CO3

Session - 2

COURSE NAME: OPERATING SYSTEMS

COURSE CODE: 23CS2104R/A

Concurrency and Process Synchronization

AIM OF THE SESSION

To familiarize students with the basic concept of Concurrency and Process Synchronization.

INSTRUCTIONAL OBJECTIVES

This Session is designed to:

1. Demonstrate what is meant by Concurrency.
2. Demonstrate what is meant by a Process Synchronization.
3. Describe the need for Process Synchronization.

LEARNING OUTCOMES

At the end of this session, you should be able to:

1. Defines what Describe Concurrency and Process Synchronization.

CONCURRENCY IN OS

- Concurrency in operating systems refers to the ability of an operating system to handle multiple tasks or processes simultaneously.
- With the increasing demand for high-performance computing, concurrency has become a critical aspect of modern computing systems.
- Concurrency is essential in modern operating systems due to the increasing demand for multitasking, real-time processing, and parallel computing.
- Operating systems that support concurrency can execute multiple tasks simultaneously, leading to better resource utilization, improved responsiveness, and enhanced user experience.

PRINCIPLES OF CONCURRENCY

The principles of concurrency in operating systems are designed to ensure that multiple processes or threads can execute efficiently and effectively

INTERLEAVING

- Interleaving refers to the interleaved execution of multiple processes or threads.

SYNCHRONIZATION

- Synchronization refers to the coordination of multiple processes or threads to ensure they do not interfere.

PRINCIPLES OF CONCURRENCY

MUTUAL EXCLUSION

- Mutual exclusion refers to ensuring that only one process or thread can access a shared resource at a time.

DEADLOCK AVOIDANCE

- Deadlock is a situation in which two or more processes or threads are waiting for each other to release a resource, resulting in a deadlock.

PROCESS OR THREAD COORDINATION

- Processes or threads may need to coordinate their activities to achieve a common goal.

RESOURCE ALLOCATION

- Operating systems must allocate resources such as memory, CPU time, and I/O devices to multiple processes or threads fairly and efficiently.

CONCURRENCY MECHANISMS

These concurrency mechanisms are essential for managing concurrency in operating systems and are used to ensure safe and efficient access to system resources.

Process vs Threads: An operating system can support concurrency using processes or threads.

Synchronization Primitives: Operating systems provide synchronization primitives to coordinate access to shared resources between multiple processes or threads.

Scheduling Algorithms: Operating systems use scheduling algorithms to determine which process or thread should execute next.

Message Passing: Message passing is a mechanism used to communicate between processes or threads.

Memory Management: Operating systems provide memory management mechanisms to allocate and manage memory resources.

Interrupt Handlings: Interrupts are signals sent by hardware devices to the operating system, indicating that they require attention.

ADVANTAGES OF CONCURRENCY

Concurrency provides several advantages in operating systems, including

Improved Performance: Concurrency allows multiple tasks to be executed simultaneously, improving the overall performance of the system.

Resource Utilization: Concurrency allows better utilization of system resources, such as CPU, memory, and I/O devices.

Responsiveness: Concurrency can improve system responsiveness by allowing multiple tasks to be executed concurrently.

Scalability: Concurrency can improve the scalability of the system by allowing it to handle an increasing number of tasks and users without degrading performance.

Fault Tolerance: Concurrency can improve the fault tolerance of the system by allowing tasks to be executed independently. If one task fails, it does not affect the execution of other tasks.

PROBLEMS IN CONCURRENCY

These problems can be difficult to debug and diagnose and often require careful design and implementation of concurrency mechanisms to avoid.

