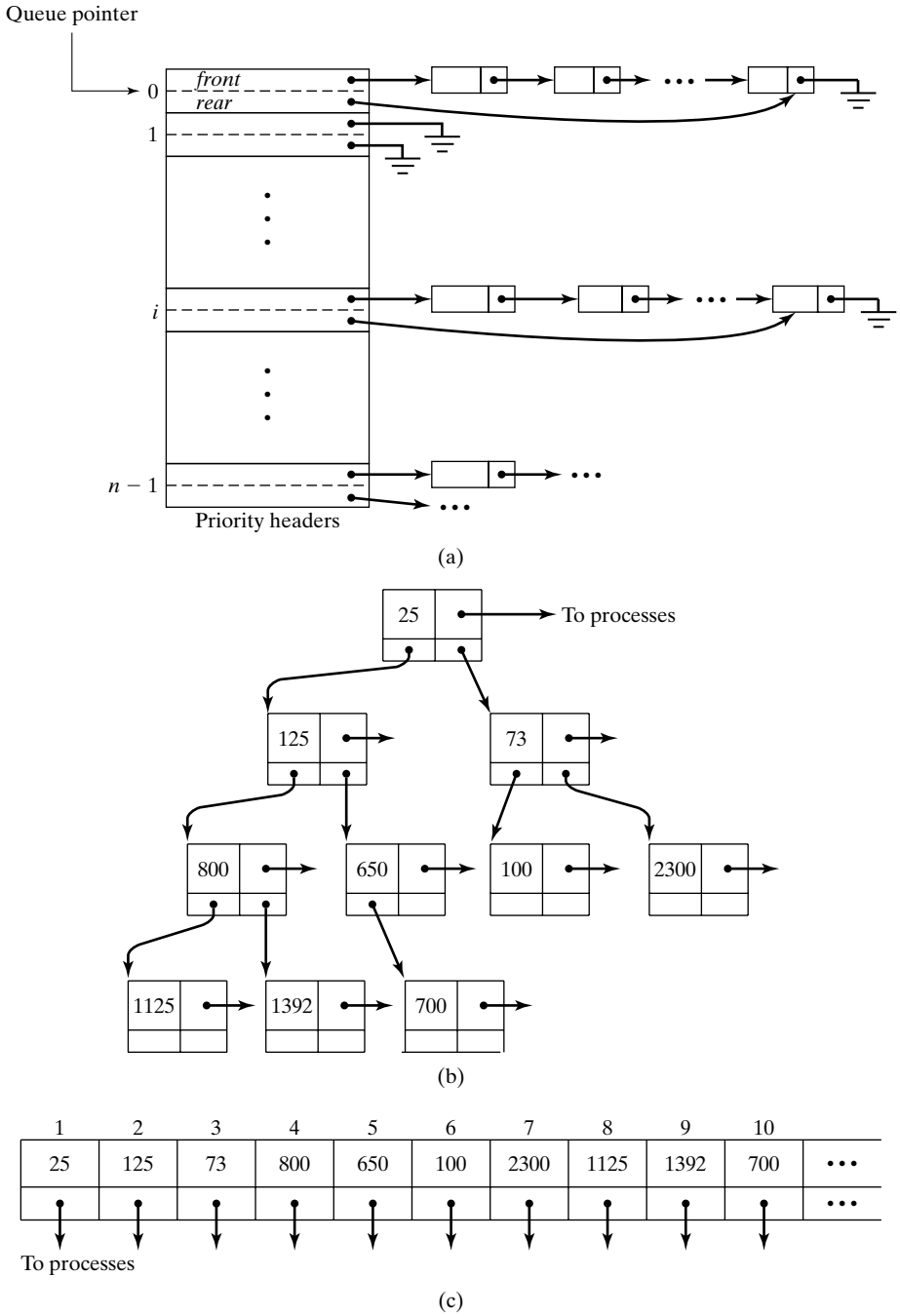

FIGURE 4-3. Priority queues: (a) array indexed by priority; (b) binary heap of priorities; and (c) array implementation of binary heap.

is a complete binary tree where the value (priority number in our case) at the root of any subtree is always smaller than the value of any other node in the subtree. Figure 4-3b shows an example of a binary heap with 10 different priority values. Each element in the heap consists of the priority value, the two pointers to its left and right subtree, and a pointer to the process (or list of processes) at that priority level.

A binary heap is also suitable for kernel-level priority queues because it can be implemented efficiently as an array. The tree root is placed at index 1 of the array. The left and right subtrees are stored recursively using the following rule: The left subtree of any node i starts at index $2i$, and the right subtree starts at index $2i + 1$. Figure 4-3c shows the array representation of the binary heap of Figure 4-3b.

Insertions and removals each take $O(\log n)$ time; thus, these operations are slightly less deterministic and efficient than the corresponding ones in a fixed-priority array implementation. However, the range of possible priority value is open-ended.

4.3 THREADS

The normal implementation of processes, especially traditional UNIX processes where each process has a single thread of execution, results in much runtime overhead—creation, termination, synchronization, and context-switching are all lengthy operations. To alleviate this problem, various user-level packages have been developed that allow multiple "lightweight" scheduling units to be implemented within a process. These units share the same resources as their host process; notably, they have the same address space. But they can be created, scheduled, and synchronized more efficiently.

A similar approach also allows resource sharing among concurrent entities but implements their scheduling as part of the OS rather than a user package. The most extreme version appeared in the Pilot OS, an early and influential single-user system that contained one process but many schedulable threads of control (Redell et al. 1980). Pilot

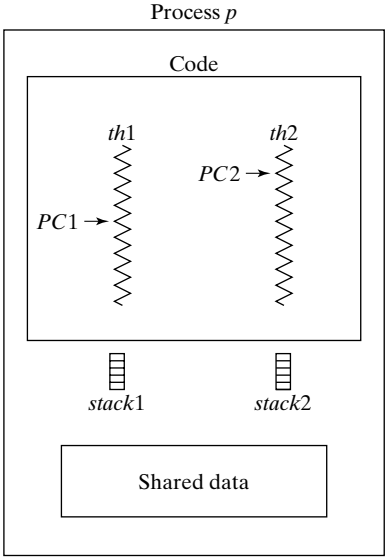

FIGURE 4-4. Threads within a process.

was written in the Mesa language and was probably the first commercial OS written using monitors. The Mesa form of threads was called a "lightweight process."

The general idea of efficiently sharing resources, especially address spaces, among related scheduling blocks was so successful that most contemporary systems provide this facility. The scheduling units are known as **threads**. Processes still exist, but these are now *passive resource holders* for threads. In this popular model, a process contains one or more active threads.

Figure 4-4 illustrates the relationship between processes and threads. It shows two threads, $th1$ and $th2$, executing within the same code segment of a process p. Each thread is uniquely characterized by its own program counter (PC_i) and function-calling stack ($stack_i$). The process data area is shared by the two threads.

CASE STUDY: USER-LEVEL AND KERNEL-LEVEL THREAD FACILITIES

Windows NT (Solomon 1998), the POSIX interface prescribed for UNIX (POSIX, 1993), and the Mach OS (Accetta et al. 1986), all support threads. The Windows NT approach is particularly interesting because it supports both kernel- and user-level threads. An NT process consists of the executable program code, data, address space, resources such as files and semaphores, and at least one kernel-level thread. Each thread is an active executable object whose state or context is maintained by the kernel. The state consists of the current values of its CPU registers, a user- and kernel-level stack, and a private systems storage area. NT threads are scheduled by the OS kernel according to the priority specified by their owning process.

Because each context switch between threads requires a call to the kernel, these threads are still relatively expensive. To solve the problem, Windows NT also supports a user-level version of threads, called **fibers**. Each kernel-level thread may define any number of fibers, and each fiber may be scheduled for execution. But the context switch between fibers is not visible to the kernel; it must be done explicitly by a function call *SwitchToFiber*, which causes the currently executing fiber to give up the CPU to the new fiber specified in the call.

Because the threads in a process share resources, they can communicate efficiently, for example, through shared memory. Threads within the same group can be created, terminated, preempted, blocked, and scheduled quickly. However, the price paid for this performance gain is a loss of protection. Standard traditional processes that do not share resources with other active objects have hardware and software guards that automatically check against or prevent accidental or malicious interferences. For example, a process cannot normally access the address space of another process. In contrast, threads belonging to the same process can easily interfere with one another since they do share the same virtual (and real) space.

It is also the case that some applications lend themselves more readily to tightly coupled threads than to more independent processes. Good examples appear in data or file sharing. Here, two related objects that must access the same data more or less concurrently can conveniently be represented by two threads; this might occur in a parallel compiler where a lexical analyzer and a syntax analyzer both use a common symbol table. Similarly, a popular service, such as a file server, can be parallelized

conveniently with threads since each concurrent object shares the same file space and, often, the same code.

Operations on threads are analogous, and in many cases, identical to those discussed for pure processes. Thus, an OS typically offers services or commands to:

- Create a new thread;
- Initiate or make a thread ready;
- Destroy or terminate a thread;
- Delay or put a thread to sleep for a given amount of time;
- Synchronize threads through semaphores, events, or condition variables;
- Perform lower-level operations, such as blocking, suspending, or scheduling a thread.

4.4 IMPLEMENTING PROCESSES AND THREADS

As noted in Section 4.2, every process and thread is represented by a data structure containing its basic state, identification, and accounting information. These system control blocks or descriptors are accessed and maintained by OS routines and collectively represent part of the state of the OS. As such, they are accessed and maintained by systems routines. This section describes the contents and organization of these descriptors, followed by the design of their operations.

4.4.1 Process and Thread Descriptors

First assume that we are dealing with a system composed solely of processes; much of the discussion will also apply to threads. A process descriptor or **process control block** (PCB) is constructed at process creation and represents the process during its existence. Process descriptors are used and modified by basic process and resource operations, interrupt routines, and schedulers. They also may be referenced during performance monitoring and analysis. Figure 4-5 shows a possible descriptor data set for a process p in a general purpose OS. Each entry may be viewed as an element of a structure, where each element type is given inside the box. The access to any element of a given descriptor is accomplished via a conventional selection mechanism. For example, *p->Memory* refers to the *Memory* field of the PCB of process p. We have grouped the items comprising the process descriptor into five categories according to their primary purpose as follows:

Identification

Each process is uniquely identified by its descriptor. Pointers to PCBs are maintained by the system. In addition, a process p often has a system-wide unique identification *p->ID* supplied by the user or the system. Its purpose is to allow convenient and explicit interprocess references. To eliminate conflicts arising from the use of the same ID for more than one process, we assume that the system assigns a new unique ID to every process at the time of its creation. New process IDs are obtained using the function:

```
pid = Get_New_PID()
```

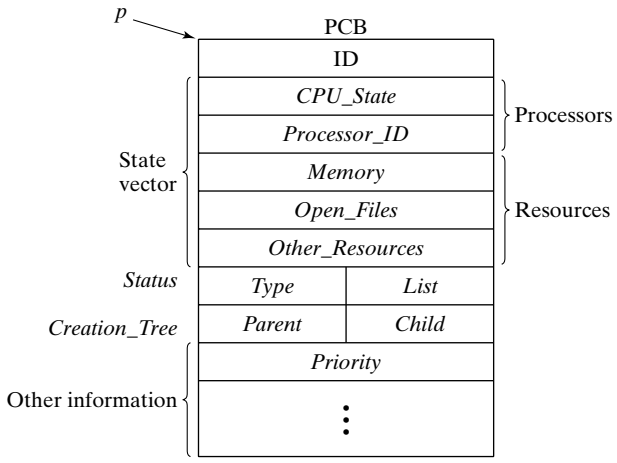

FIGURE 4-5. Structure of process descriptor.

