OPERATING SYSTEM

Process Synchronization

Process Synchronization

Synchronization is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet to communicate one process must perform some action such as setting the value of a variable or sending a message that the other detects. This only work if the event perform an action or detect an action that are constrained to happen in that order.

Thus one can view synchronization as a set of constrained on the ordering of events. The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy each constraints.

Functions of Process Synchronization

1. Cooperating processes: It made possible by multiprogramming, time sharing and parallel processing.
2. Multiple cooperating processes introduce potential problems like non determinacy in the computation, deadlocks, etc.
3. Race condition: The situations where several processes access and manipulate the same data concurrently, and the outcome of the execution depend on the particular order in which he access takes place.

Process Synchronization

Process synchronization techniques are :

- Mutual Exclusion
- Critical Section
- Semaphores
- Monitors

Mutual Exclusion

“ Each thread proceeds to update the shared variable, all others are excluded from doing so simultaneously.”

Processes frequently need to communicate with other processes. When a user wants to read from a file, it must tell the file process that it wants, then the file process has to inform the disk process to read the required block.

Processes that are working together often share some common storage that may be in main memory or it may be shared.

The key issue involving shared memory or shared files is to find to prohibit more than one process from reading and writing the shared data at the same time. What we need is mutual exclusion – some way of making sure that if one process in executing in its critical execution, the other processes will be excluded from doing the same thing.

The following pseudo code properly describes the email counting mechanism of mutual exclusion. Notice that we use the words Entermutualexclusion() and Exitmutualexclusion(). These words construct that encapsulate the thread's critical section.

When a thread want to enter its critical section, the thread must first execute Entermutualexclusion(). When a thread wants to exit the critical section it executes Exitmutualexclusion(). Because these constructs invoke the most fundamental operations inherent to mutual exclusion they are sometimes called mutual exclusion primitives.

Example:-

```
while(true)
{
Receive email                 //executing outside CR
Entermutualexclusion()        //want to enter CR
Increment mailcount           //executing inside CR
Exitmutualexclusion()         //leaving CR
}
```

Critical Section

Section of code that performs operations on a shared resource (e.g. writing data to shared variable). To ensure program correctness, at most one thread can simultaneously execute in its critical section.

Assume that thread T1 and T2 of the same process are executing in the system. Each thread manages email messages and each thread contains instructions that correspond to the preceding pseudo code.

When T1 reaches the Entermutualexclusion() line, the system must determine whether T2 is already in its critical section. If T2 is not in its critical section, then T1 enters its critical section, increments shared variable mailcount and executes Exitmutualexclusion() to indicate that the T1 has left its critical section. If, on the other hand, when T1 executes Entermutualexclusion() T2 is in its critical section, then T1 must wait until T2 executes Exitmutualexclusion(), at which point T1 enters its critical section.

If T1 and T2 simultaneously executes Entermutualexclusion(), then only one of the threads will be allowed to proceed, and one will be kept waiting.

Semaphores

"Mutual exclusion abstraction that uses two atomic operations (p and v) to access a protected integer variable that determines if threads may enter their critical section."

In mutual exclusion problem, the solution we presented did not solve all the problems of mutual exclusion. The Dutch mathematician Dekker is believed to be the first to solve the mutual exclusion problem but its original algorithm works for two processes only and it cannot be extended beyond that number.

To overcome this problem, synchronization tool called Semaphore was proposed by Dijksha, which gained wide acceptance and implemented in several operating system though system calls or as built-in functions.

A semaphore is a variable which accepts non-negative integer values and except for initialization may be accessed and manipulate through two primitive operations – Wait and Signal (originally defined as p and v respectively). These names comes from the Dutch words problem (to test) and verogen (to increment).

The two primitives take only argument as the semaphore variable and may be defined as follows –

1. wait(s):

   While s<=0 do(keep testing)

   S:=s-1

   Wait operation decrements the value of semaphore variable as soon as it would become non-negative

2. Signal(s):

   s:=s+1

   signal operation increment the value of semaphore variable.

