Synchronization in Operating Systems

1. Race Condition

A **race condition** occurs when multiple processes access and manipulate shared data **concurrently**, and the outcome depends on the sequence of execution.

Example:

- Two processes, **P1** and **P2**, increment a shared counter.
- If both read the value before one updates it, they may overwrite each other's update, leading to incorrect results.

Solution: Proper synchronization mechanisms are needed to avoid race conditions.

---

2. Critical Section Problem

A **critical section** is a part of a program where shared resources (data, files, etc.) are accessed.

Solution Requirements (Critical Section Problem Conditions)

1. **Mutual Exclusion** – Only one process can be in its critical section at a time.
2. **Progress** – If no process is in the critical section, others should proceed.
3. **Bounded Waiting** – No process should wait indefinitely to enter the critical section.

---

3. Peterson’s Algorithm

A software-based solution to **ensure mutual exclusion** in a two-process system.

Working of Peterson’s Algorithm

- Uses two variables:
    1. flag[i] = true (Indicates process wants to enter)
    2. turn = j (Indicates the other process's turn)

Algorithm

For **process P0**:

cpp

CopyEdit

flag[0] = true;

turn = 1;

while (flag[1] && turn == 1); // Wait

// Critical Section

flag[0] = false; // Exit critical section

For **process P1**:

cpp

CopyEdit

flag[1] = true;

turn = 0;

while (flag[0] && turn == 0); // Wait

// Critical Section

flag[1] = false; // Exit critical section

Advantages:

- Ensures **mutual exclusion**.
- Works only for **two processes**.

---

4. Bakery Algorithm

A generalization of Peterson's Algorithm for **multiple processes**.

Concept:

- Each process takes a **number** like a bakery queue system.
- The process with the **smallest number** enters the critical section first.

Algorithm Steps

1. Assign each process a **unique number**.
2. Compare numbers to decide execution order.
3. If two numbers are equal, process IDs break ties.

---

5. Synchronization Hardware

Modern processors provide **hardware-level** solutions for synchronization.

Locking Mechanisms

1. **Test and Set (TSL) Instruction**
   - Atomically checks and modifies a memory location.
2. **Compare and Swap (CAS)**
   - Compares a value and updates only if unchanged.
3. **Disable Interrupts**
   - Prevents context switching to ensure atomic execution.

---

6. Synchronization Software Tools

1. Mutex Lock

A **Mutex (Mutual Exclusion Lock)** is a binary lock (0 or 1) used to ensure only **one process** accesses a resource at a time.

Operations

- lock() – Acquires the lock.
- unlock() – Releases the lock.

Example:

cpp

CopyEdit

```
pthread_mutex_t lock;

pthread_mutex_lock(&lock);

// Critical section

pthread_mutex_unlock(&lock);
```

---

2. Semaphore

A **semaphore** is an integer variable used to **control access** to shared resources.

Types of Semaphores

1. **Binary Semaphore (0 or 1)**

  - Works like a Mutex.

2. **Counting Semaphore (0 to N)**
   - Allows multiple processes to access a resource.

Semaphore Operations

- **Wait (P) Operation:**

cpp

CopyEdit

```
wait(S) {
   while (S <= 0); // Busy wait
   S = S - 1;
}
```

- **Signal (V) Operation:**

cpp

CopyEdit

```
signal(S) {
   S = S + 1;
}
```

---

7. Classic Synchronization Problems

1. Bounded Buffer Problem (Producer-Consumer)

- **Producers** generate items and place them in a buffer.
- **Consumers** remove items and process them.
- **Issues**:
  - Buffer overflow (producer adds when full).
  - Buffer underflow (consumer removes when empty).

Solution:

- Use semaphores to synchronize access.

cpp

CopyEdit

```
Semaphore empty = N;  // Number of empty slots

Semaphore full = 0;   // Number of filled slots

Semaphore mutex = 1;  // Mutual exclusion


Producer:

wait(empty);

wait(mutex);

add item to buffer;

signal(mutex);

signal(full);


Consumer:

wait(full);

wait(mutex);

remove item from buffer;

signal(mutex);

signal(empty);
```

---

2. Readers-Writers Problem

- **Readers** can read **simultaneously**.
- **Writers** need **exclusive access**.

Solution:

- Maintain **reader count**.
- Ensure **writers get exclusive access**.

cpp

CopyEdit

```
Semaphore mutex = 1;  // Mutual exclusion for modifying reader count

Semaphore wrt = 1;    // Control access for writers

int read_count = 0;


Reader:

wait(mutex);

read_count++;

if (read_count == 1) wait(wrt);

signal(mutex);


perform reading;


wait(mutex);

read_count--;

if (read_count == 0) signal(wrt);

signal(mutex);


Writer:

wait(wrt);

write operation;

signal(wrt);
```

---

3. Dining Philosophers Problem

- **N philosophers** sit around a table.
- Each has **one fork**; needs **two forks** to eat.
- **Issues:** Deadlock, starvation.

Solution:

- Use semaphores for forks.
- Limit philosophers eating at once.

cpp

CopyEdit

```
Semaphore fork[N];

Philosopher(i):
wait(fork[i]);
wait(fork[(i+1)%N]);
eat();
signal(fork[i]);
signal(fork[(i+1)%N]);
```

---

8. Monitor

A **monitor** is a high-level synchronization construct that encapsulates:

- Shared data
- Operations on data
- Synchronization mechanisms

Example of Monitor in C++

cpp

CopyEdit

```
monitor SharedBuffer {
   int buffer;
   condition full, empty;

   void insert(int item) {
```

```
    if (buffer is full) wait(full);

    add item;

    signal(empty);

  }


  int remove() {

    if (buffer is empty) wait(empty);

    remove item;

    signal(full);

  }

};
```

---

9. Synchronization in Windows

Windows provides several synchronization tools:

- **Mutexes** (CreateMutex)
- **Semaphores** (CreateSemaphore)
- **Events** (CreateEvent)
- **Critical Sections** (EnterCriticalSection)

Example: Using Windows Mutex

cpp

CopyEdit

```
HANDLE mutex = CreateMutex(NULL, FALSE, NULL);

WaitForSingleObject(mutex, INFINITE);

// Critical Section

ReleaseMutex(mutex);
```

Conclusion

- **Synchronization** ensures correct execution of concurrent processes.
- **Race conditions** must be avoided using **locks, semaphores, and monitors**.
- **Classic problems** demonstrate real-world challenges in process coordination.