Since all pointers to process descriptors are maintained by the system, we also assume the existence of a function:

```
p = Get_PCB(pid)
```

that takes a unique *pid* as argument and returns the pointer *p* to the process descriptor.

State Vector

The execution of a process or a thread *p* can be described as a sequence of **state vectors** $s_0, s_1, \ldots, s_i, \ldots$, where each state vector s_i contains the address of the current program instruction to be executed and the values of all variables of the program. It also comprises the state of the processor executing *p*, the allocated address space, and any other resources currently associated with *p*. In other words, a state vector of a process *p* is that amount of information required by a processor to run *p* or restart *p* after it has been interrupted. The state vector can be changed either by the execution of *p* or by the execution of other objects sharing state vector components with *p*.

The state vector portion in Figure 4-5 comprises the five components following *ID*, and may be subdivided further into information about the processor(s) and the global resources.

Processors

CPU_State contains copies of all hardware registers and flags of the CPU. This includes the program counter, instruction and data registers, and protection and error flags. In general, *CPU_State* contains the information stored in a typical machine state-word; this data is saved automatically by the hardware when an interrupt occurs. Thus, as long as a process is running, its *CPU_State* field is meaningless—it contains obsolete values saved during the last interrupt. When the process is not running, its *CPU_State* contains the necessary data to restart the process at the point of its last interruption.

The *Processor_ID* field identifies the CPU that is executing *p*. This is meaningful only in a multiprocessor system; in a uniprocessor, it is omitted.

Resources

The *Memory* field gives the main memory map for the process. It describes its real address space and, in case of a virtual memory (see Chapter 8), the mapping between the virtual name space and physical memory locations. Thus, *Memory* could contain upper- and lower-bounds registers for a contiguous partition in main memory, a list of memory blocks allocated to the process, or a pointer to a page or segment table.

The next component *Open_Files* specifies all files that are currently open. Typically, it points to a list of open file descriptors that record such information as file type, disk location, current reading or writing position, and buffer state. The last field, *Other_Resources*, represents jointly all other interesting resources owned by the process. These might be resources such as peripherals, secondary storage space, or files.

Status Information

The status of a process *p* may be described by a structure with two components: *Status.Type* and *Status.List*. *Status.Type* is one of **ready**, **running**, or **blocked**. The meaning of each of these values is given by Table 4-1.

Generally, the status of a *running* process changes to *blocked* if the process issues a resource *Request* that cannot be met. A *blocked* process becomes *ready* as a result of the *Release* of the resource it is waiting for. The changes between *ready* and *running* are the results of scheduling the process for execution.

The field *Status.List* points to one of several possible lists on which the process may reside. When the process is running or ready to run, it has an entry on the *Ready List* of the process scheduler. When the process blocks on a resource, this entry is moved from the *Ready List* to the *Waiting List* associated with that resource. When the process acquires the resource, its entry is moved back to the *Ready List*. The field *Status.List* points to either the *Ready List* or one of the *Waiting Lists*, depending on the process status. This information is essential for efficiency. For example, when the process is to be destroyed, we must be able to find quickly the list on which the process resides.

The three basic status types—*running, ready*, and *blocked*—can handle many situations, but there are some applications for which a finer division is desirable. Consider the following two examples:

- A user is interactively debugging a running program. Often, the user wishes to suspend execution to examine the state of the computation, possibly make some changes, and either continue or terminate the execution.

TABLE 4-1. Status type of process

running	*p* is currently running on a processor (the one designated by *p->Processor_ID*).
ready	*p* is ready to run, waiting for a processor.
blocked	*p* cannot logically proceed until it receives a particular resource for example, a lock, file, table, message, I/O device, or semaphore.

- An internal process might wish to suspend the activity of one or more other processes to examine their state or modify them. The purpose of the suspension may be, for example, to detect or prevent a deadlock, to detect and destroy a "runaway" process, or to temporarily swap the process out of main memory.

In both cases, we could explicitly block the process to achieve the suspension. However, a process could be already blocked when the suspension is desired. Unless we wish to allow a process to be blocked at the same time for more than one reason, a new "suspended" status is required. We define it as follows.

A process is either **active** or **suspended**. If active, it may be running, ready, or blocked, denoted by a *Status.Type* of *running, ready_a*, or *blocked_a*, respectively. When the process is suspended, the *Status.Type* is *ready_s* or *blocked_s*. The possible status changes of a given process *p* are shown in Figure 4-6. Each change is the result of an operation performed by either the process *p* itself (e.g., request a resource) or another process (e.g., *suspend/activate* process *p*). The implementation of these operations will be discussed in detail in Section 4.4.2.

Creation Tree

Earlier in this chapter, the concept of spawning hierarchies of processes was briefly introduced and illustrated in Figure 4-1. Each process *p* has a creator, typically called **parent**, which created the process, and owns and controls any of its offsprings. When the system is first started, one initial process is typically created, which becomes the root of the creation tree. The parent of any process (except the root) is recorded in the field *Creation_Tree.Parent*, usually and conveniently as a pointer to the parent's descriptor. Similarly, every process may create other processes. The *Creation_Tree.Child* field identifies all direct offsprings of a process, say, by pointing to a linked list of their descriptors. This list can be implemented efficiently by distributing the list elements over the PCBs, instead of maintaining it as a separate dynamically allocated list (see the Linux case study below.)

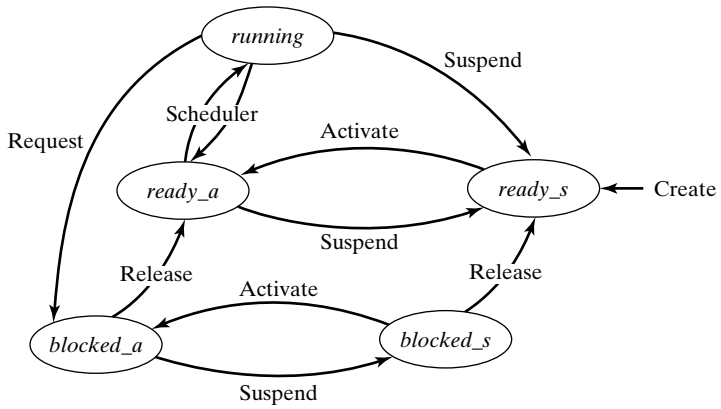

FIGURE 4-6. Process status changes.

Other Information

The *Priority_ID* field indicates the importance of the process relative to other processes. This information is used by the scheduler when deciding which process should be running next. In the simplest case, *Priority* could be a single-integer value, assigned to the process at the time of creation. More typically, the priority of a process consists of two components: a statically assigned **base** priority and a dynamically changing **current** priority. The latter can be derived using a complex function of the base priority, the resource demands of the process, and the current state of the environment, i.e., other processes and resources in the system. (The topic is discussed in detail in Chapter 5.)

The last part of the process descriptor also may contain a variety of other fields that are useful for scheduling, accounting, resource allocation, and performance measurement. Typical elements include CPU time used by the process, time remaining (according to some estimate), resources used, resources claimed, resource quotas, and the number of I/O requests since creation.

CASE STUDY: LINUX PROCESS DESCRIPTOR

The basic data structure for processes in Linux, called the *process descriptor*, contains essentially the information presented above (e.g., [Bovet and Cesati 2001]). We outline some of the differences and more interesting features.

- Linux distinguishes the following main states of a process p:

running	p is either using or waiting for the CPU; Thus this state jointly represents the two states we defined as *running* and *ready_a*; the distinction is implied by the assignment of the CPU
interruptible	p is blocked on a resource; this corresponds to our *blocked_a* state; when the resource becomes available, p is moved to the *running* state
stopped	p has been explicitly suspended; this corresponds jointly to our two states *ready_s* and *blocked_s*
zombie	p has terminated its execution but its PCB is kept active so that its parent can obtain information about p's status

- There are multiple lists linking the PCBs together. A *process lists* links *all* existing PCBs together; a *running list* (corresponding to our *Ready List*) links the PCBs of running processes. These two lists are embedded within the PCBs. That is, each PCB contains two pairs of pointers. Each pair points to the predecessor and the successor PCB on each of the two lists. In addition, there are multiple waiting lists (called wait queues), one for each resource a process may be blocked on. These are implemented as separate linked lists, where each list element points to a PCB.

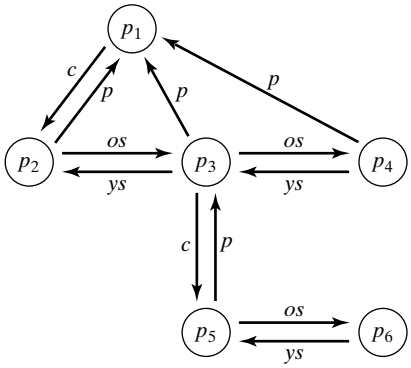

FIGURE 4-7. Representing process creation hierarchy.

- The creation tree of processes also is distributed throughout the PCBs. Each PCB contains four pointers: one for its parent, one for its first child, one for its younger sibling, and one for its older sibling. Figure 4-7 illustrates this scheme for six processes. The pointer labels p, c, ys, and os correspond to the four possible pointer types. The parent, p_1 points to its youngest child p_2 (pointer c). The three processes p_1, p_2, and p_3 are all children of p_1 and point to it using parent pointers. They also point to each other using the sibling pointers. p_3 has two children of its own, p_5 and p_6, linked to it in an analogous manner.

Several other features of the Linux PCBs are worth mentioning:

- In addition to open files, the working directory of the process is also accessible through the descriptor.
- A *signal* component stores the set of signals that have been sent to the process, as well as other related information such as how they should be handled.
- Scheduling data includes the priority and the policy to be executed, such as round-robin or FIFO. Fields are also available for quotas on various resources; examples are file size, stack size, and execution time quotas.

The full descriptor is a complex and lengthy object, leading to a heavyweight process. Linux provides creation of lightweight versions of these with sharing of different components, i.e., the equivalent of threads through the *clone* operation discussed in Chapter 2.

