2.6.2 Serializers

Hewitt and Atkinson [8] proposed *serializers* as a synchronization mechanism to overcome some of the deficiencies of monitors. Serializers allow concurrency inside and thus the shared resource can be encapsulated in a serializer. Serializers replace explicit signaling required by monitors with automatic signaling. This is achieved by requiring the condition for resuming the execution of a waiting process to be explicitly stated when a process waits.

The basic structure of a serializer is similar to a monitor. Like monitors, serializers are abstract data types defined by a set of procedures (or operations) and can encapsulate the shared resource to form a protected resource object. The operations users invoke to access the resource are actually the operations of the serializers. Only one process has access to the serializer at a time. However, procedures of a serializer may have *hollow* regions wherein multiple processes can be concurrently active (see Fig. 2.7). When a process enters a hollow region, it releases the possession of the serializer and consequently, some other process can gain possession of the serializer. Any number of processes can be active in a hollow region. A hollow region in a procedure is specified by a *join-crowd* operation that allows processes to access the resource while releasing (but not exiting) the serializer, thereby allowing concurrency. The syntax of the join-crowd command is

join-crowd (<crowd>) **then** <body> **end**

On invocation of a join-crowd operation, possession of the serializer is released, the identity of the process invoking the join-crowd is recorded in the *crowd*, and the list of statements in the *body* is executed. At the end of the execution of the body, a *leave-crowd* operation is executed which results in the process regaining the possession of the serializer and the removal of the identity of the process from the crowd.

The queue of a serializer is somewhat different than a monitor queue. Instead of condition variables, a serializer has *queue* variables. An *enqueue* operation, along with

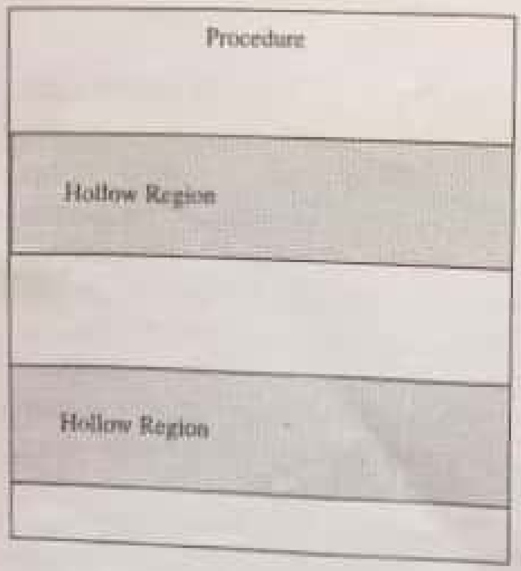

FIGURE 2.7
The structure of a procedure of a serializer.

the condition the process is waiting for, provides a delaying or blocking facility. The syntax of the enqueue command is

enqueue (<priority>, <queue-name>) **until** (<condition>)

where *priority* specifies the priority of the process to be delayed. A process invoking the enqueue is placed at an appropriate position (based on the priority specified) of the specified queue and the condition is not checked until the process reaches the head of the queue. The serializer mechanism automatically restarts the process at the head of the queue when condition for which it is waiting is satisfied and no other process has control of the serializer. No explicit signaling is required, in contrast to monitors.

Serializers derive their name from the fact that all of the events that gain and release possession of the serializer are totally ordered (serial) in time. A typical sequence of events occurring in the use of a protected resource follows. A process gains possession of the serializer as a result of an *entry* event. The process waits (possession of the serializer is released) until a proper condition is established before accessing the resource. An *establish* event regains possession of the serializer as a result of a *guarantee* event, with the proper condition for accessing the resource established to be true. Then, a *join-crowd* event releases the possession of the serializer, records that there is another process in the crowd—an internal data structure of the serializer—that keeps track of which processes are using the resource. The *leave-crowd* event releases the resource, regaining possession of the serializer. An *exit* event releases the serializer. There is also a *timeout* event which regains possession of the serializer as a result of waiting for a condition longer than the specified period.

Example 2.5. Figure 2.8 gives a solution to the readers-priority problem using serializers. Note that *empty(crowd)* and *empty(queue)* operations permit us to check if a crowd or queue is empty. On invoking the procedure *read*, a reader process is blocked if there is an active writer (i.e., wcrowd is not empty); otherwise, reader proceeds and executes join-crowd operation thereby joining the crowd *rcrowd* (i.e., the presence of a reader in the resource (db) is recorded) and releasing the possession of the serializer. Releasing the serializer facilitates concurrent access to the resource by allowing another reader to gain the control of the serializer and execute a join-crowd operation. On completing the read operation, a reader leaves the body of the join-crowd which causes the automatic execution of a leave-crowd operation. The leave-crowd operation results in a reader regaining control of the serializer and its removal from the rcrowd (i.e., the reader is no longer accessing the resource).

