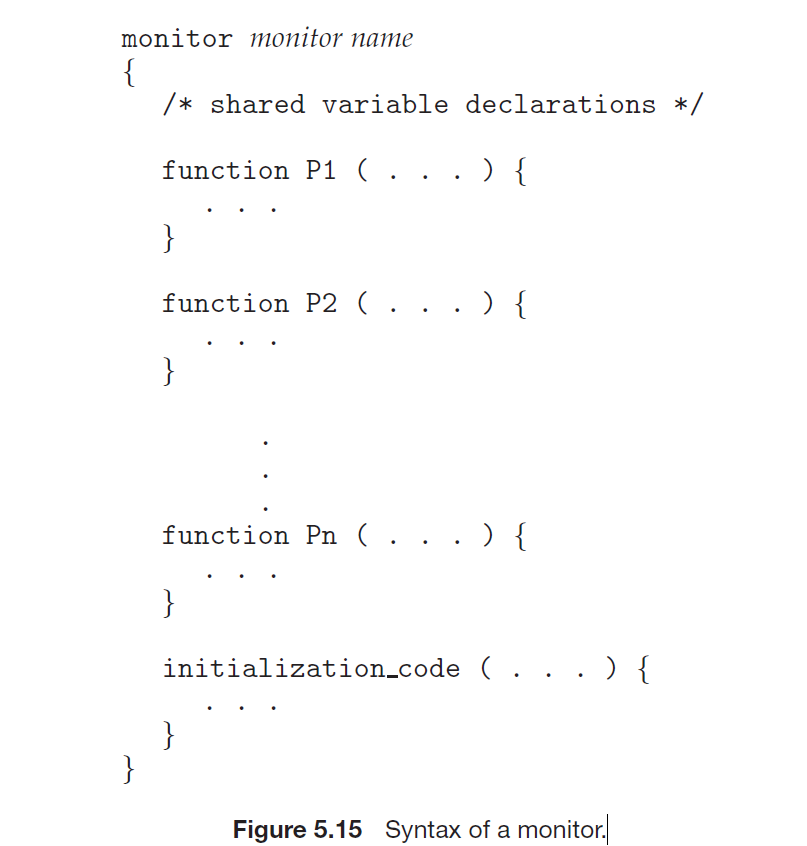
**MONITORS**

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect and also deadlocks occur. To overcome this problems monitor are used.