Let us now consider a system that implements multiple threads within each process. Each process and each thread will have its own descriptor. The thread data structure is essentially the *CPU_State*, the *Status*, and the *Priority* fields of the PCB of Figure 4-5. The state changes shown in Figure 4-6 now apply separately to each individual thread. The execution stack of a thread is part of its private, nonshared, space and is pointed to from the descriptor. In addition, there is a pointer to the thread's host process. A process PCB in the threaded environment contains essentially the information about the shared resources, i.e., the memory, files, and other resource fields of Figure 4-5. It also keeps a record of its corresponding threads.

CASE STUDY: THREAD AND PROCESS DATA STRUCTURES IN WINDOWS NT

The record for an NT thread includes its name, priority options, runable processor list, execution times in user and kernel mode, and the basic CPU and stack data. The descriptor for a process object contains the standard information already discussed, as well as a security field with an *access token* that defines the resources accessible by the process.

4.4.2 Implementing Operations on Processes

In Section 2.2, we specified a variety of language mechanisms for creating, activating, and terminating processes and threads. Some of the underlying operations were implicit, and some were explicit. These included *cobegin/coend*, *forall*, *fork*, *join*, *quit*, explicit process declarations and instantiations through a *new* operation, *PAR*, and *thread_name.start*. Here, we show how a representative set of these primitive operations can be implemented.

We view processes and threads as basic OS objects that can be represented by abstract data types. These consist of private data structures—the process descriptors—and the interface operations that manipulate these descriptors. The primitives are considered indivisible CSs and are protected by a common "busy wait" or "spinning" type lock. This kind of lock and its construction are discussed in the next section. For clarity, we omit the locks in the following descriptions; error and protection checking also are omitted. These topics are addressed separately in Chapter 13.

We define four operations on processes:

1. *Create:* Establish a new process.
2. *Destroy:* Remove one or more processes.
3. *Suspend:* Change process status to "suspended."
4. *Activate:* Change process status to "active."

Figure 4-6, introduced in the last section, traces the status changes caused by these operations.

Create

To create a new child process, the parent process calls the *Create* primitive with the input parameters: initial CPU state *s0*, initial main memory *m0*, and priority *pi*. The *m0* field could be a fairly complex structure, for example, encompassing real and virtual space for programs, data, and stacks. The initial status will be *ready_s*—*ready* because the new process should be in position to proceed without waiting for any resources and *suspended* because it may be desirable to create a process well before its activation. The *Create* primitive returns the unique id (*pid*) of the new process as an output parameter.

```
Create(s0, m0, pi, pid) {
   p = Get_New_PCB();
   pid = Get_New_PID();
   p -> ID = pid;
   p -> CPU_State = s0;
```

```
    p -> Memory = m0;
    p -> Priority = pi;
    p -> Status.Type = 'ready_s';
    p -> Status.List = RL;
    p -> Creation_Tree.Parent = self;
    p -> Creation_Tree.Child = NULL;
    insert(self -> Creation_Tree.Child, p);
    insert(RL, p);
    Scheduler(); }
```

The first instruction creates a new instance of the process descriptor (Fig. 4-5) with *p* pointing to that instance. The next two instructions create and assign a unique ID to the new process. Subsequent instructions fill the individual fields of the descriptor using given parameters. We assume that any process is able to obtain the pointer to its *own* descriptor, either through an agreed-upon register or through a kernel call; the name *self* designates this pointer. *Create* adds *p* to *self*'s child list and inserts *p* on the Ready List *RL*. The initial resources—here only main memory—are usually shared resources for the new process and, typically, must be a subset of those belonging to the parent process. The parent may share any of its resources with other child processes in a similar manner. The created process, in turn, can share its resources with any children it creates. At the end, the scheduler is called to select the process to run next.

Suspend

A process is generally permitted to suspend only its descendants. Suspension could be treated in two different ways. A call, *Suspend(pid)*, could suspend only the named process, or it could suspend all its descendants; both these options may be desirable. The latter possibility is somewhat tricky, since a descendent may already have been suspended by their ancestors (which are descendants of *pid*). For example, the RC4000 system (Brinch Hansen, 1970) suspends in this manner but, as a consequence, requires more complex process status types (see Exercise 3). For simplicity, our solution permits suspension of only one process at a time.

```
Suspend(pid) {
   p = Get_PCB(pid);
   s = p -> Status.Type;
   if ((s == 'blocked_a') || (s == 'blocked_s'))
      p -> Status.Type = 'blocked_s';
   else p -> Status.Type = 'ready_s';
   if (s == 'running') {
      cpu = p -> Processor_ID;
      p -> CPU_State = Interrupt(cpu);
      Scheduler();
   }
}
```

This function obtains a pointer *p* to the descriptor of the process with the unique identifier *pid* and sets the *Status.Type* to suspended. If the process was running, the processor executing the process is stored in the variable *cpu* and the machine is interrupted.

We assume that the *Interrupt(cpu)* function returns the current state of the CPU, which is saved in the *CPU_State* field of the PCB. The interrupt also tags the CPU as free. The *Scheduler* is called at the end to allocate the CPU to another *ready_a* process. The suspended process remains linked within the list it occupied prior to its suspension, i.e., the ready list or one of the waiting lists.

Activate

Process activation is straightforward, involving a status change to active and a possible call on the scheduler. The latter permits the option of preemption scheduling if the activated process becomes *ready_a*. A process may activate any of its known descendants, in particular, its child.

```
Activate(pid) {
   p = Get_PCB(pid);
   if (p -> Status.Type == 'ready_s') {
      p -> Status.Type = 'ready_a';
      Scheduler();
   }
   else p -> Status.Type = 'blocked_a';
}
```

Destroy

For destroying a process, the same alternatives are possible as for the *Suspend* operation—we can either remove a single process (a descendant) or remove that process and all of its descendants. If the first policy is selected, the process hierarchy can easily fragment, potentially leaving isolated processes in the system with no control over their behavior. We therefore require that *Destroy* removes the named process and *all* its descendants.

```
Destroy(pid) {
   p = Get_PCB(pid);
   Kill_Tree(p);
   Scheduler();
}

Kill_Tree(p) {
   for (each q in p -> Creation_Tree.Child) Kill_Tree(q);
   if (p -> Status.Type == 'running') {
      cpu = p -> Processor_ID;
      Interrupt(cpu);
   }
   Remove(p -> Status.List, p);
   Release_all(p -> Memory);
   Release_all(p -> Other_Resources);
   Close_all(p -> Open_Files);
   Delete_PCB(p);
}
```

The function is invoked by the call *Destroy(pid)*, where *pid* is the ID of the root process of the subtree to be removed. The procedure obtains the pointer to this process and calls the routine *Kill_Tree(p)*, which recursively eliminates the entire tree. It calls *Kill_Tree(q)* for each process *q* on the current process child list. For each process in the tree, *Kill_Tree* proceeds as follows. If the process is currently running, it is stopped by interrupting its CPU and marking it as free. The procedure then removes the process from the list pointed to by *Status.List*. This is the ready list if the process was running or ready, and otherwise a waiting list associated with the resource upon which the process was blocked. Next, the procedure releases all resources currently allocated to the process and closes all files opened by the process. This potentially unblocks other processes waiting for the released resources. Finally, the process descriptor is deleted and its space returned to the system. When the entire tree is eliminated and *Kill_Tree(p)* returns to *Destroy(pid)*, the process scheduler is called to determine which process(es) should be running next. This may include the current process—the one who issued the *Destroy* command—and any number of processes that may have become unblocked as a result of the *Destroy* operation.

4.4.3 Operations on Threads

Now consider the case of a process/thread framework. A *Create_Process* operation typically establishes the resources for a set of yet-to-be-defined threads, and creates one initial thread. Thus, both a new process descriptor and a new thread descriptor are created. Other resources are shared by future threads within the process. Thread execution may be controlled by four operations analogous to those for processes: *Create_Thread*, *Activate_Thread*, *Suspend_Thread*, and *Destroy_Thread*.

CASE STUDY: POSIX THREADS

The POSIX thread library provides a function *pthread_create()*, which corresponds to the above *Create_Thread* primitive. It starts a new thread that executes a function specified as a parameter. To destroy a thread, the function *pthread_cancel* (and several related ones) are provided. This corresponds to *Destroy_Thread* but cancels only the named thread; its descendants all continue running. Another important distinction is that a *pthread* may control whether and at which points it may be canceled. For example, it may prevent its cancellation while executing in a CS. This capability is similar to the signal-handling options of UNIX processes described in Section 2.5, where a process may ignore or catch a signal, rather than being killed.

4.5 IMPLEMENTING SYNCHRONIZATION AND COMMUNICATION MECHANISMS

Chapter 3 presented a large variety of mechanisms by which processes and threads interact to synchronize, communicate, and share resources. Here, we outline some standard methods for implementing many of these schemes. Most are used primarily at the OS kernel level, but many also can be used at higher OS levels and by user programs. In particular, we introduce basic techniques for building semaphores, locks, monitors, timer services, and message-passing.

Semaphores, locks, monitors, messages, time, and other hardware and software objects can be viewed as resources that are requested and released by processes or threads. These *Request* and *Release* operations may cause a process or thread to change its status between *blocked* and *ready*, as indicated in Figure 4-6. We outline a generic version of the *Request*/*Release* primitives, that is applicable to a single-unit reusable resource, such as a lock, table, I/O device, or memory area. The following pseudocode outlines the necessary actions:

```
Request(res) {
   if (Free(res))
      Allocate(res, self)
   else {
      Block(self, res);
      Scheduler();
   }
}

Release(res) }
   Deallocate(res, self);
   if (Process_Blocked_on(res, pr)) {
      Allocate(res, pr);
      Unblock(pr, res);
      Scheduler();
   }
}
```

The *Request* function first checks whether or not the specified resource *res* is free. If yes, *res* is allocated to the invoking process (*self*); if not, the process is blocked on the resource and the *Scheduler* is called to give the CPU to a *ready* process. When a process no longer needs resource *res*, it does a *Release*. The resource is then deallocated, i.e., removed from the calling process (*self*). If a process, say *pr*, is blocked waiting for *res*, the resource is allocated to that process, and the process is unblocked, which involves changing its status from *blocked* to *ready*, and moving its PCB from the waiting list of *res* to the ready list. Finally, the *Scheduler* is invoked to decide which process should continue.

There are many implementations and forms of the above generic *Request* and *Release* functions, depending on the resource type and the system. They could be calls to the OS kernel for specific resources, for example, *Request_Memory(number_of_blocks)*. Alternately, they could be implemented as library functions to extend the basic functionality of the kernel. In the remainder of this section, we present four specific instantiations of *Request/Release*: 1) as *P* and *V* operations on semaphores (Section 4.5.1); 2) as operations embedded as monitor procedures (Section 4.5.2); 3) as calls to manage clocks, timers, delays, and timeouts (Section 4.5.3); and 4) as *send/receive* operations (Section 4.5.4).

4.5.1 Semaphores and Locks

Few, if any, computers have hardware instructions, such as *P* and *V* operations, which directly address mutual exclusion locks, binary semaphores, or general semaphores.

