PROCESS SYNCHRONIZATION

Process synchronization in an operating system is a mechanism that ensures multiple processes or threads can execute concurrently in a safe and orderly manner while accessing shared resources, without causing race conditions, data inconsistency, deadlock, or starvation. It is essential in multi-threaded and multi-processing environments to maintain data integrity and correct program execution.

Key Synchronization Concepts:

1. Cooperating Processes

A cooperating process is a process that can affect or be affected by other processes executing in the system. Such processes share data or resources either directly through shared memory or indirectly through files or message passing. Because of this shared access, synchronization is necessary to maintain correctness.

2. Critical Section

A critical section is a part of the program where a process accesses shared resources (e.g., shared variables, memory, files). If multiple processes enter the critical section at the same time, it may lead to inconsistencies.

Solution: Ensure only one process enters the critical section at a time.

3. Race Condition

A race condition occurs when two or more processes try to read/write a shared resource. Simultaneously, leading to unpredictable behavior. Example: Two threads update a shared counter simultaneously without synchronization.

4. Mutual Exclusion

Ensures that only one process can enter the critical section at a time. Example: Using locks, semaphores, or monitors to prevent multiple processes from executing critical code simultaneously.

5. Deadlock

Deadlock occurs when two or more processes are stuck in a circular wait, each waiting for a resource held by another. Example: Process A holds Resource 1 and waits for Resource 2, while Process B holds Resource 2 and waits for Resource 1.

Solution: Deadlock prevention, detection, and avoidance strategies.

6. Starvation

Starvation happens when a process waits indefinitely because higher-priority processes keep executing before it. Example: In a scheduling system, if high-priority tasks keep arriving, a low priority process may never get CPU time.

Solution: Implementing fair scheduling algorithms like Round Robin.

7. Semaphore

A semaphore is a synchronization primitive used to control access to shared resources.

It is an integer variable that can be:

Binary (0 or 1) → Acts like a mutex (lock).

Counting (N resources available) → Used when multiple instances of a resource exist.

Operations:

Wait(S): Decrease the semaphore (block if S <= 0).

Signal(S): Increase the semaphore (release resource).

Definition of the wait () operation

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

Definition of the signal () operation

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

8. Mutex (Mutual Exclusion Lock)

A mutex is a binary semaphore (0 or 1) used for locking a critical section. Only one process can acquire the lock at a time. Example: Used in multi-threaded programming to prevent simultaneous modification of shared data.

Problems

1. Critical Section Problem

The Critical Section Problem is a fundamental synchronization problem where:

- Multiple processes (or threads) need to access a shared resource (such as a counter, bank balance, or shared variable).
- Only one process can access the critical section (the code segment that accesses the shared resource) at a time to prevent data inconsistency.
- If two or more processes enter the critical section simultaneously, it leads to a race condition, causing unpredictable or incorrect results.

Requirements for the Solution

A correct solution to the critical section problem must satisfy these three conditions:

Mutual Exclusion: Only one process can execute in its critical section at a time. If one process is inside the critical section, others must wait.

Progress: If no process is executing in the critical section and some processes wish to enter, the system must select one of them without unnecessary delay. Processes that want to enter the critical section should not be ignored indefinitely.

Bounded Waiting: There must be a limit on the number of times other processes can enter their critical sections after a process has requested entry. This ensures no process waits forever (prevents starvation).

Race Condition Explained

When multiple processes access and update a shared variable without synchronization, their operations may overlap.

For example, two processes might both read the same value of a counter, increment it, and write it back.

This leads to lost updates because increments are not atomic.

The final counter value will be less than expected, demonstrating a race condition.

Demonstration of Race Condition (Pseudocode)

Shared variable: counter = 0

Process Pi:

```
for i = 1 to N do
    temp = counter        // Read shared counter
    temp = temp + 1       // Increment locally
    counter = temp        // Write back to shared counter
end for
```

If multiple processes execute this code simultaneously, increments can be lost due to overlapping reads and writes.

Semaphore-Based Solution

To prevent race conditions and enforce mutual exclusion, we use a binary semaphore called mutex initialized to 1.

wait(mutex):

Decrements mutex. If mutex is 0, the process blocks and waits until it becomes positive.

signal(mutex):

Increments mutex, releasing the lock and allowing other waiting processes to enter the critical section.

Critical Section Algorithm with Semaphore (Pseudocode)

```
Initialize:
   semaphore mutex = 1
   shared variable counter = 0

Process Pi:
   for i = 1 to N do
      wait(mutex)              // Request access to critical section

      // Critical Section
      temp = counter
      temp = temp + 1
      counter = temp

      signal(mutex)            // Release critical section
   end for
```

Expected Outcome:

Without synchronization:

The final value of counter will often be less than N * number_of_processes due to race conditions.

With semaphore synchronization:

The final value of counter will be exactly N * number_of_processes, showing that all increments were correctly performed without interference.

2. Readers–Writers Problem

The Readers–Writers Problem is a classic synchronization problem where:

- Multiple readers can simultaneously read data from a shared database or resource.
- Writers need exclusive access to modify the shared data.
- Readers are allowed to read concurrently only if no writer is writing.

- Writers must wait if there are any active readers or other writers.

Requirements for the Solution

A correct solution to the readers–writers problem must satisfy these conditions:

- Mutual Exclusion for Writers:
  Only one writer can access the shared resource at a time.
  Writers must have exclusive access — no readers or other writers allowed during writing.
- Concurrent Reads:
  Multiple readers can access the resource simultaneously without interfering with each other.
- No Reader-Writer Conflicts:
  Readers and writers cannot access the shared resource simultaneously.
- Progress and Bounded Waiting:
  The system must ensure that neither readers nor writers are starved.
  Both get fair chances to access the resource within a bounded waiting time.