A writer, on the other hand, invokes the *write* procedure and proceeds to execute a join-crowd operation, only if there are no active writers (wcrowd is empty), no active readers (rcrowd is empty), and no readers are waiting (readq is empty); otherwise, a writer process is blocked and is queued in the *writeq*. On executing a join-crowd operation, a writer joins the crowd *wcrowd* and releases the possession of the serializer. Reader and writer processes trying to gain access to the resource when a writer is active are blocked and queued in readq and writeq, respectively. On completion of the access, a writer leaves the body of the join-crowd, causing the [illegible] The leave-crowd operation results

That is, a writer does not have to wait for readq to become empty. Consequently, when a writer departs and both a reader and a writer are waiting, dequeue conditions for both are satisfied and one of them is dequeued randomly.

Writer's priority solution. To obtain a writer's priority solution, we need to replace the enqueue command in writer procedure by the following command:

enqueue (writeq) **until** (empty(wcrowd) **AND** empty(rcrowd));

and also replace the enqueue command in the reader procedure by the following command:

enqueue (readq) **until** (empty(wcrowd) **AND** empty(writeq));

A major drawback of serializers is that they are more complex than monitors and therefore less efficient. For example, crowd is not a simple counter, but a complex data structure that stores the identity of processes. Also, the automatic signaling feature, while simplifying the task of a programmer, comes at a cost of higher overhead. Automatic signaling requires testing conditions waited upon by processes at the head of every queue every time possession of the serializer is relinquished.

2.6.3 Path Expressions

The concept of *path expression* was proposed by Campbell and Habermann [2]. Conceptually, a "path expression" is a quite different approach to process synchronization. A path expression restricts the set of admissible execution histories of the operations on the shared resource so that no incorrect state is ever reached and it indicates the order in which operations on a shared resource can be interleaved. A path expression has the following form

path *S* **end**;

where *S* denotes possible execution histories. It is an expression whose variables are the operations on the resource and whose operators are:

Sequencing (;). It defines a sequencing order among operations. For example, **path** open; read; close; **end** means that an open must be performed first, followed by a read and a close in that order. There is no concurrency in the execution of these operations.

Selection (+). It signifies that only one of the operations connected by a + operator can be executed at a time. For example, **path** read + write **end** means that only read or only write can be executed at a given time, but the order of execution of these operations does not matter.

Concurrency ({}). It signifies that any number of instances of the operation delimited by { and } can be in execution at a time. For example, **path** {read} **end** means that any number of read operations can be executed concurrently. The path expression **path** write; {read} **end** allows either several read operations or a single write operation to be executed at any time (read and write operations exclude each other). However, whenever the system is empty after all readers have finished, the writer must execute first. Between every two write operations, at least one read operation must be executed. The path expression **path** {write; read} **end** means that at any time there can be any number of instantiations of the path write; read. At any instant, the number of read operations executed is less than or equal to the number of write operations executed. The path expression **path** {write + read} **end** is meaningless and does not impose any restriction on the execution of read and write operations.

Example 2.6. The following path expression gives the weak reader's priority solution to the readers-writers problem:

path {read} + write **end**

Clearly, this path expression allows either several read operations or a single write operation to be executed at any time (read and write operations exclude each other). Due to {read}, if a reader is reading, subsequent readers gain immediate access to the file. A waiting writer must wait until all readers have completed reading. This is a weak reader's priority solution because when a writer exits and both a reader and a writer are waiting, then any one of them can get access next randomly.

Example 2.7. A writer's priority solution is implemented by the following combination of path expressions (Note that when there is more than one path expression, the order of operations indicated by all the path expressions must be satisfied.):

path start-read +{ start-write ; write} **end**
path { start-read ; read} + write **end**

A reader executes start-read followed by read and a writer executes start-write followed by write. Start-write and start-read are dummy procedures used solely for achieving desired synchronization. Due to the second path expression, no reader can succeed in executing start-read operation when a writer is performing a write operation. However, due to the first path expression, a writer will be able to execute a start-write operation when a writer is doing a write operation or when a reader is doing a read operation. Thus, writers succeed in gaining priority over readers by executing a start-write operation. When a writer is done, a waiting writer gets to execute write first, and all waiting readers are blocked until no writer is left.

2.6.4 Communicating Sequential Processes (CSP)

Hoare [10] suggests that input and output commands can be treated as synchronization primitives in a programming language. In communicating sequential processes (CSP), concurrent processes communicate through input-output commands and are synchronized by requiring input and output commands to be synchronous. Communication occurs whenever (1) an input command in one process specifies the name of another process as its source, (2) an output command in the other process specifies the name of the first process as its destination, and (3) the target variable of the input command matches in type with the value denoted by the expression of the output command. Under these conditions, the processes are said to *correspond*. *Synchronous I/O* means that the command that happens to be ready first is delayed until the correspondin
occurs or is ready.