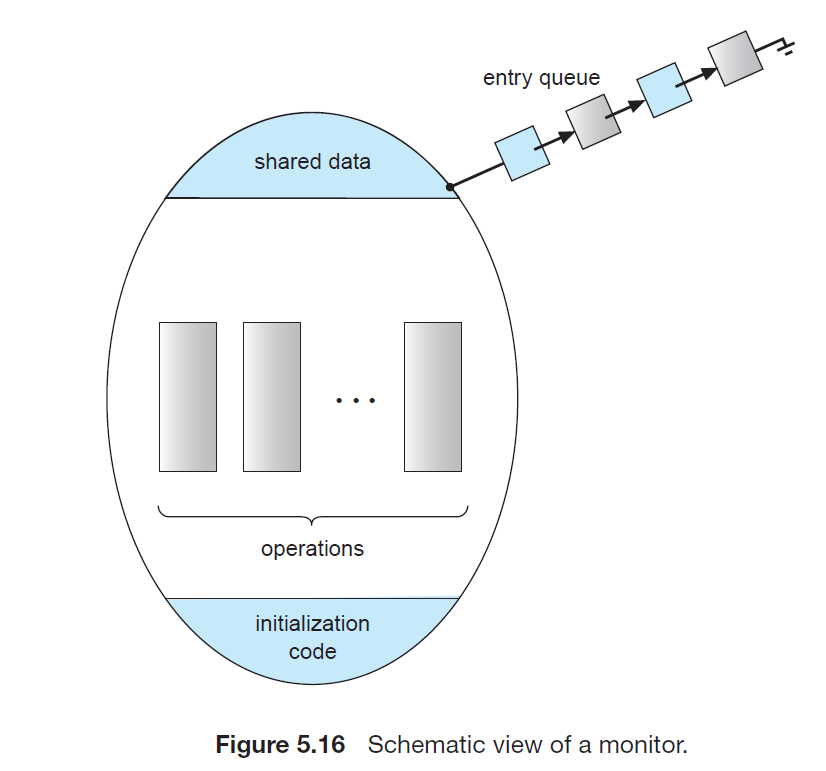
**1)Monitor Usage:**

A feature of programming languages called monitors helps control access to shared data. A monitor type is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. Several programming languages, including Java, C#, Visual Basic, Ada, and concurrent Euclid, support the use of monitors.

The syntax of a monitor type is shown below



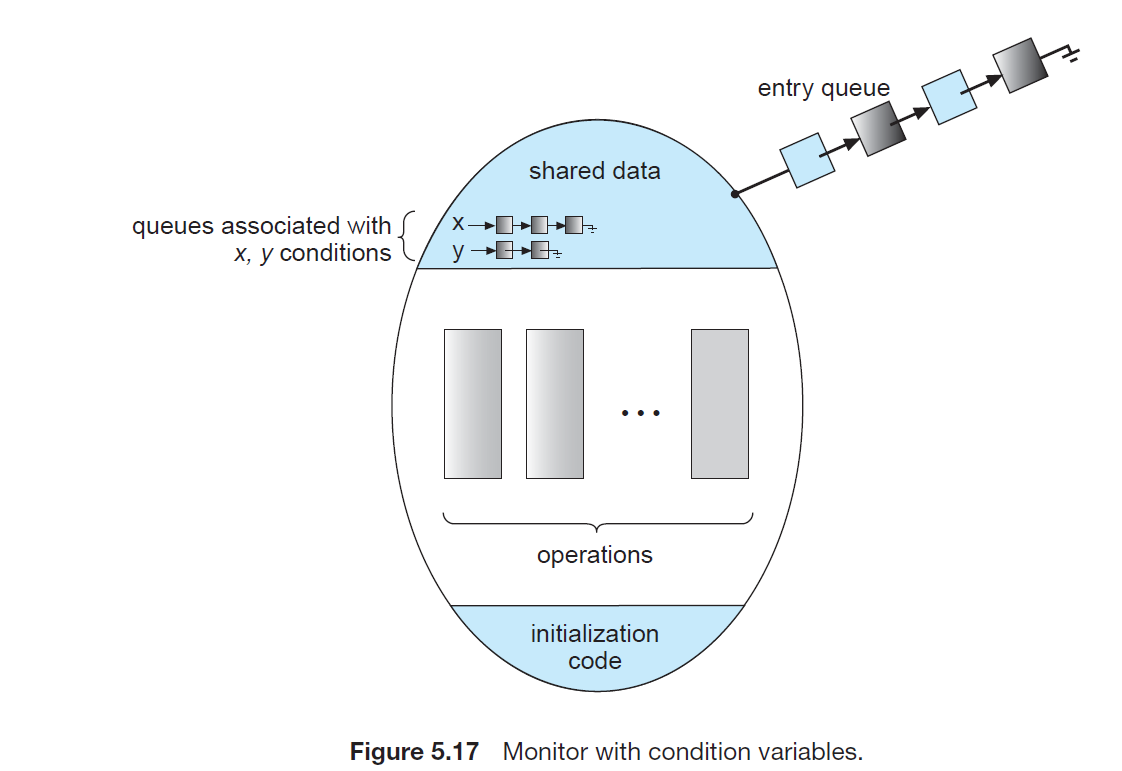
A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions. The monitor construct ensures that only one process at a time is active within the monitor.



However, the monitor construct, as defined so far, is not sufficiently powerful for modelling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct.

condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal(). The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x.signal(); The x.signal() operation resumes exactly one suspended process.



**2)Dining-Philosophers Solution Using Monitors:**

we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher i can set the variable state[i] = EATING only if her two neighbours are not eating: (state[(i+4) % 5] != EATING) and(state[(i+1)% 5] != EATING).

