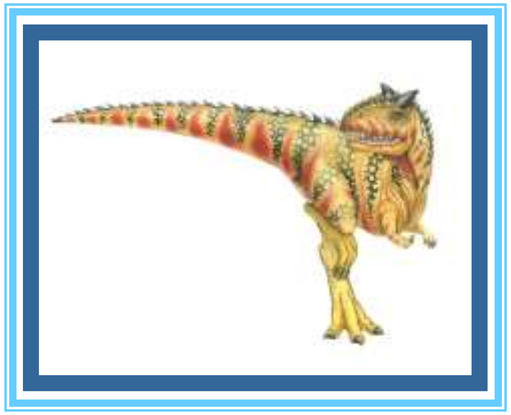
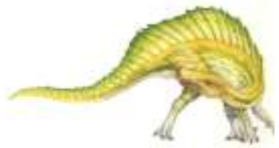
Synchronization Tools



Process Synchronization

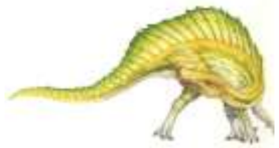
- Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner.
- Objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access.
- In a multi-process system, synchronization is necessary to ensure data consistency and integrity



Process Synchronization

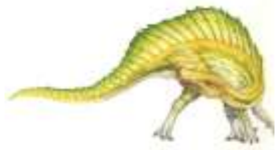
P1	**P2**
1)A=variable;	1) B=variable;
2)A++;	2)B--;
3) Sleep(1) ;	3)sleep(1);
4) variable=A;	4) variable=B;

Variable is used by both P1 and P2, initially variable=4



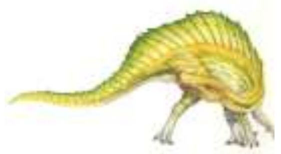
Process Synchronization

- **Expected Behaviour (No Race Condition)**
- If they ran **one after the other** (no overlap):
- **Case 1: P1 then P2**
  - P1: A=4 → A=5 → variable=5
  - P2: B=5 → B=4 → variable=4
  - Final result = **4**
- **Case 2: P2 then P1**
  - P2: B=4 → B=3 → variable=3
  - P1: A=3 → A=4 → variable=4
  - Final result = **4**



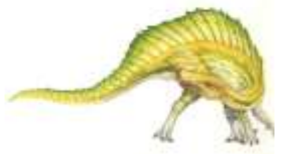
Process Synchronization

- **What Actually Happens (With Race Condition)**
- Because of sleep(1), both processes **pause before writing back**. This lets their operations overlap.
- **Interleaving Example (Race Condition):**
- variable = 4
- **P1 executes step 1:** A = 4
- **P2 executes step 1:** B = 4
- **P1 executes step 2:** A = 5
- **P2 executes step 2:** B = 3
- **Both sleep(1) at the same time**
- **P1 executes step 4:** variable = A = 5
- **P2 executes step 4:** variable = B = 3



Process Synchronization

- **Possible Outcomes**
- If **P1 writes last** → variable = 5
- If **P2 writes last** → variable = 3
- If they run in sequence → variable = 4 (correct)
- So final value of variable is **non-deterministic**: could be **3, 4, or 5** depending on timing.



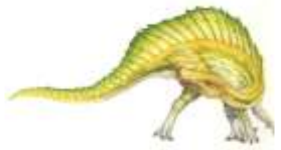
Solution

- mutex is a **binary semaphore**, initialized to **1** (available).
- Meaning:
  - mutex = 1 → critical section is **free**
  - mutex = 0 → some process is **inside**
- **wait(mutex)** checks if mutex > 0.
- If yes → decrements it (mutex = mutex - 1).
- If no → process is blocked until it becomes > 0.

When a process enters, mutex goes from **1** → **0**.

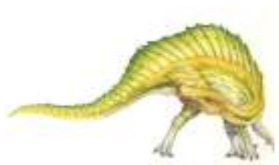
- **signal(mutex)** increments it (mutex = mutex + 1).
- So after exiting, mutex = 0 + 1 = 1.
- Now another waiting process can enter.