Modification to integer value of the semaphore in the wait and signal operations are executed individually i.e. when one process modifies the semaphore no other process can simultaneously modify the same semaphore value.

In addition in the case of wait(s), the testing of integer value of s(s<=0) and its possible modification (s:=s-1) must also be executed without any interruption.

Monitors

- It is difficult to use semaphore level synchronization primitives for complex synchronization problems.

- Monitor is a high level synchronization construct. A monitor has –

  1. Shared data.
  2. A set of atomic operations on that data
  3. A set of condition variables.

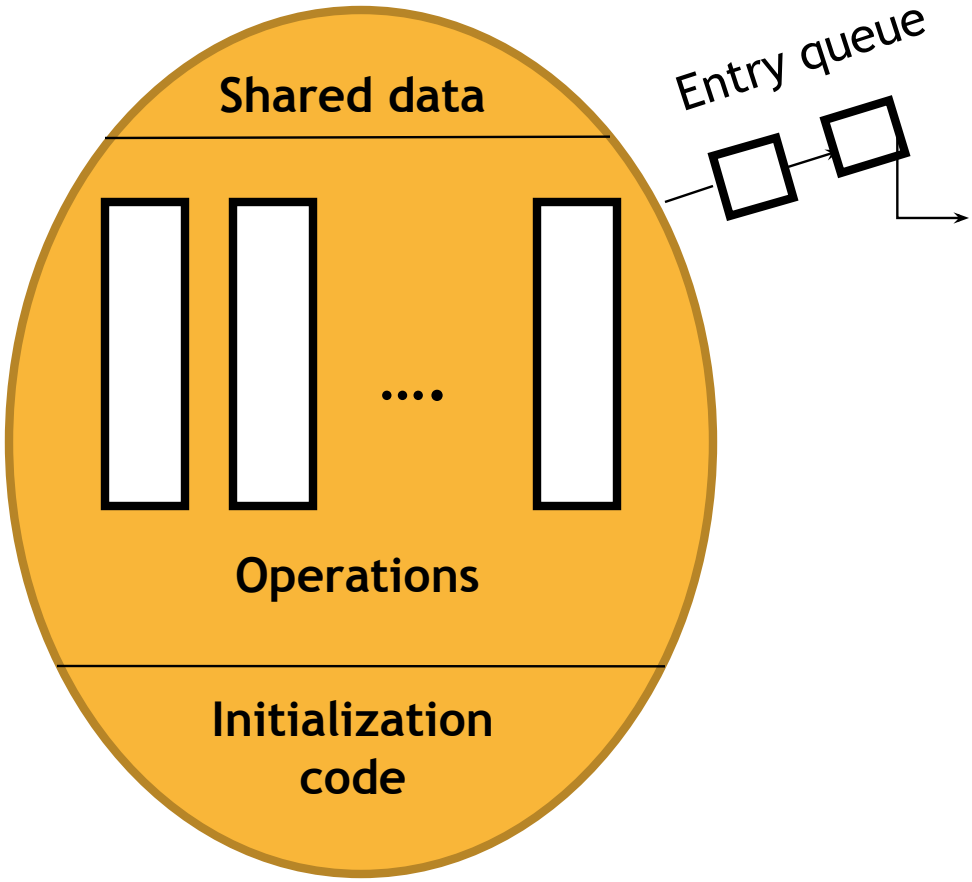
- Monitor has a set of programmer defined operators. They do not have to be implemented wholly within the O.S. Instead, the OS must provide some kind of synchronization primitives (semaphore, locks, etc) then monitors can be implemented as library code.

