

What is Concurrency?

It refers to the execution of multiple instruction sequences at the same time. It occurs in an operating system when multiple process threads are executing concurrently. These threads can interact with one another via shared memory or message passing. Concurrency results in resource sharing, which causes issues like deadlocks and resource scarcity

Principle of Concurrency:

1. The way operating system handles interrupts
2. Other processes' activities
3. The operating system's scheduling policies

Problems in Concurrency:

1. Locating the programming errors
2. Sharing Global Resources
3. Locking the channel
4. Optimal Allocation of Resources

Issues in Concurrency/Some Related Term with Concurrency:

Deadlock: A situation in which two or more than two processes are unable to proceed because each one is waiting for the other to do something

Livelock: A situation in which two or more than two processes are continuously changing their state as response to changes in the processes without doing any useful work.

Race Condition: A situation in which multiple threads or processes read and write a shared data item and the final result is based on the relative timing of their execution.

Starvation: Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time

Advantages of Concurrency:

1. Better Performance
2. Better Resource Utilization
3. Running Multiple Applications

Disadvantages of Concurrency:

1. It is necessary to protect multiple applications from each other.

2. It is necessary to use extra techniques to coordinate several applications.
3. Additional performance overheads and complexities in OS are needed for switching between applications.

Race Condition:

A situation in which multiple threads or processes read and write a shared data item and the final result is based on the relative timing of their execution.

Example:

1. Consider two processes, P3 and P4, that share global variables b and c, with initial values $b = 1$ and $c = 2$.
2. At some point in its execution, P3 executes the assignment $b = b + c$. • At some point in its execution, P4 executes the assignment $c = b + c$.
3. Note that the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments.

Mutual Exclusion:

It is a property of concurrency control of the process. It prevent the race condition. It is a way of make sure that when one process is executing in the critical section than no other process can be enter in this critical section. It prevent the simultaneously access to shared resources.

Semaphore:

Semaphore is an integer value that solve the critical section problems. After initialization, it can only be accessed by two atomic operation.

Atomic Operation of Semaphores:

1. Wait()
2. Signal()

• definition of wait() is as follows:

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

• definition of signal () is as follows:

```
signal(S) {
    S++;
}
```

Solution of Critical Section Using Semaphores:

Let P1, P2, P3 and Pn are the process that want to go under critical sections

do {

wait(s):

// Critical Section

Signal(s):

// Remainder Section

} while(T)

Binary and Counting Semaphore Differences:

Counting Semaphore	Binary Semaphore
No mutual exclusion	Mutual exclusion
<u>Any integer value</u>	<u>Value only 0 and 1</u>
<u>More than one slot</u>	Only one slot
Provide a set of Processes	It has a mutual exclusion mechanism.

Weak and Strong Semaphores:

Strong/Weak Semaphores

☹ A queue is used to hold processes waiting on the semaphore

Strong Semaphores

- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- the order in which processes are removed from the queue is not specified

Bounded Buffer Problem/Producer-Consumer Problem:

Classic Problems of Synchronization (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.



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.

NESO ACADEMY

Solution to the Bounded Buffer Problem using Semaphores:

We will make use of three semaphores:

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)

NESO ACADEMY


Reader-Writer Problem

Introduction:

Classic Problems of Synchronization (The Readers-Writers Problem)

- A database is to be shared among several concurrent processes.
- Some of these processes may want **only to read the database**, whereas **others may want to update** (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as **Readers** and to the latter as **Writers**.
- Obviously, if two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.


To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.

Introduction > This synchronization problem is referred to as the readers-writers problem. 

Solution:

Solution to the Readers-Writers Problem using Semaphores:

We will make use of **two semaphores** and an **integer variable**:

1. **mutex**, a semaphore (initialized to 1) which is used to ensure mutual exclusion when **readcount** is updated i.e. when any reader enters or exit from the critical section.
2. **wrt**, a semaphore (initialized to 1) common to both reader and writer processes.
3. **readcount**, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object. 

Writer Process	Reader Process
<pre> do { /* writer requests for critical section */ wait(wrt); /* performs the write */ // leaves the critical section signal(wrt); } while(true); </pre>	<pre> do { wait (mutex); readcnt++; // The number of readers has now increased by 1 if (readcnt==1) wait (wrt); // this ensure no writer can enter if there is even one reader signal (mutex); // other readers can enter while this current reader is inside the critical section /* current reader performs reading here */ wait (mutex); readcnt--; // a reader wants to leave if (readcnt == 0) //no reader is left in the critical section signal (wrt); // writers can enter signal (mutex); // reader leaves } while(true); </pre>

NESO ACADEMY

Dining Philosopher Problem

Only two possibility

1. Thinking
2. Eating

Solution:

Maximum 2 person can eat at a time

One simple solution is to represent each fork/chopstick with a semaphore.

A philosopher tries to grab a fork/chopstick by executing a wait () operation on that semaphore.

He releases his fork/chopsticks by executing the signal () operation on the appropriate semaphores.

Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1.

where all the elements of chopstick are initialized to 1.

The structure of philosopher i

```
do {
    wait (chopstick [i] );
    wait(chopstick [ (i + 1) % 5] );
    ....
    // eat
    signal(chopstick [i] );
    signal(chopstick [(i + 1) % 5]);
    // think
}while (TRUE);
```

Although this solution guarantees that no two neighbors are eating simultaneously, it could still create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick. All the elements of chopstick will now be equal to 0.

When each philosopher tries to grab his right chopstick, he will be delayed forever.

For Avoid Deadlock:

Some possible remedies to avoid deadlocks:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available (to do this he must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.



Monitor in Operating System

Monitor is a module that contains:

- Shared data
- Procedure that operate on shared data

Syntax of Monitor:

Monitor {

 Condition Variable // operate on two atomic operation wait() and signal()

 Variable

 Procedure1 {

 // Code

 }

 Procedure2 {

 // Code

 }

}

