Process Synchronization

1

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!

2

2

Producer Consumer Problem

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

- 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.

3

3

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.

4

4

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

$register_1$ = counter
$register_1$ = $register_1$ + 1
counter = $register_1$

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

$register_2$ = counter
$register_2$ = $register_2$ - 1
counter = $register_2$

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

5

5

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Clearly, we want the resulting changes not to interfere with one another. Hence we need process synchronization.

6

6

The Critical-Section Problem

Consider a system consisting of n processes { P_0, P_1, ..., 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

7

7

- 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.

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (TRUE);
```

Figure: General structure of a typical process.

8

8

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.

9

9

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"]

10

10

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 process 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.

11

11

Disadvantages of Semaphores

- The main disadvantage of the semaphore definition that was discussed is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

12

12

To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations.

- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.

13

13

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.

P0
wait (S);
wait (Q);
.
.
.
signal (S);
signal (Q);

P1
wait (Q);
wait (S);
.
.
.
signal (Q);
signal (S);

14

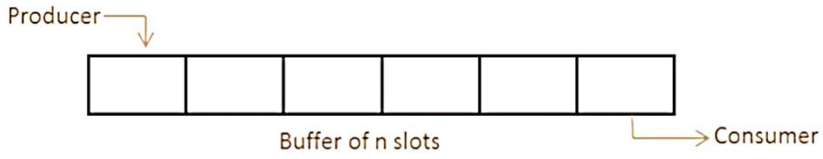
14

The Bounded-Buffer Problem

The Bounded Buffer Problem **(Producer Consumer Problem)**, is one of the classic problems of synchronization.

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, **Producer** and **Consumer**, which are operating on the buffer.

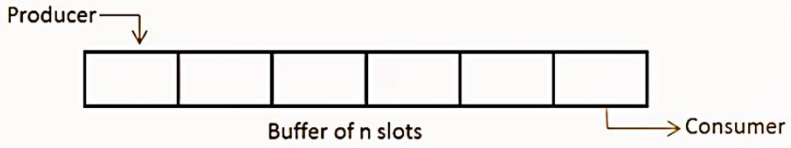


15

15

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, **Producer** and **Consumer**, which are operating on the buffer.



- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

16

16

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
               then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

17

17

If there are six processes operating simultaneously in the system that share resources, the shared resource allows only one process to access it at a time. The order in which the processes require the shared resource, the time each process needs the processor when in the critical section, and the priority of the processes are as follows:

The sequence of processes requiring the shared resource : A B C D E F.
The priority of the processes (1 being the highest priority) : 1 2 4 2 5 1.
The time each process needs to access the shared resource : 2 4 2 1 3 1.
Describe the execution sequence of processes when semaphores are used to synchronize these six processes.

18

18

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.

19

19

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.

20

20

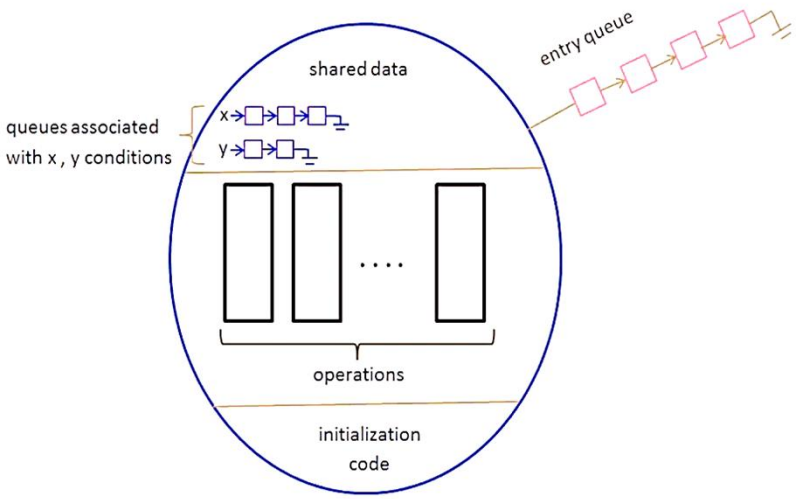


Fig: Schematic view of a monitor

21

21

```
type Bounded_buffer = monitor
Item buffer[n]; // buffer that stores items of type Item in n-sized buffer
int full = 0;
Condition buffer_full;
Condition buffer_empty;

monitor entry produce_info ( );
{
        If (full = n)
                wait (buffer_empty);
        produce ();
        full++;
        signal (buffer_full);
}
monitor entry consume_info ( );
{
        if (full = 0)
                wait (buffer_full);
        Consume ();
        full– –;
        signal (buffer_empty);
}
```

```
Producer ()
{
        Bounded_buffer B;

        while (true)
        {
                B.produce_info ();
        }
}
```

```
Consumer ()
{
        Bounded_buffer B;

        while (true)
        {
                B.consume_info ();
        }
}
```

Fig. Producer–consumer problem's solution with monitors

22

22