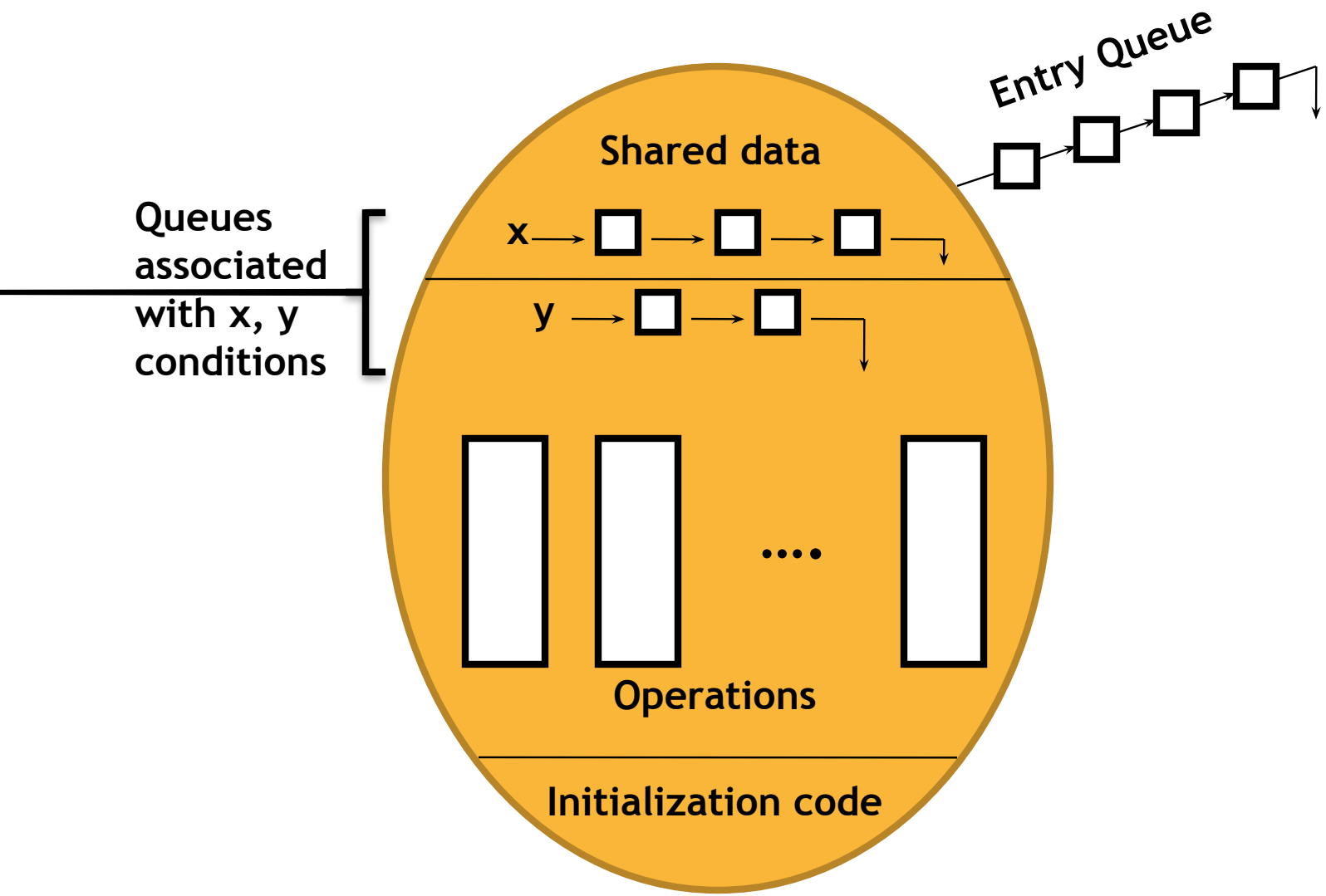
- Monitors simplify the complexity of synchronization problems by abstracting away the details. Abstract data types in which only a single process can be executing at a time. That means if one process calls a member function for the monitor, then it cannot be interrupted by another process that wants to call a member function in the same monitor.

- Sometimes, a process is executing away in a monitor and it discovers that it needs to have some other process do something to the monitor before it can proceed.(e.g. it might need some resources protected by he monitor).

- Rather than just backing out of the monitor call, the process is allowed to block itself waiting for the desired condition to come true. The condition variable is the mechanism for doing this. One interesting thing about condition variables is that they look just like events that can only be used by monitor functions

Fig. Schematic view of a monitor




Entry Queue
Shared data
x
y
Queues associated with x, y conditions
....
Operations
Initialization code


Thank You !