Race Condition Explained

Without proper synchronization:

- Writers can modify the data while readers are reading, causing inconsistent or corrupted data reads.
- Multiple writers could write simultaneously, causing data loss or corruption.
- This leads to race conditions and incorrect behavior.

Demonstration of Race Condition (Pseudocode)

Shared variable: data (e.g., a database record)
Reader process Pi:
Read data // No synchronization, could read partial or inconsistent data

Writer process Pj:
Write data // No synchronization, could overwrite while readers read
If multiple readers and writers execute without coordination, data inconsistency and race conditions occur.

Semaphore-Based Solution

To coordinate access, we use:

- A binary semaphore wrt to allow writers exclusive access.
- A binary semaphore mutex to protect the shared reader count.
- An integer readcount to keep track of active readers.

Readers–Writers Algorithm with Semaphores (Pseudocode)

Initialization:

```
semaphore mutex = 1
semaphore wrt = 1
int readcount = 0
```

Reader process Pi:

```
wait(mutex)              // Lock readcount update
readcount = readcount + 1
if readcount == 1 then
   wait(wrt)             // First reader locks writers
signal(mutex)            // Unlock readcount update

// Reading data (critical section for readers)

wait(mutex)              // Lock readcount update
readcount = readcount - 1
if readcount == 0 then
   signal(wrt)           // Last reader releases writers
signal(mutex)            // Unlock readcount update
```

Writer process Pj:

```
wait(wrt)                // Request exclusive access
```

// Writing data (critical section for writers)

signal(wrt) // Release exclusive access

Expected Outcome

Without synchronization:

Readers and writers accessing data concurrently cause inconsistent or corrupted data reads and writes.

With semaphore synchronization:

Multiple readers can read simultaneously without conflicts.

Writers get exclusive access to prevent interference.

Data consistency is maintained and race conditions are avoided.

3. Dining Philosophers Problem

The Dining Philosophers Problem is a classic process synchronization problem that demonstrates issues of mutual exclusion, deadlock, starvation, and race conditions.

Problem Description

Five philosophers sit around a circular table.

Each philosopher alternates between thinking and eating.

There is one chopstick between each pair of philosophers (total 5 chopsticks).

To eat, a philosopher must hold both the left and right chopsticks.

A chopstick is a shared resource, and only one philosopher can use a chopstick at a time.

The goal is to design a synchronization protocol so that philosophers can eat without causing:

Race conditions, Deadlock, Starvation

Requirements for the Solution

A correct solution to the Dining Philosophers Problem must satisfy the following conditions.

Mutual Exclusion

Only one philosopher can hold a chopstick at any time. Two philosophers must not use the same chopstick simultaneously.

Deadlock Avoidance

The system must prevent the situation where all philosophers hold one chopstick and wait forever for the second chopstick.

Starvation Freedom (Bounded Waiting)

Every philosopher who wants to eat must eventually be allowed to eat. No philosopher should wait indefinitely.

Progress

If chopsticks are available and a philosopher wants to eat, they should be allowed to proceed without unnecessary delay.

Race Condition and Deadlock Explained

Without proper synchronization:

- All philosophers may try to pick up chopsticks at the same time.
- If every philosopher picks up their left chopstick first, each will hold one chopstick and wait for the right chopstick.
- Since each right chopstick is held by another philosopher, no philosopher can proceed.
- This situation causes deadlock, and no philosopher can eat.
- Simultaneous attempts to access the same chopstick may also lead to race conditions.

Demonstration of Race Condition (Pseudocode)

```
Shared Resources
chopstick[0..4]     // Each chopstick is shared between two philosophers

Philosopher Pi
while true do
   Think

   wait until chopstick[i] is available
   pick up chopstick[i]                // Pick up left chopstick

   wait until chopstick[(i + 1) % 5] is available
   pick up chopstick[(i + 1) % 5]        // Pick up right chopstick

   Eat

   put down chopstick[i]
   put down chopstick[(i + 1) % 5]
end while
```

Problem with This Approach

- If all philosophers pick up their left chopstick simultaneously:
- Each philosopher holds one chopstick.
- Each waits forever for the right chopstick.
- Deadlock occurs.
- No philosopher can eat or release chopsticks.

Semaphore-Based Solution

To prevent race conditions, deadlock, and starvation, semaphores are used.

- Use one binary semaphore for each chopstick, initialized to 1.
- This ensures exclusive access to each chopstick.
- Break circular wait by changing the order of picking up chopsticks:
    - Even-numbered philosophers pick up the right chopstick first, then the left.
    - Odd-numbered philosophers pick up the left chopstick first, then the right.

This ordering prevents all philosophers from holding one chopstick and waiting indefinitely.

Dining Philosophers Algorithm with Semaphores (Pseudocode)

```
Initialization
for i = 0 to 4 do
   semaphore chopstick[i] = 1
end for

Philosopher Pi
while true do
   Think

   if (i is even) then
      wait(chopstick[(i + 1) % 5])      // Pick up right chopstick
      wait(chopstick[i])                // Pick up left chopstick
   else
      wait(chopstick[i])                // Pick up left chopstick
      wait(chopstick[(i + 1) % 5])      // Pick up right chopstick
   end if

   Eat
```

```
    signal(chopstick[i])                // Put down left chopstick
    signal(chopstick[(i + 1) % 5])      // Put down right chopstick
end while
```

Expected Outcome

Deadlock is avoided; philosophers do not wait forever holding one chopstick.

No philosopher starves; each philosopher eventually gets a chance to eat.

Chopsticks are never shared simultaneously.

Philosophers smoothly alternate between thinking and eating.

The system produces correct, synchronized, and deadlock-free behavior.