However, it is not difficult to program the logical equivalents of these objects, provided that our computer has an instruction that can do two things as a global CS. The most common form of such an instruction is to **test and set** a memory location in *one indivisible, atomic* operation.

Let our version of such an instruction be designated *TS(R,X)*. The operand *R* is a CPU register, and the operand *X* is an arbitrary memory location that can contain either a 0 (*false*) or a 1 (*true*). Thus, we treat *X* as a Boolean variable. *TS(R,X)* performs the following two actions indivisibly, i.e., without interruption or interleaving with any other operations:

```
R = X ;
X = 0 ;
```

Thus *TS(R,X)* always sets its operand *X* to 0 and stores the previous value of *X* in the register *R* where it can later be accessed.

In other words, the value of *R* indicates whether a change has been made to the value of *X*.

Spinning Locks on Binary Semaphores

First consider a restricted version of the semaphore operations, one that operates only on **binary** semaphores, i.e., semaphores that may only have the value 0 or 1. We denote such restricted operations as *Pb* and *Vb*. They are defined as follows:

1. *Pb(sb)*: If *sb* is 1 then set it to 0 and proceed; otherwise, i.e., if *sb* is already 0, wait until it becomes 1, then complete the operation.
2. *Vb(sb)*: Set *sb* to 1; note that *Vb* has no effect if the semaphore already is 1.

These operations may be implemented using the *TS* instruction as follows:

```
Pb(sb) is equivalent to:    do TS(R,sb) while (!R):/*wait loop*/
Vb(sb) is equivalent to:    sb = 1;
```

When the semaphore *sb* has the value *false*, a process attempting to execute *Pb* will wait by repeatedly executing the *while*-loop. This form of waiting is referred to as **spinning** or **busy waiting**, since the process consumes memory cycles and processor time while executing the loop. Note that *R* is a register and thus each process has its own private copy of its content. The execution of *TS* is atomic and thus only a single process is able to pass the busy-wait loop.

These spinning operations are used extensively for mutual exclusion locks, where *Pb(sb)* is a lock request, say *mutex_lock(sb)*, and *Vb(sb)* is an unlock, say *mutex_unlock(sb)*. They appear especially in multiprocessor synchronization where several processors may compete for the same lock, and as a means for implementing higher-level methods including the general semaphore algorithms outlined next.

General Semaphores with Spinning Locks

The *P* and *V* operations on a **general** semaphore *s* may be implemented using the above *Pb* and *Vb* operations with two binary semaphores, say *mutex_s* and *delay_s*. The lock

mutex_s assures the logical indivisibility of the code while *delay_s* provides for a busy-wait when necessary on a *P*. Initially, *mutex_s = 1* and *delay_s = 0*.

The *P* operation is defined as:

```
P(s) {
     Inhibit_Interrupts;
     Pb(mutex_s);
     s = s-1;
     if (s < 0) {Vb(mutex_s); Enable_Interrupts; Pb(delay_s);}
     Vb(mutex_s);
     Enable_Interrupts;
}
```

The *V* operation is defined as:

```
V(s) {
     Inhibit_Interrupts;
     Pb(mutex_s);
     s = s+1;
     if (s <= 0) Vb(delay_s); else Vb(mutex_s);
     Enable_Interrupts;
}
```

The locking semaphore *mutex_s* guarantees that only one process at a time will be able to access and manipulate the semaphore variable *s*. Initially, *s* is set to the desired semaphore value and is decremented with each *P* operation and incremented with each *V* operation. Since *s* may become negative, it serves a *dual purpose*. As long as it is greater or equal zero, it corresponds to the actual semaphore value; whenever it becomes negative, it represents the number of processes blocked or waiting on that semaphore.

A process executing a *P* or *V* is protected from processor preemption by inhibiting interrupts for the duration of these operations. This is necessary, even though the semaphore variable *s* is protected by the *mutex_s* semaphore. Without inhibiting the interrupts, the following undesirable situation could occur. Assume that a process executing in the middle of a *P* or *V* operation is preempted and the processor is assigned to another higher-priority process. Suppose that the latter now attempts to execute a *P* or *V* operation on the same semaphore. It would find *mutex_s* equal to 0 and, as a result, remain in the corresponding *while* loop of the *Pb* operation forever. Infinite loops of this type are a form of **deadlock** and must be avoided.

On a uniprocessor system, inhibiting interrupts would normally be sufficient to prevent other processes from simultaneously executing operations on any semaphore. In this case, the *P(mutex_s)* and *V(mutex_s)* operations are not necessary. On a multiprocessor, however, inhibiting interrupts on one processor does not prevent processes running on another processor from accessing the semaphore. This is achieved by the *Pb(mutex_s)* operation, which guarantees that at most one *P* or one *V* is acting upon any given semaphore.

When a process executes a *P* operation and finds the semaphore value less than zero, it blocks itself by executing *P(delay)*. Note that this implements a busy-wait—the

process continues running in a wait loop while another process is in the CS. Busy-waits for CSs are acceptable if the CS is short. Frequently, this is the case. CSs should be designed to have this property if at all possible. But in an environment involving lengthy CSs and synchronizations—for example, when a process is waiting for an I/O operation to be completed, for a hardware resource to become available, or for a message from another process that can arrive at any arbitrary time—busy waits are unsatisfactory. They can degenerate a multiprogrammed system into a uniprogrammed one, as well as increase the response time for real-time events to intolerable levels.

Avoiding the Busy-Wait

The alternative to busy-waiting which avoids the above problems is to *block* processes on unsuccessful *P* operations and provide for the possible activation of processes on *V* operations. Such a blocking policy is employed for most OS applications and for most synchronization and resource allocation mechanisms, including semaphores. We develop one implementation of this general strategy in this section.

The *P* operation is defined as:

```
P(s) {
    Inhibit_Interrupts;
    Pb(mutex_s);
    s = s-1;
    if (s < 0) {     /*Context Switch*/
        Block(self, Ls);
        Vb(mutex_s);
        Enable_Interrupts;
        Scheduler();
    }
    else {
        Vb(mutex_s);
        Enable_Interrupts;
    }
}
```

The operation always decrements *s*. As long as *s* remains nonnegative, the invoking process or thread proceeds as in the case of busy-waits: It releases the *mutex_s* semaphore, enables interrupts, and continues executing the next instruction. When *s* falls below zero, the process cannot proceed. Instead, it blocks itself by invoking the procedure *Block(self, Ls)*. This first saves the current state in the field *CPU_State* of the process descriptor. This information is necessary to restart the process at a later time. The procedure then inserts the process (a pointer to its PCB) on a **blocked list** *Ls* associated with the semaphore *s*. It also updates the *Status.Type* and *Status.List* fields of the descriptor to reflect these changes. Next, the scheduler is invoked, which selects another ready process from a ready list (*RL*) of processes and transfers control to it. Thus, the new process continues executing on the processor instead of the original process. Such a process reassignment is called a **context switch**.

The *V* operation is defined as:

```
V(s) {
    Inhibit_Interrupts;
    Pb(mutex_s);
    s = s+1;
    if (s <= 0) {
        Unblock(q, Ls);
        Vb(mutex_s);
        Enable_Interrupts;
        Scheduler();
    }
    else {
        Vb(mutex_s);
        Enable_Interrupts;
    }
}
```

The operation first increments *s*. If *s>0*, there are no processes blocked on this semaphore, and the operation simply exits by releasing *mutex_s* and enabling interrupts. Otherwise, the operation must unblock the process, *q*, at the head of the list *Ls* associated with *s*. The *Unblock* operation moves the process *q* from the *Ls* list to the *RL* and changes its status accordingly. The scheduler is again called to resume the unblocked process *q*. If there is a free processor, *q* will be started immediately; otherwise, it remains on the RL. Note that, depending on the scheduling policy and the relative priorities of the processes, *q* also could preempt the process that called *V(s)* or another running process.

Since interrupts are inhibited and a busy-wait exists inside the *Pb* operation, it is important that the above code for *P* and *V* operations be short and efficient for the same reasons discussed earlier. The main improvement of this implementation of *P* and *V* over the previous one is that a process busy-waits only when another process is inside a *P* and *V* operation, but not for the entire duration of the CS. This permits CSs to be of arbitrary length, without causing any performance degradation as a result of busy-waiting.

4.5.2 Monitor Primitives

Request/Release operations on resources may be implemented as monitor procedures. There are two main advantages of such an implementation. First, the resource is encapsulated within the monitor as an abstract data type. Second, monitors may be implemented at the user level. The blocking and wake-up of processes or threads is enforced elegantly through the monitor *wait/signal* operations, without modifying the kernel. (Note that monitors are used most naturally in a shared-memory environment, and the active objects are threads in most contemporary systems. We will continue to use the word *process* to mean either process or thread.)

A monitor implementation must enforce mutual exclusion among all monitor procedures. It also must implement the *wait* and *signal* operations. Thus to implement a basic monitor facility, it is necessary to construct code sequences for:

- *Entering* a given monitor: This code is inserted in front of every procedure body within the monitor;

- *Leaving* a monitor: This code is inserted at the end of every monitor procedure;
- *Waiting* on a given condition variable *c*: This code replaces every *c.wait* operation;
- *Signaling* a condition variable *c*: This code replaces every *c.signal* operation.

We assume Hoare monitors and show in detail how they may be implemented using semaphores. Recall the basic semantics of Hoare monitors:

- All procedures or functions must be executed under mutual exclusion.
- Executing a *c.wait* immediately blocks the process on a queue associated with *c*.
- When a *c.signal* is executed, the monitor must determine whether any processes are waiting on the condition *c*. If so, the current process, i.e., the one executing the *signal*, is suspended, and one of the waiting processes is reactivated. Usually, the selection of the process is based on FIFO, but it also could be based on process priorities. If there are no waiting processes, the signaling process continues.
- Whenever a process exits a monitor or issues a *wait*, there may be processes waiting to enter or reenter the monitor. Processes that were suspended earlier as a result of *signal* operations are chosen over those that are queued on initial monitor entry.

To enforce the above semantics, we use three types of semaphores:

- *mutex:* This is used to enforce the mutual exclusion requirement among procedures.
- *condsem_c:* One such semaphore is defined for each condition variable *c* to block processes executing *c.wait*. An associated integer counter *condcnt_c* keeps count of the number of processes currently blocked on *condsem_c*.
- *urgent:* This semaphore is used for blocking processes executing a *c.signal*. An integer counter *urgentcnt* keeps count of the number of processes currently blocked on the semaphore *urgent*.

Initially, *mutex = 1*; *condsem_c = 0* and *condcnt_c = 0* for all *c*; and *urgent = 0* and *urgentcnt = 0*.

The body of each procedure is then surrounded by entry and exit codes to provide mutual exclusion and priority to suspended processes as follows.

```
P(mutex);
procedure_body;
if (urgentcnt) V(urgent); else V(mutex);
```

The *if-statement* checks for processes currently on the *urgent* queue (as a result of an earlier *c.signal* operation). If so, one of them is re-admitted (*V(urgent)*); otherwise, the *V(mutex)* allows a new process to enter one of the monitor procedures by executing the corresponding *P(mutex)*.