- wait(mutex);   // Lock critical section Process P1
- A = variable;
- A++;
- sleep(1);
- variable = A;
- signal(mutex);   // Unlock critical section
- //
- wait(mutex);   // Lock critical section Process P2
- B = variable;
- B--;
- sleep(1);
- variable = B;
- signal(mutex);   // Unlock critical section



Race Condition

- When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a **race condition.**
- Several processes access and process the manipulations over the same data concurrently, and then the outcome depends on the particular order in which the access takes place.



Critical Section

- **What is the Critical Section?**
- A **critical section** is a part of a program where a process/thread accesses **shared resources** (like variables, files, buffers).
- Since multiple processes may try to enter at the same time → if not controlled, you get **race conditions**.

Critical-Section Problem

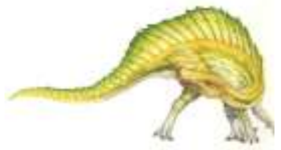
- **The Critical Section Problem**
- **Solution - How do we design a protocol so that:**
- **Mutual Exclusion** → At most **one process** is in the critical section at a time or If process $\boldsymbol{P_i}$ is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** → If no process is in the critical section, then one of the waiting processes should enter (no deadlock).
- **Bounded Waiting** → A process waiting for the critical section should get a chance within a **finite time** (no starvation) or A bound must exist 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

Classic Solutions

- Classic Solutions
  - **Synchronization tools**
    - *Semaphores* (counting & binary)
    - *Mutex locks*
    - *Monitors / Condition variables*

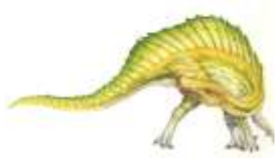
Mutex Locks

- Process synchronization plays an important role in maintaining the consistency of shared data.
- Mutex provide synchronization services
- Mutex is a mutual exclusion object that synchronizes access to a resource.
- It is created with a unique name at the start of a program. The mutex locking mechanism ensures only one process can acquire the mutex and enter the critical section. This process only releases the mutex when it exits in the critical section.



Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic (cannot be interrupted)**
- But this solution requires **busy waiting**



Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

            critical section

        release lock

remainder section
}
```

Semaphore

- A **semaphore** is a synchronization tool used to control access to shared resources in concurrent programming. Think of it as a counter + signaling mechanism.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

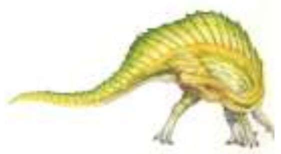
```
wait(S) { //equivalent to acquire
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) { //equivalent to release
    S++;
}
```

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems



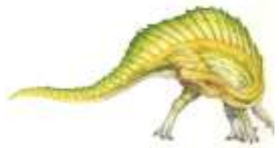
Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**`mutex`**" initialized to 1

```
wait(mutex);
   CS
signal(mutex);
```

A counting semaphore is initialized to **8**. **3** wait ( ) operations and **4** signal ( ) operations are applied. Find the current value of semaphore variable.

- Wait() operation decrements the value of a semaphore.
- Signal() operation increments the value of a semaphore.
- 3 wait() implies -3 to value of semaphore.
- 4 signal() implies +4 to value of semaphore.
- So 8-3+4=9 is the value of semaphore after 3 wait() and 4 signal() operations.



- Initial value of counting semaphore is 7. Then 20 P() operations and x V () operations are performed. Final value of semaphore is 5.Then the value of x will be

Final semaphore value= initial semaphore value-P() +V()

5 =7-20+x

x = 20+5-7

Hence x= 18

- At a particular time of computation, the value of a counting semaphore is 10. Then 12 P operations and “x” V operations were performed on this semaphore. If the final value of semaphore is 7, x will be:
  (A) 8 (B) 9 (C) 10 (D) 11
- Initially, the value of a counting semaphore is 10 Now 12 P operation are performed. Now counting semaphore value = -2 “x” V operations were performed on this semaphore and final value of counting semaphore = 7 i.e x + (-2) = 7 x = 9

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section

Suppose S = 1.

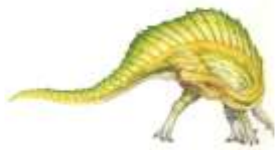
Process P1 executes wait(S):

checks S > 0 and is about to do S--.

Process P2 executes wait(S) at the same time.

Both see S > 0, both decrement → result: S = -1

This is exactly a **race condition inside the semaphore itself**.



Classical problems of synchronization Producer-Consumer Problem (Bounded Buffer Problem)

- **Producer**