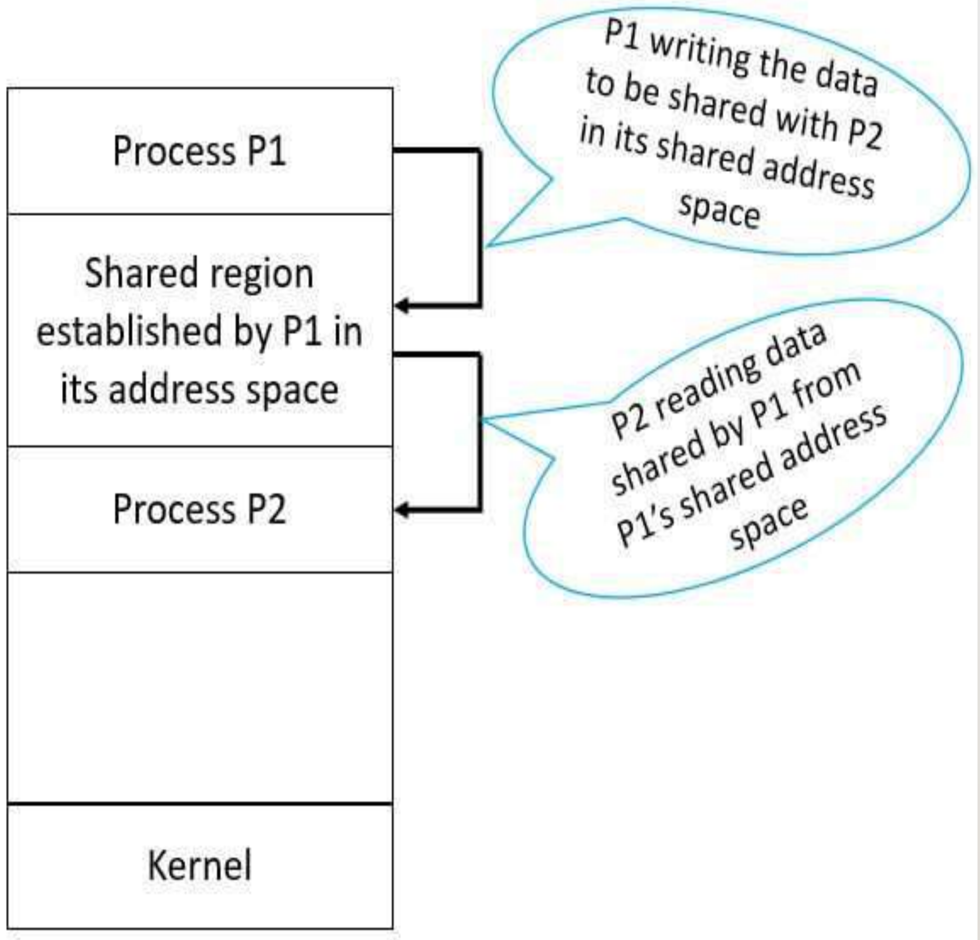
- **Sharing global resources**
  Sharing of global resources safely is difficult. If two processes both make use of a global variable and both perform read and write on that variable.
- **Optimal allocation of resources**
  It is difficult for the operating system to manage the allocation of resources optimally.
- **Locating programming errors**
  It is very difficult to locate a programming error because reports are usually not reproducible.
- **Locking the channel**
  It may be inefficient for the operating system to simply lock the channel and prevents its use by other processes.

COMMUNICATION IN CONCURRENCY

Concurrency is the execution of multiple instruction sequences at the same time. It happens in the operating system when several process threads are running in parallel. The running process threads always communicate with each other through shared memory or message passing.