Each *c.wait* within a procedure body is coded as:

```
condcnt_c = condcnt_c + 1;
if (urgentcnt) V(urgent); else V(mutex);
P(condsem_c); /* The process waits here. */
condcnt_c = condcnt_c - 1;
```

The process entering this code is about to block itself on *P(condsem_c)*. Prior to doing so, it takes two actions. First, it increments the number of processes blocked on this semaphore. Second, it admits another process into the monitor. The choice is analogous to the statement following each procedure body. If the *urgent* queue is not empty, one of these processes is readmitted; otherwise, a new process is admitted.

After the process wakes up from the *P(condsem_c)* operation, it decrements the count of blocked processes and continues its execution within the monitor procedure.

Every *c.signal* in a monitor is replaced by the code:

```
if (condcnt_c) {
    urgentcnt = urgentcnt + 1;
    V(condsem_c); P(urgent);
    urgentcnt = urgentcnt - 1;
}
```

This code blocks the executing process on *P(urgent)* if the queue associated with condition *c* is not empty. Prior to blocking itself, it increments the *urgentcnt*, which is tested by every exiting procedure and every *c.wait* operation to determine which process to readmit.

The implementation of monitors can be simplified when various restrictions are placed on the use of *wait* and *signal* (see Exercise 15). For example, a *signal*, if it appears at all, is often the last executed statement in a monitor procedure. All the examples in Chapter 3 were of this nature. Given this restricted use of signals, there is no need for *urgent* and *urgentcnt*. It is also the case that a *wait*, if it appears at all, is almost always the first executed statement of a monitor procedure, perhaps guarded by a Boolean expression. In fact, this feature is directly employed in protected types (see Section 2.6.2). Simpler implementations are also possible with other monitor versions, particularly Mesa and Java monitors, which employ a *notify* instead of a *signal*.

Finally, we emphasize the central use of semaphores in implementing monitors, taking advantage of the state-saving, queuing, blocking, and scheduling that are already built into semaphores. A more direct approach may be more efficient, but hardly as clear.

4.5.3 Clock and Time Management

OS and user programs require facilities to access, wait on, and signal **time**—both relative time intervals and absolute "wall-clock" or calendar time. Applications for the OS include performance measurement, processor scheduling, time-stamping events such as I/O and file systems calls, and deadlock and other fault detection.

Current computers provide time support through a variety of mechanisms ranging from a straightforward periodic clock interrupt to a fairly complex clock chip that implements many timer functions and can be controlled by operating software. For our

discussion, we assume the availability of a hardware **ticker** that issues an interrupt at a fixed period. A typical tick interval might be one millisecond. The *Alarm_Clock* monitor presented in Chapter 3 shows how such a tick interrupt is interfaced to a time service.

Another basic timer module that is found in most computers is a hardware **countdown timer**. This device has a program-settable register that is decremented by one at each clock tick; when the register value reaches zero, an interrupt is issued. The tick granularity here can be quite small—on the order of microseconds. In Section 4.6, we discuss in some detail methods and models for connecting interrupts, for example, from clocks, countdown timers, and I/O devices, to OS software.

Given a periodic hardware clock signal and a countdown timer, the goal is to build higher-level clock and timer services for use by the OS and applications. Many of these services appear in the kernel because they are used by other kernel services, must be protected from errors or abuse, and require indivisible execution at a high priority to produce correct results.

Wall Clock Timers

The principal function of a computer's wall clock service is to maintain and return an *accurate* **time of day**. Because computer clocks, like most of those on our wrists, are ultimately controlled by a periodic quartz crystal, they drift over time. For example, a typical crystal might lose or gain up to 10^{-5} μsec per second, which roughly equals one second per day. To maintain accuracy, the clocks are synchronized periodically with accurate standard clocks, such as universal coordinated time (UTC), which can be obtained through GPS receivers or over a network.

The OS and applications also rely on the **monotonicity** of clock values—for two successive clock readings, the value returned by the second reading should be greater or equal to that from the first reading. For example, time-stamps on file updates are commonly employed to determine the most recent update. Monotonicity is violated if a clock is reset back in time during synchronization (or when daylight saving time is reset back to standard time). When a backward change is required, a solution that maintains monotonicity continues running the clock forward but at a *slower* rate until the change catches up.

EXAMPLE: Maintaining Monotonicity

Suppose that a computer clock reads 3:00 P.M., but should be set back one hour to the correct time of 2:00 P.M. If we run the clock at a slower rate, say at one-half the real-time rate, computer time and correct real-time will meet at 4:00 P.M., while the clock is always increasing monotonically. Running the clock at one-half the real-time rate means essentially ignoring every other tick interrupt. At 4:00 P.M., the clock rate of the computer is reset to its normal real-time rate.

An object implementing a wall clock service typically offers three functions:

1. *Update_Clock*: The current time, say *tnow*, is updated. This function is invoked on each clock tick interrupt and must be tightly coupled to it. The update must occur as a CS and, generally, without interruption. The value of *tnow* is typically a single

positive integer indicating the number of clock ticks since some known starting time, making the *Update_Clock* function very fast and deterministic.

2. *Get_Time*: The current clock value is returned. This can be the integer value *tnow*, or it might involve a computation. For example, *Get_Time* could return the current time in the form of a sixtuple [month, year, day, hour, minutes, seconds], where the individual components are derived from the value *tnow*.
3. *Set_Clock(tnew)*: The current time is explicitly set to a new value *tnew*.

Count-Down Timers

Another basic set of timer functions is related to **alarm clocks**. A process or thread requires a *timeout* signal at some specified time in the future. In its purest form, the process wishes to delay, sleep, or block until awakened by the timer signal event. Thus the basic function provided by a countdown timer is:

- *Delay(tdel)*: block the invoking process for the period of time specified by the parameter *tdel*; this is typically a nonnegative time interval relative to the current time *tnow*; i.e., the process remains blocked until the wall clock time reaches *tnow*+*tdel*.

To implement this function, assume first that a dedicated countdown timer is allocated to each process. The following function is provided to operate this counter:

- *Set_Timer(tdel)*: The timer is set to the starting countdown value *tdel*. When the value reaches zero, an interrupt is generated, which invokes a function *Timeout()*

Using the above countdown timer, we can implement the *Delay* function as follows:

```
Delay(tdel) {
  Set_Timer(tdel);
  P(delsem); /* Wait for interrupt. */
}

Timeout() {
  V(delsem);
}
```

The *Delay* function simply loads the hardware timer with the *tdel* value and blocks the calling process on a binary semaphore, say *delsem*, associated with the hardware timer and initialized to zero. When the counter reaches zero, the interrupt routine *Timeout* wakes up the blocked process by performing a *V* operation on the semaphore *delsem*.

In most realistic cases, hardware timers are shared among a group of processes or threads. Each process has its own logical timer, just as each process has its own logical CPU. More generally, a process may desire several logical timers. This leads to the following operations defined for a logical countdown timer object:

1. *tn = Create_LTimer()*: Creates a new timer, returning its identifier in *tn*.
2. *Destroy_LTimer(tn)*: Destroys the logical timer identified by *tn*.

3. *Set_LTimer(tn, tdel)*: This is the logical equivalent to the *set_Timer(tdel)* function defined for a single hardware timer. It loads the timer *tn* with the value *tdel*. When the value reaches zero, *Timeout()* is invoked by the interrupt. Loading the time with a value of zero disables the timer, no interrupt is generated.

The main question is then how to implement *multiple logical* timers using a *single hardware* timer. We present two possible approaches to this problem.

Using a Priority Queue with Absolute Wakeup Times

The first approach uses both a countdown timer and a wall clock timer. The wakeup times of the blocked processes are kept in a priority queue. This could be implemented as a sorted list or a binary heap (see Section 4.2.2). Let the priority queue be designated *TQ* and each element contain a triple *(p, tn, wakeup)*, where *p* is a process identifier, *tn* is a logical countdown timer, and *wakeup* is a future time value until which the process *p* wishes to delay itself. The priority queue orders elements such that lowest *wakeup* values have highest priority.

EXAMPLE: Priority Queue with Absolute Wakeup Times

Figure 4-8a shows an example of such a priority queue, organized as a simple linked list. (We show only the process names and their *wakeup* times; the names of the logical clocks are omitted for clarity.) The figure shows entries for four processes: p_1 wishes to wake up at wall clock time 115, p_2 at time 135, and so on. The figure also shows the current value of the wall clock (103) and the current value of the countdown timer (12). The next interrupt will occur 12 time units later, at time 115.

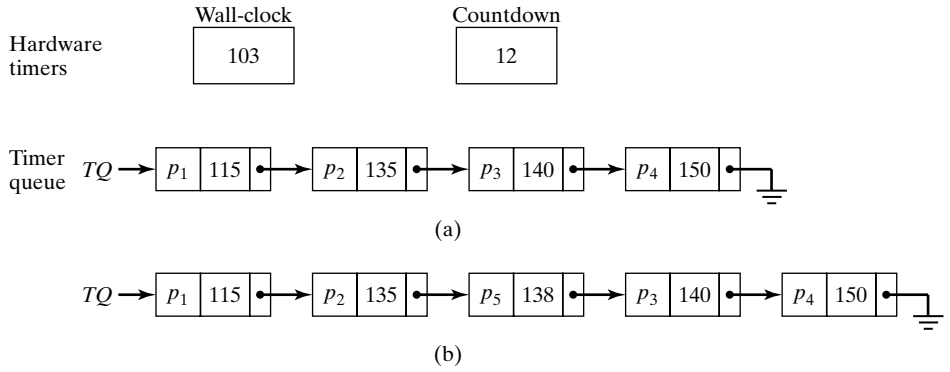

FIGURE 4-8. Timer queue.

The following algorithm then implements the *Set_LTimer* function for logical clocks:

```
Set_LTimer(tn, tdel) {
    wnew = Get_Time() + tdel;
    wold = first(TQ)->wakeup;
```

```
            /* Get wakeup value of first element.*/
    if ( wnew < wold ) Set_Timer(tdel);
    insert(TQ, (*, tn, wnew));
            /* Insert current process (*) into TQ */
            /* in the order determined by the wakeup field. */
}
```

The algorithm uses the computer wall clock to obtain the current time (*Get_Time*). The delay value *tdel* is added to the current time to determine where to place the new entry within the list. If *wnew* is less than the wakeup value of the first element in the queue (*wold*), the new element becomes the head of the queue. In this case, the countdown timer must be set to the new delay interval *tdel* using the *Set_Timer(tdel)*. The new element is then inserted into *TQ* according to the *wakeup* value.

Due to the time-sensitive nature of the operations, the function is executed as a CS. In particular, interrupts from the timer are inhibited during *Set_LTimer*.