```
while(true) {
//produces an item and put in next produced
while(count==Buffersize) ;
Buffer[in]=nextproduced;
in=(in+1) %Buffersize;
count++}
```

Consumer

```
while(true) {
while(count==0) ;
Nextconsumed=buffer[out];
out=(out+1) %Buffersize;
count--}
```

Classical problems of synchronization

Producer

count=count+1

A:R1=count

B:R1=R1+1

C:count=R1

Consumer

count=count-1
D:R2=count
E:R2=R2-1
F:count=R2

//

count=7

A:R1=7

B:R1=8

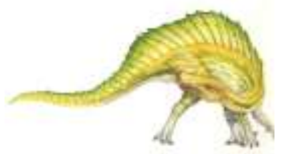
D:R2=7 (value of current count)
E:R2=6 (R2 decremented)

C:count=8 (value of current R1)
F:count=6 (value of current R2)

//race condition

Classic Problems of synchronization
Readers Writers Problem

- Consider a situation where we have a file shared between many people.
- If one of the person tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.



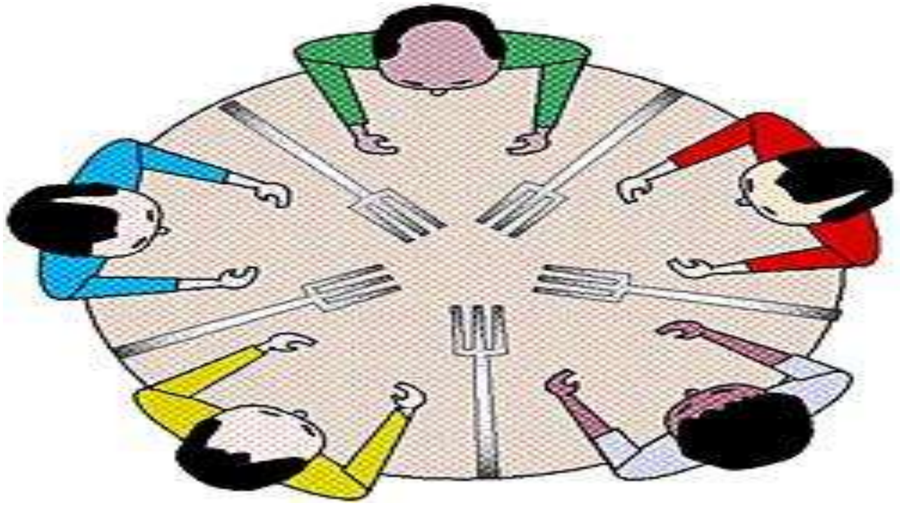
Classic Problems of synchronization Readers Writers Problem

- **Readers-Writers problem**
- Problem parameters:
- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Classic problems of synchronization- Dining philosophers problem

- There is a round dining table and there are 5 philosophers and 5 chopstick/fork. Philosopher can be in two states
- 1) Eat 2) Think
- Thinking- While thinking he will not interact with anyone.
- Eating- When hungry, he will eat. While eating he take 2 forks that are adjacent to him and starts eating. When finishes eating he place fork in place. Forks are limited



Classic problems of synchronization- Dining philosophers problem

- **Criteria**
- No two philosophers that are adjacent to each other should try to eat at the same time.
- Cannot pick 2 forks at same time (he should pick one and then the other fork)
- One cannot pick a fork that is already in the hand of a neighbour.
- When a hungry philosopher finishes his eating he will place both the forks back.
- The problem is that since the forks are limited no two philosophers should be allowed to access same fork at the same time (limited resources should be shared among processes in a synchronized manner).

Classic problems of synchronization- Dining philosophers problem

- The dining philosopher's problem leads to deadlock when every philosopher simultaneously picks up their left fork and then attempts to pick up their right fork, which is held by their neighbor.
- This creates a circular wait where no philosopher can acquire both forks needed to eat, resulting in a standstill where everyone is waiting for a resource held by someone else.
- Since every philosopher is waiting for a fork that their neighbor is holding, and no one will release their fork until they have both, the system enters a state where all philosophers are waiting indefinitely.