We also need to declare condition self[5]; This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

monitor DiningPhilosophers

{

enum {THINKING, HUNGRY, EATING} state[5];

condition self[5];

void pickup(int i)

{

state[i] = HUNGRY;

test(i);

if (state[i] != EATING)

self[i].wait();

}

void putdown(int i)

{

state[i] = THINKING;

test((i + 4) % 5);

test((i + 1) % 5);

}

void test(int i)

{

if ((state[(i + 4) % 5] != EATING) &&

(state[i] == HUNGRY) &&

(state[(i + 1) % 5] != EATING))

{

state[i] = EATING;

self[i].signal();

}

}

initialization code()

{

for (int i = 0; i < 5; i++)

state[i] = THINKING;

}

}

Figure: A monitor solution to the dining-philosopher problem.

Each philosopher, before starting to eat, must invoke the operation pickup().This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

DiningPhilosophers.pickup(i);

...

eat

...

DiningPhilosophers.putdown(i);

It is easy to show that this solution ensures that no two neighbours are eating simultaneously and that no deadlocks will occur.

**3) Implementing a Monitor Using Semaphores:**

Consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

Since a signalling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signalling processes can use next to suspend themselves. An integer variable next count is also provided to count the number of processes suspended on next.

Thus, each external function F is replaced by

wait(mutex);

...

body of F

...

if (next count > 0)

// if some process already waiting to take the monitor you signal the "next" semaphore and let it take the monitor.

signal(next);

else

signal(mutex);

Mutual exclusion within a monitor is ensured.

**4) Resuming Processes within a Monitor:**

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the conditional-wait construct can be used.

This construct has the form

x.wait(c);

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a priority number, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the ResourceAllocator monitor shown below, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

R.acquire(t);

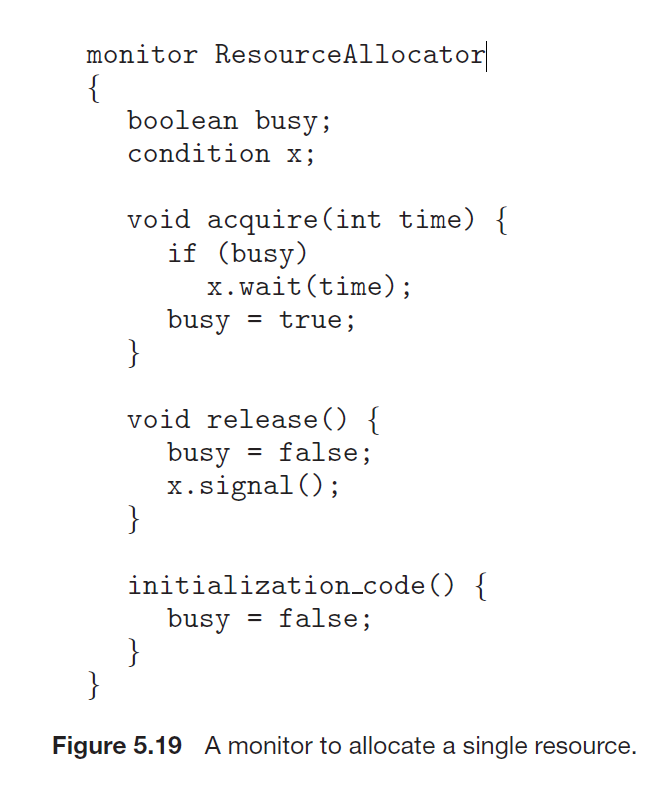
...

access the resource;

...

R.release();

where R is an instance of type ResourceAllocator.



Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

• A process might access a resource without first gaining access permission

to the resource.

• A process might never release a resource once it has been granted access

to the resource.

• A process might attempt to release a resource that it never requested.

• A process might request the same resource twice (without first releasing

the resource).

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the ResourceAllocator monitor and its managed resource. We must check two conditions to establish the correctness of this system.

i)user processes must always make their calls on the monitor in a correct sequence.

ii)we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols.

Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.