EXAMPLE: Effect of `Set_LTimer`

To illustrate the above code, consider the state of the system in Figure 4-8a and assume that a new process p_5 issues the call *Set_LTimer(tn, 35)*. The value of *wold* is 115; the value of *wnew* is $103 + 35 = 138$. The new element is inserted as shown in Figure 4-8b, and the value of the countdown timer remains unchanged. In contrast, a new element created with the call *Set_LTimer(tn, 5)* would become the current head of the queue and the timer would be set to 5.

The *Delay* function can now be implemented using logical timers in the same way as hardware timers. It loads the logical timer with *tdel* value using the above function *Set_LTimer(tn, tdel)*. It then blocks the invoking process on a *P* operation.

When the value in the physical countdown timer reaches zero, the interrupt routine services the first element in the queue *TQ*. It removes this element and wakes up the corresponding process using a *V* operation. It also sets the timer to a new value, computed as the difference between the *wakeup* value of the next element in the queue and the current time, *Get_Time*, i.e.:

```
Set_LTimer(tn, first(TQ)->wakeup - Get_Time())
```

EXAMPLE: Effect of Timer Expiration

In Figure 4-8, when the countdown timer reaches zero, the wall clock time reads 115. The first element of TQ is removed, process p_1 is woken up, and the time is set to $135 - 115 = 20$, to wake up process p_2 20 time units later.

Using a Priority Queue with Time Differences

The previous solution made no assumptions about the implementation of TQ. Figure 4-8 shows an implementation as a sorted linked list, but TQ also could be implemented as a binary heap to speed up insertion or another form of priority queue. A linked-list implementation is suitable when the number of logical timers is relatively small, say on the order of 20 or less. In that case, a more efficient variant is possible which uses only the countdown

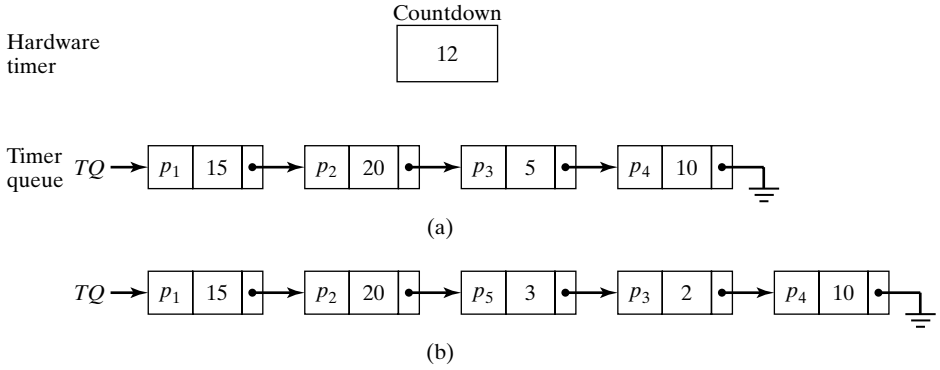

FIGURE 4-9. Timer queue with differences.

timer; no wall clock is required. The idea is to record only the **differences** between wakeup times in successive elements, rather than the absolute wakeup time values.

Figure 4-9a illustrates this idea for the same four processes of Figure 4-8a. Process p_1 is at the head of the queue and so it will wake up when the countdown timer reaches 0, i.e., after 12 time units. p_2 is to wake up 20 time units after p_1, p_3 is to wake up 5 time units after p_2, and so on. Thus, the interrupt routine is quite simple: Whenever the time expires and the process at the head of the queue is removed, the countdown timer is set to the value stored in the next element. In Figure 4-9a, when p_1 is removed, the counter is loaded with the value 20.

The *Set_LTimer(tn, wakeup)* is more complicated. Its main task is to find the two elements, say *left* and *right*, between which the new request is to be inserted. It does so by adding the differences in the entries, starting with the current value of the countdown timer, until the *wakeup* value is reached. The difference currently stored in the *right* element is then divided into two portions. One is stored in the new element and represents the difference between the left and the new element. The other is stored in the right element and represents the difference between the new and the right element.

EXAMPLE: Priority Queue with Time Differences

At the time represented by Figure 4-9a, process p_4 is scheduled to wake up in $12+20+5+10=47$ time units. Suppose that a *Set_LTimer(tn, 35)* is executed at this time by a process p_5. The effect is shown in Figure 4-9b. The entry for p_5 is inserted between p_2 and p_3 since p_5's wakeup time lies between two processes: $(12+20)=32<35<(12+20+5)=37$. The original difference in p_3, (5), is divided into $3+2$, where 3 is stored in the new element p_5, and 2 is stored in its right neighbor, p_3.

The main purpose of countdown timers is to generate **timeout** signals. Synchronization primitives that involve possible blocking or waiting, such as those defined in semaphores, message-passing, or rendezvous, often have built-in timeouts. For example, instead of a regular *P* operation on a semaphore, it is also common to support a *P* with timeout. Such an operation might have the form:

```
int P(sem, dt)
```

which returns 1 for a normal *P* operation on the semaphore *sem*, and 0 if the calling process is still blocked after *dt* time units have expired.

4.5.4 Communication Primitives

We introduce the implementation principles of building centralized and distributed communications. The most basic form involves **message-passing** and is often directly supported at the kernel level. More abstract mechanisms, such as rendezvous and RPC, can be realized with message-passing operations, as was illustrated in Section 2.2. Message-passing is most natural with processes or threads executing on separate machines or with separate address spaces that wish to communicate or synchronize.

A generic form of message-passing primitives includes the following two basic operations:

- *send(p, m)*: Send message *m* to process *p*.
- *receive(q, m)*: Receive a message *m* from process *q*.

Either operation could be blocking or nonblocking, and it could be selective or nonselective (i.e., name a specific process *p* or send to/receive from any process).

Consider first a shared-memory system with one or more processors. To implement any form of the send/receive operations, we need two **buffers**: one in the user space of the sender to hold the message while it is being sent, and the other in the user space of the receiver to hold the received copy of the message. Figure 4-10a illustrates the situation. The parameter *m* of the *send* operations points to the sender's buffer, *sbuf*, which holds the message to be sent. Similarly, the *receive* operation identifies a receiving buffer, *rbuf*. When *send* is invoked, the system copies the message from *sbuf* to *rbuf*, where it may be retrieved by the *receive* operation.

There are two important issues that must be resolved with the above simple scenario:

1. How does the sender process know that *sbuf* has been copied and may be reused?
2. How does the system know that the contents of *rbuf* are no longer needed by the receiver process and may be overwritten?

There are two possible solutions to the first problem. One is to block the process inside the *send* operation until *sbuf* has been copied. This results in a *blocking send*. If the *receive* also is blocking, the communication is completely synchronous; both processes must reach their respective send/receive points before either one may continue.

Such tight synchronization is not always necessary or even desirable. For example, the sender may not wish to reuse *sbuf* and need not be blocked. To permit the sender to continue, the system could provide a flag associated with *sbuf* to indicate whether *sbuf* has already been copied. This results in a *nonblocking send*; however, the need to explicitly poll the flag is awkward. Alternately, an interrupt could be generated when the buffer has been copied to inform the sender, but the resulting nondeterminism makes programming very difficult.

A similar difficulty arises at the receiver's site (Question 2 above). A possible solution is to associate a flag with *rbuf* to inform the system that *rbuf* is no longer needed. But this flag must be explicitly set by the receiving process and repeatedly tested by the system. Consequently, such mechanisms are not very convenient.

A more elegant solution to the synchronization problems at both ends is to use additional intermediate buffers, rather than attempting to copy *sbuf* directly to *rbuf*. Figure 4-10b shows the general organization. A pool of **system buffers** is used at each end. The *send* operation simply copies the message from the user buffer *rbuf* to one of the system buffers. For selective sends, it also includes the name of the receiving process. Once the copy is made, the sender is free to proceed; thus, the send becomes *nonblocking*, and with a sufficiently large pool of system buffers, allows the sender to continue generating and sending messages. The system copies each full buffer, along

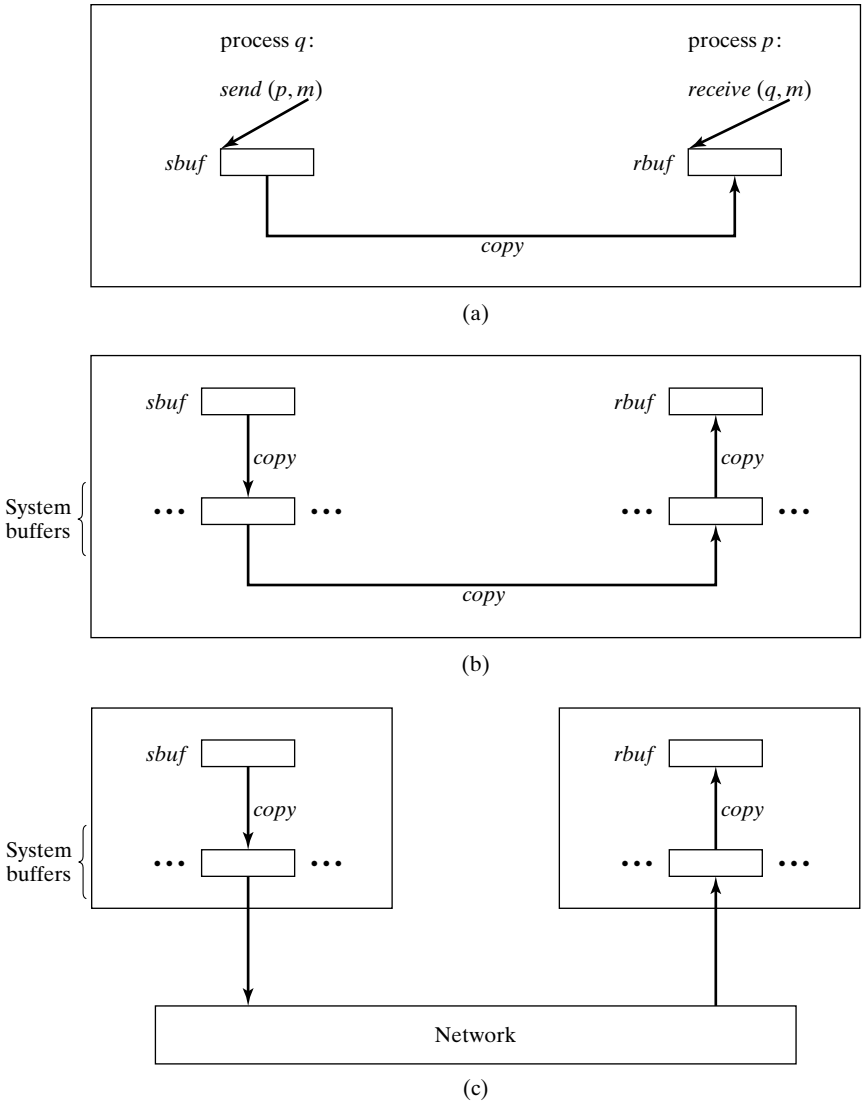

FIGURE 4-10. Send/receive buffers; (a) copying user buffers; (b) copying through system buffers; and (c) copying across network.

with the sender's name, into a corresponding empty system buffer at the receiver's end. (Or it simply reallocates the full buffer to the receiver, avoiding one copy operation.) Thus, the set of system buffers acts as a message port for receiving messages.