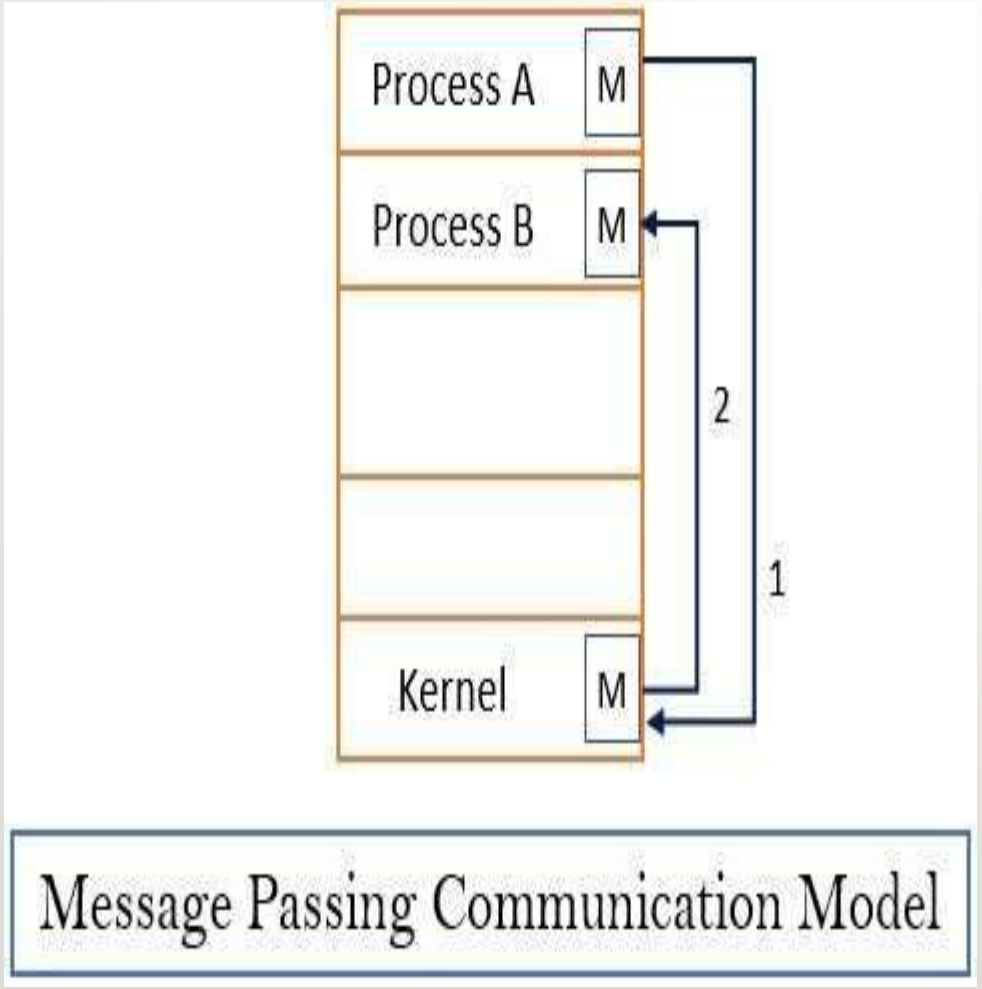
SHARED MEMORY SYSTEMS

- Inter process Communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically a shared memory region resides in the address space of the process creating the shared memory segment.
- Other processes that wish to communicate using this shared memory segment must attach it to their address space.
- Normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.


Process P1
Shared region established by P1 in its address space
Process P2
Kernel
P1 writing the data to be shared with P2 in its shared address space
P2 reading data shared by P1 from P1's shared address space

Shared Memory System

MESSAGE-PASSING SYSTEMS



Message Passing Communication Model

- Message passing provides a mechanism to allow processes to communicate and synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

PROCESS SYNCHRONIZATION

- Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.
- In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

PROCESS SYNCHRONIZATION

What is Process?

- A process is a program that is currently running or a program under execution is called a process. It includes the program's code and all the activity it needs to perform its tasks, such as using the CPU, memory, and other resources.

The process is categorized into two types on the basis of synchronization and these are given below:

1. Independent Process
2. Cooperative Process

1. Independent Processes

- Two processes are said to be independent if the execution of one process does not affect the execution of another process.

2. Cooperative Processes

- Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

BACKGROUND

- Processes can execute concurrently
    - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem: Suppose that we wanted to provide a solution to the producer-consumer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

PRODUCER-CONSUMER PROBLEM

- The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.
- There is one Producer in the producer-consumer problem, the Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed size.

PRODUCER-CONSUMER PROBLEM OR BOUNDED BUFFER PROBLEM

Producer

```
while (true) { /* produce an item in
                  next produced */
while (counter == BUFFER SIZE) ;
                  /* do nothing */
buffer[in] = next produced;
 in = (in + 1) % BUFFER SIZE;
counter++;
}
```

Consumer

```
while (true) {
while (counter == 0) ;   /* do nothing */
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
counter--;        /* consume the item in  next
                     consumed */
}
```

PRODUCER-CONSUMER PROBLEM

Before Starting an explanation of code, first, understand the few terms used in the above code:

- “**in**" used in a producer code represents the next **empty buffer**
- "**out**" used in consumer code represents a first **filled buffer**
- Count keeps the count number of elements in the buffer.
- Count is further divided into 3 lines of code represented in the block in both the producer and consumer code.

PRODUCER-CONSUMER PROBLEM

- The producer should produce data only when the buffer is not full.
- In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

NEED FOR PROCESS SYNCHRONIZATION

Race condition

A race condition occurs when two or more processes access and manipulate a shared resource simultaneously, leading to unexpected results. In this example, we will use a shared variable "counter" and two processes, "Process A" and "Process B".

Process A:

- Read the value of the counter
- Increment the value of the counter by 1
- Save the new value of the counter

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

Process B:

- Read the value of the counter
- Decrement the value of the counter by 1
- Save the new value of the counter

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

EXAMPLE OF RACE CONDITION

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer execute register1 = counter          {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = counter          {register2 = 5}
S3: consumer execute register2 = register2 – 1    {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4}
```

RACE CONDITION

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

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.

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

CRITICAL SECTION PROBLEM

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has a **critical section** segment of code
  - Process may be changing common variables, updating the table, writing a file, etc
  - When one process in the critical section, no other may be in its critical section
- ***The critical section problem*** is to design a protocol to solve this
- Each process must ask permission to enter the critical section in the **entry section**, may follow the critical section with the **exit section**, then the **remainder section**

CRITICAL-SECTION PROBLEM

```
do {
        Entry section
                Critical section
        Exit section
        Remainder section
    }
while (true);
```

The general structure of a typical process Pi with Critical Section.

ALGORITHM FOR PROCESS P_i

```
do {

        while (turn == j);

            critical section
        turn = j;

            remainder section
    } while (true);
```

SOLUTION TO THE CRITICAL SECTION PROBLEM

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

- **1. Mutual exclusion:** If process Pi 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 deciding 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

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **int turn;**
  - **Boolean flag[2]**

- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = *true*** implies that process $\mathbf{P_i}$ is ready!

ALGORITHM FOR PROCESS P_i

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

            critical section

        flag[i] = false;

            remainder section
    } while (true);
```

PETERSON'S SOLUTION (CONT.)

- Provable that the three CS requirements are met:
  1. Mutual exclusion is preserved

     P$_i$ enters CS only if:

     either **flag[j] = false** or **turn = i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

THANK YOU

Team – Operating System