UNIT-2(PART-1)
CONCURRENCY

Process synchronization

Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes can either

- directly share a logical address space (that is, both code and data)
- or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency!

In this chapter, we discuss various mechanisms to ensure -

The orderly execution of cooperating processes that share a logical address space,

Producer Consumer Problem

A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.

Two kinds of buffers:

Unbounded buffer

Places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

Bounded buffer

Assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

counter variable = 0

counter is incremented every time we add a new item to the buffer counter++

counter is decremented every time we remove one item from the buffer counter--

----------Example----------

- Suppose that the value of the variable counter is currently 5.
- The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

Example

- Suppose that the value of the variable counter is currently 5.
- The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

"counter++" may be implemented in machine language (on a typical machine) as:

$$\begin{aligned} register_1 &= counter \\ register_1 &= register_1 + 1 \\ counter &= register_1 \end{aligned}$$

"counter--" may be implemented in machine language (on a typical machine) as:

$$\begin{aligned} register_2 &= counter \\ register_2 &= register_2 - 1 \\ counter &= register_2 \end{aligned}$$

critical- section problem

The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, \ldots, P_n\}$.

Each process has a segment of code, called a

critical section

in which the process may be changing common variables, updating a table, writing a file, and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

That is, no two processes are executing in their critical sections at the same time.

The critical-section problem is to design a protocol that the processes can use to cooperate.

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.

Figure: General structure of a typical process.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion:

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress:

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting:

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson‘s Solution

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

Peterson's solution requires two data items to be shared between the two processes:

int turn

→ Indicates whose turn it is to enter its critical section.

boolean flag [2]

→ Used to indicate if a process is ready to enter its critical section.

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

```
do {
    flag [i] = true ;
    turn = j ;
    while ( flag [ j ] && turn == [ j ] ) ;

        critical section

    flag [i] = false ;

        remainder section

    } while (TRUE) ;
```

Structure of process P_j in Peterson's solution

```
do {
    flag [ j ] = true ;
    turn = i ;
    while ( flag [ i ] && turn == [ i ] ) ;

        critical section

    flag [ j ] = false ;

        remainder section

    } while (TRUE) ;
```

semaphores

Semaphores

- Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.

Semaphores

- Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.
- Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait ( ) and signal ( ).

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other proces can simultaneously modify that same semaphore value.

Types of Semaphores:

1. Binary Semaphore:

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
That is, when one process modifies the semaphore value, no other proces can simultaneously modify that same semaphore value.

Types of Semaphores:

1. Binary Semaphore:

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

2. Counting Semaphore:

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

monitors

Monitors

- A high level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor type presents a set of programmer-defined operations that provide mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

Syntax of a Monitor

```
monitor monitor_name
{
// shared variable declarations
procedure P1 ( . . . ) {
. . .
}
procedure P2 ( . . . ) {
. . .
}
.
.
procedure Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```


- 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 procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

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.

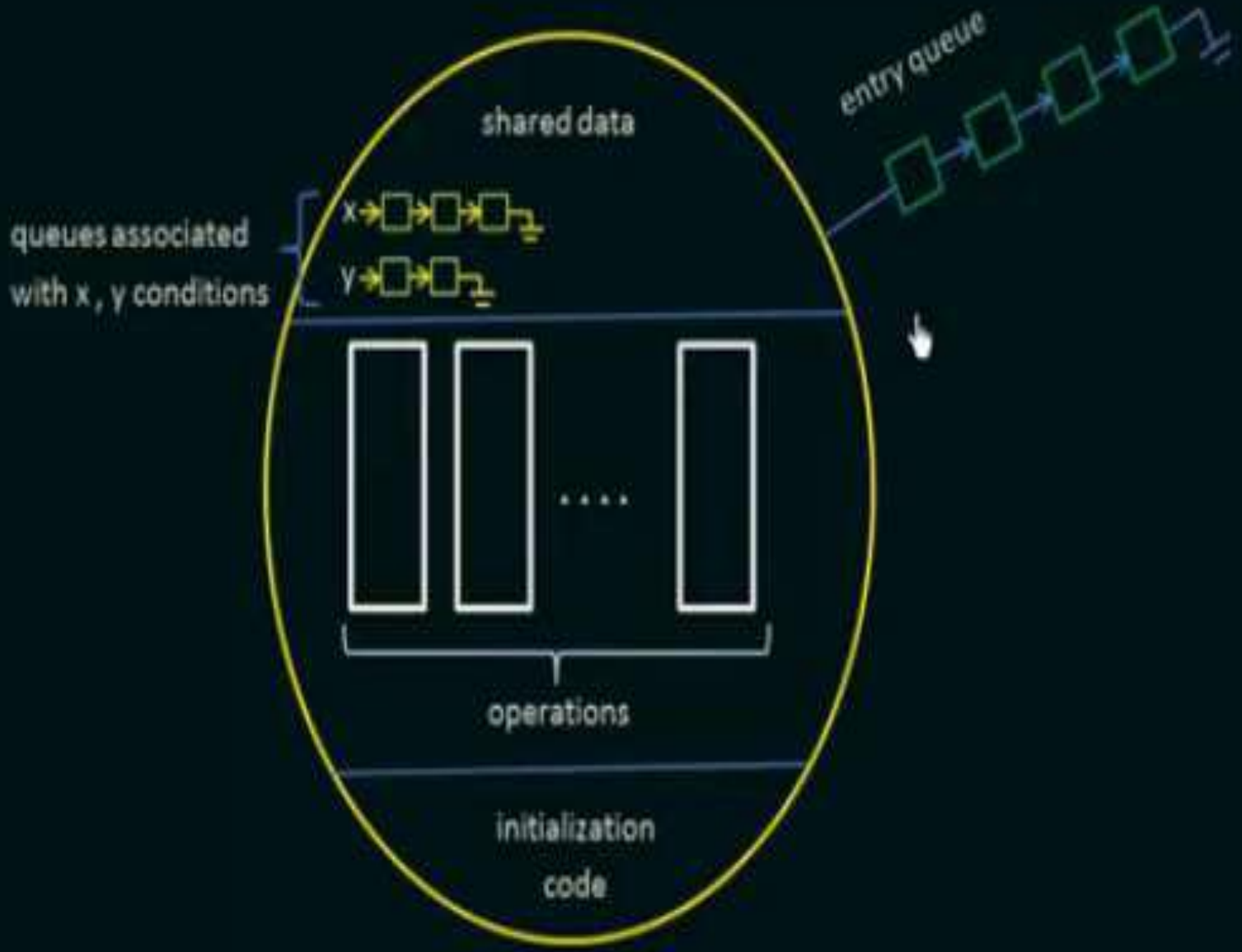

Fig: Schematic view of a monitor