Each *receive* operation then simply copies the next system buffer into the *rbuf* of the receiver, and the system buffer is freed. It is up to the receiver to determine when *rbuf* is no longer needed; it will be overwritten whenever the process issues the next *receive* operation.

Using this scheme, *receive* also may be implemented as *blocking* or *nonblocking*. In the first case, the operation blocks if no system buffer contains any data for the process. It is awakened when a message arrives and is copied into *rbuf*. A nonblocking version only checks the availability of full system buffers. If none are available, a corresponding error condition is returned, instead of blocking the process.

CASE STUDY: RC4000 PRIMITIVES

The multiprogramming system for the RC4000 computer (Brinch Hansen 1970) was an early influential OS that pioneered the use of processes. An interesting and elegant set of message transmission operations was defined, consisting of an asynchronous *sendmessage*, a blocking *receivemessage*, an asynchronous *sendanswer*, and a blocking *waitanswer*. Under normal conditions, a sender issues a *sendmessage*, followed later by a *waitanswer*; a receiver's sequence is *waitmessage*, followed by *sendanswer*. Each process has a message queue, essentially a set of buffers linked together in the order of message sends, to store received messages and answers.

The basic *send/receive* scheme described above works provided that the processes and communication channels are error-free. In reality, common errors include issuing a *send* to a nonexistent process, transmitting bursts of many *send*s to the same process, resulting in buffer overflows and deadlocks where a *receive* might be blocked forever. The system will attempt to hide errors from the application, for example, by automatic retransmissions and the use of error-correcting codes. However, some errors cannot be handled fully transparently. A common approach to error detection is to return a *success/fail* flag (or an error code) on each function call, and let the process take appropriate actions. Another mechanism is to add a timeout parameter to the blocking primitives (here, the blocking *receive*) and indicate to the process (by an error code) whether the operation succeeded or timed out.

The use of system buffers is particularly important in multiprocessor or multicomputer systems, where no shared memory is available between different machines and the delays in message transmissions are large. Figure 4-10c shows such a system with two machines communicating with each other through a network—LAN or WAN.

The copying of the system buffers involves much additional processing, generally performed by a specialized communications (sub)kernel to handle the I/O across the network. The kernel is typically responsible for breaking messages into fixed-size transmission packets, routing the packets through the network, reassembling them into messages at the destination, and handling a variety of transmission errors. Thus, the copying of each message buffer between different machines passes through multiple

layers of communication protocols. For example, the Open Systems Interconnection Reference Model (OSI), developed by the International Standards Organization, specifies seven different layers, interfaces, and protocols for communications: physical, data link, network, transport, session, presentation, and application. Another international standard, promoted by telephone companies as a means to handle both voice and data messages, is Asynchronous Transfer Mode (ATM). This layered protocol is a suitable choice for multimedia applications.

4.6 INTERRUPT HANDLING

The term *interrupt* generally refers to an event occurring at an unpredictable time that forces a transfer of control out of the normal processing sequence of a computer. One insightful view is that the purpose of interrupt handling is to remove the notion of asynchronous events from higher levels of the kernel, the OS, and applications. Ultimately, it is through interrupts that the abstraction of processes—almost independent activities operating logically in parallel—is implemented. The asynchrony of events, such as I/O completion, also is hidden by the interrupt handling mechanisms.

As described briefly in Section 1.2.2, interrupts are often classified as **internal** or **external**, depending on their source. External interrupts are generated by hardware that is extraneous to a computer and essentially runs in parallel with it. The standard examples of this extraneous hardware are, of course, I/O devices and controllers (see Chapter 11). Through interrupts, they signal events such as the end of an I/O read or write, the arrival of a message, or an error during I/O. Other external interrupt sources are timers, other processors in a multiprocessor system, and machine-monitoring systems that check for hardware errors.

Internal interrupts or traps are those that occur directly as a consequence of the current computation. These also are called **exceptions** because they identify some exceptional, usually error, condition associated with the current instruction. By nature, exceptions occur synchronously with the current computation. Similarly, traps to the supervisor, such as SVC instructions, are also synchronous. Our discussion in this section pertains primarily to external interrupts, which represent asynchronous events produced by physically concurrent hardware.

Interrupts from the same source are generally treated as a class in terms of their hardware access, shared data, and operations. The different classes are sorted by their importance. Common operations on classes of interrupts include the following:

- *Enable*: Activate the interrupts. Any pending (previously inhibited) interrupts are now processed and future interrupts are processed immediately.
- *Disable*: Deactivate the interrupts. All future interrupts are ignored. Note that some critical classes of interrupts, such as a power failure, cannot be disabled.
- *Inhibit*: Delay the processing of interrupts. The interrupts are kept pending until the class is again enabled using the above operation.

When an interrupt occurs, further interrupts from classes at the same or lower priority level are normally automatically inhibited until a special instruction, often called *return_from_interrupt*, is executed. Built-in hardware priorities permit handlers of high-priority events to preempt those of lower priority, much in the same way as a CPU scheduler ensures that high-priority processes preempt lower-priority ones.

Regardless of the source and type of interrupt, the OS goes through a standard interrupt-handling sequence whenever the CPU is interrupted:

1. Save the state of the running process or thread so that it can be restarted later. Most of this is done automatically by the CPU interrupt hardware.
2. Identify the type of interrupt and invoke an *interrupt service routine*, generally called the **interrupt handler** (IH), associated with this type of interrupt.
3. Execute the IH, which services the interrupt. Because the IH is considered to be a CS, every effort is made to keep it short.
4. Restore the state of the interrupted process on the CPU, which resumes its execution at the point of the interruption. Alternatively, another process could be selected for execution on the interrupted CPU. This is frequently the case, because the IH may have awakened processes that were waiting for the interrupt-causing event to occur.

Figure 4-11a illustrates the use of this interrupt-handling sequence in the design of a hardware/software interface. We assume the existence of an external hardware device. To use this device, a process p invokes a procedure, Fn, which initiates the device, passes parameters to it from the calling process, and, when the device terminates, returns results to the process. The procedure Fn, after initiating the device, blocks itself to await the completion of the device operation. The OS takes over and selects another process to run in the meantime. When the device terminates, it generates an interrupt, which saves the state of the currently running process and invokes the IH. The IH services the interrupt, unblocks process p, and issues the *return_from_interrupt* instruction, which transfers control back to the OS. The scheduler now selects the next process to run, and restores its state. We assume that the original process p is restarted. This allows the procedure Fn to complete and return to its caller.

This example illustrates some of the main difficulties in developing interrupt-handling mechanisms to address asynchronous events. First, the procedure Fn must be able to block itself on a given event. If this procedure is to be written by the user, this requires knowledge (or possibly even a modification) of the OS kernel. Second, the IH must be able to unblock a process associated with the event. Third, the IH must be able to "return" from the interrupt, i.e., pass control to the OS. These issues must be addressed by specially designed kernel mechanisms, even if application programmers are allowed to develop their own interrupt-handling facilities. Designing the interfaces among the hardware, the kernel routines, and the rest of the system coherently and without error has always been a major challenge. We present one model that provides a framework for building these interfaces in a more uniform abstract manner.

The solution is to extend the process model down into the hardware itself, so that interrupts and their handlers are somehow included. At the same time, we replace the blocking and wakeup facilities by standard interprocess synchronization constructs, such as P/V, *send/receive*, or monitor operations, which do not require any knowledge of or extension to the kernel.

Figure 4-11b shows the modified organization corresponding to the situation of Figure 4-11a. We view the hardware device as a separate process, which is started (at least conceptually) by the *Init* operation issued by the procedure Fn. The procedures Fn and IH, which are designed to operate the hardware, are implemented in the form

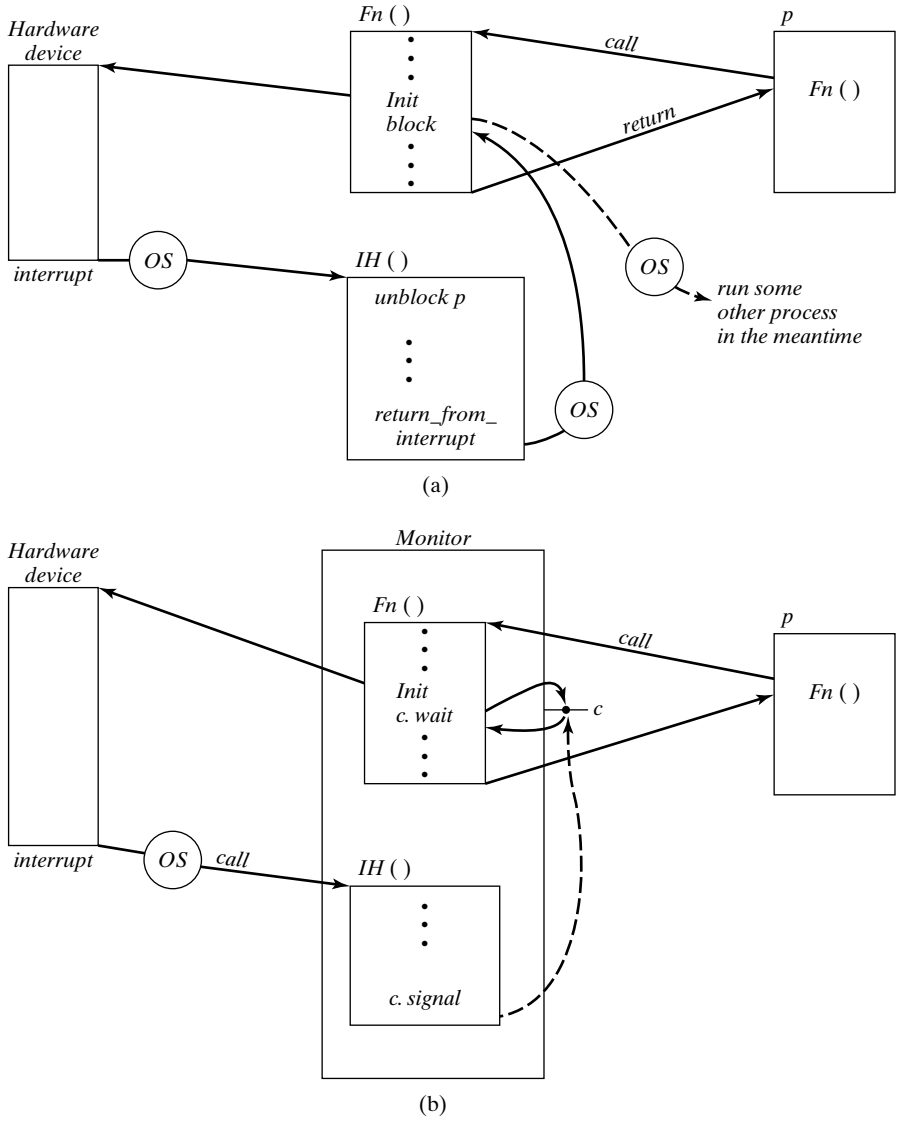

FIGURE 4-11. Using a hardware device; (a) basic interrupt sequence; and (b) using a monitor.

of a monitor placed between the hardware and the process *p*. The process uses the hardware by invoking *Fn* as before. *Fn* initiates the hardware process and blocks itself using *c.wait*, where the condition *c* is associated with the device. When the hardware terminates, the interrupt is implemented as a call to the *IH* function of the monitor. This services the interrupt and issues a *c.signal* to wake up the function *Fn*.

The main advantages of this solution over that in Figure 4-11a are the following. The procedures *Fn* and *IH* require no knowledge of the OS kernel; they interact purely through the monitor primitives and may be written at the application level. The blocking,

which also involves the OS and causes another process to execute in the meantime, is completely transparent to Fn. The *Init* procedure is a process-creation call to the hardware process, similar to a *fork*, or a wakeup message to the hardware process. The interrupt is a procedure call to a monitor procedure, rather than special signal. The hardware must still save the state of the interrupted process, but from p's perspective, IH is just like any other procedure of the monitor.

Note that some of the procedures (e.g., Fn) are called by the software process, whereas others (e.g., IH) are called by the hardware process. Thus, the monitor serves as a uniform hardware/software interface. It provides CS protection, it allows the hardware to signal the arrival of data when appropriate (by calling IH), and allows the processes to wait for data or events when logically necessary.

An interesting question is then: To which process does the IH procedure belong? One answer that fits in well with our abstractions and leads to straightforward and efficient code is: Let IH belong to the hardware process. Thus, the monitor is shared by the process p and by the hardware process.

EXAMPLE: Simple Clock Server

Consider a simple clock server with three procedures, *Update_Clock*, *Get_Time*, and *Set_Clock*, as described in Section 4.5.3. *Update_Clock* is directly connected to a periodic hardware ticker, whereas the other two procedures are used by processes to access and set the current time (Fig. 4-12). The code for the monitor is outlined below:

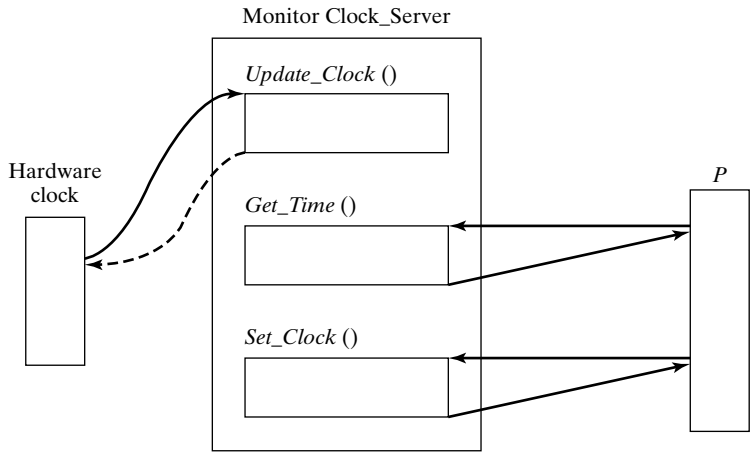

FIGURE 4-12. Using a timer through a monitor.

```
monitor Clock_Server {
. . .
Update_Clock() {
. . .
    tnow = tnow + 1;
    /* Perhaps update time structure also.*/
}
```

```
int Get_Time() {
. . .
    return(tnow);
    /* Perhaps return some more complex time structure
        instead. */
}

Set_Clock(int tnew) {
. . .
    tnow = tnew;
}
}
```

Each clock tick is generated by a hardware process that calls *Update_Clock*. In the underlying implementation, the clock interrupt mechanism automatically saves the currently running process state and calls the *Update_Clock* procedure. When it exists, the interrupt handler restores the state of the original process.

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

Activate a process	Process control block (PCB)
Asynchronous Transfer Mode (ATM)	Priority queue
Binary heap	Process descriptor
Blocked list	Process state vector
Busy wait	Process status
Clock server	Queue
Context switch	Ready list (RL)
Control block	Release operation
Countdown timer	Request operation
Descriptor	Spinning lock
Exception	Suspend a process
Interrupt handler (IH)	Test and set instruction (TS)
Kernel	Thread
Nucleus	Thread descriptor
Open Systems Interconnection Reference Model (OSI)	

EXERCISES

1. Consider a producer/consumer situation where the producer must never be blocked. To avoid the overhead of implementing the buffer as a linked list, the following scheme is used. The buffer is implemented as fixed size array, $A[n]$, where n is large enough most of the time. In the case where the producer finds the buffer full, the array size n is temporarily extended to accommodate the spike in the production. The

extension is removed when the number of elements falls again below n. Compare the effectiveness of this scheme with a linked list implementation, given the following values:

- tl is the time to perform one insert or remove operation in a linked list implementation;
- ta is the time to perform one insert or remove operation in the proposed array implementation;
- oh is the overhead time to temporarily extend the array;
- p is the probability that any given insert operation will overrun the normal array size n.

(a) Derive a formula for computing the value of p below which the proposed scheme will outperform the linked list implementation.

(b) What is the value of p when $tl = 10 \times ta$ and $oh = 100 \times ta$?

2. Assume that each resource is represented by a data structure called RCB (resource control block). This contains information about the resource and a waiting queue for processes blocked on this resource. Consider a single-processor system with currently four processes p_1 through p_4, and two resources, r_1 and r_2. Process p_1 has created p_2 and p_3; process p_3 has created p_4. Presently, p_1 is *running*, p_2 is *ready*, and p_3 and p_4 are both *blocked* on the resource r_2. The ready list is a simple linked list of pointers to processes. Show the details of all process descriptors and their interconnections (pointers) reflecting the above system state.

3. Suppose that the *Suspend* and *Activate* operations are similar to *Destroy* in that they suspend not only the named process but also its descendants. Modify *Suspend* and *Activate* accordingly. To simplify the solution, assume that a process may only suspend and activate a process it has created as its direct child process (see Exercise 4).
Note that processes lower in the hierarchy may already be suspended. Make sure that any suspended descendant process is activated only by the process that originally suspended that process. (*Hint:* Extend the process control block to keep track of who suspended the process.)
When devising your algorithms, consider what should happen when a process attempts to suspend an already suspended child processes, or activate an already active child process.

4. In the previous exercise, it was assumed that a process may suspend and activate only its immediate child processes.

(a) What difficulties would arise if a process was allowed to suspend and activate any of its descendants (i.e., child, grandchild, grand-grandchild, etc.)? (Hint: Consider what should happen if two processes attempt to suspend and later activate the same process.)

(b) What difficulties would arise if a process was allowed to suspend itself (or possibly one of its ancestors)?

5. Can a process apply the function *Destroy()* to itself to successfully terminate its own execution? If not, what changes must be made?

6. Simplify the *Create, Destroy, Suspend*, and *Activate* operations for a system with only a single CPU.

7. Consider a system with only three possible process states: running, ready, and blocked, i.e., there is no distinction between *active* and *suspended*. Modify the process state diagram (Fig. 4-5) to reflect the simplification.

8. Consider a machine instruction, *SWAP*, which swaps the contents of a memory location, M, with the contents of a register, R, in a single indivisible operation. That is,

SWAP is defined as follows:

```
SWAP(R, M){temp = R; R = M; M = temp; ; }
```

Implement the Pb and Vb operations on binary semaphores using *SWAP*. (Hint: Use the fact that the memory location M is shared, and each process has a private copy of the register R.)

9. Consider a machine instruction, *TSB*, which performs the following function as a single indivisible operation:

```
TSB(X, L) {if(X == 0) goto L else X = 0; }
```

That is, *TSB* tests the variable X. Depending on the outcome of the test, the instruction either branches to an address L, or it sets X to zero, and execution continues with the next instruction following *TSB*.
Implement the Pb and Vb operations on binary semaphores using *TSB*.

10. Is the implementation of the general semaphore operations using spinning locks (Section 4.5.1) still correct if one of the following changes are made:
- **(a)** The *Pb(mutex_s)* statement is moved in front of the *Inhibit_Interrupts* statement in the P operation.
- **(b)** The *Vb(mutex_s)* statement is moved in front of the *if* $(s < 0)$ $\{\ldots\}$ statement in the P operation.

11. The UNIX OS has no general interprocess communication or synchronization scheme. Instead, binary semaphore operations can be simulated by creating and deleting a known file. Find out how files are created and what happens when an already existing file is "created." Show how this idea can be implemented.

12. Consider the following synchronization primitives:
- *ADVANCE(X)*: increments the variable X by 1;
- *AWAIT(X,C)*: blocks the process executing this instruction until $X \geq C$.

Using these primitives, develop a solution to the bounded-buffer problem.

13. Assume that the following function is part of a monitor body:

```
f(x) {
   if (x) c1.wait;
   x++;
   c2.signal;
   x = 0;
}
```

Translate this code using P and V operations to implement the *wait* and *signal* operations and the mutual exclusion as required by the definition of Hoare monitors.

14. Demonstrate that monitors and semaphores have equal expressive power. (Hint: Consider the implementation of monitors given in the text. Then write a monitor that emulates P/V operations on general semaphores.)

15. Under each of the following assumptions, the code implementing a monitor (see Section 4.5.2) may be simplified. Show the new code for each case:
- **(a)** The monitor contains no *wait* or *signal* operations.
- **(b)** The last instruction of the monitor body is *signal*. (Other *signal* operations may occur earlier in the body.)
- **(c)** The use of *signal* is restricted such that it may occur *only* as the last instruction of any monitor.

16. Show how Mesa monitors (see Section 4.5.2) can be implemented using *P* and *V* operations on general semaphores.

17. Modify the *Set_LTimer()* function of Section 4.5.3 such that it blocks the invoking process until an absolute wall clock time *tabs*, i.e., the function has the format *Set_LTimer(tn, tabs)*. What should happen when the specified wakeup time *tabs* is already in the past?

18. Consider the *Delay(tdel)* function in Section 4.5.3. Implement an analogous function, *Delay(tn,tdel)*, which uses a logical countdown timer *tn* instead of a physical one. Show the pseudocode for the function and the associated *Timeout()* function.

19. Assume at time 500 there are four processes, p_1 through p_4, waiting for a timeout signal. The four processes are scheduled to wake up at time 520, 645, 695, 710.

(a) Assuming the implementation using a priority queue with time differences (see Section 4.5.3), show the queue and the content of the countdown timer at time 500.

(b) After p_1 wakes up, it immediately issues an another call *Set_LTimer(tn,70)*. Assuming that processing the call takes no time, show the priority queue after the call.

(c) Assume p_1 issues the same call again (*Set_LTimer(tn,70)*) immediately after it wakes up for the second time. Show the new priority queue.