Experiment Title: Concurrency

Aim/Objective: Students should be able to understand the concepts of Concurrency. It helps in techniques like coordinating the execution of processes, memory allocation, and execution scheduling for maximizing throughput.

Description:

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. Concurrency results in the sharing of resources resulting in problems like deadlocks and resource starvation.

Prerequisite:

- **Basic idea on concurrent data structures Linked lists and queues**
- **Basic idea on concurrent hash tables**
- **Producer and consumer problems**
- **Dining-Philosophers problem**

Pre-Lab Task:

Concept	FUNCTIONALITY
Concurrency	Simultaneous execution of multiple tasks to improve efficiency.
Semaphores	Control access to resources, ensuring mutual exclusion and synchronization.

Concept	FUNCTIONALITY
Dining-Philosophers problem	Synchronize resource sharing to avoid deadlocks and starvation.
Producer – Consumer problem	Coordinates data production and consumption using shared buffers.
Readers-Writers Problem	Manages resource access fairness between readers and writers.

In Lab Task:

1. Write a C program to implement the Producer-Consumer problem.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 10

sem_t empty_count, full_count;
int buffer[BUFFER_SIZE];
int buffer_in = 0, buffer_out = 0;

void *producer(void *arg) {
  while (1) {
    int item = rand() % 100;

    sem_wait(&empty_count);

    buffer[buffer_in] = item;
    buffer_in = (buffer_in + 1) % BUFFER_SIZE;

    sem_post(&full_count);
  }
}

void *consumer(void *arg) {
  while (1) {
    sem_wait(&full_count);

    int item = buffer[buffer_out];
    buffer_out = (buffer_out + 1) % BUFFER_SIZE;

    sem_post(&empty_count);

    printf("Consumed item: %d\n", item);
  }
}
```

```
int main() {
   sem_init(&empty_count, 0, BUFFER_SIZE);
   sem_init(&full_count, 0, 0);

   pthread_t producer_thread, consumer_thread;

   pthread_create(&producer_thread, NULL, producer, NULL);
   pthread_create(&consumer_thread, NULL, consumer, NULL);

   pthread_join(producer_thread, NULL);
   pthread_join(consumer_thread, NULL);

   sem_destroy(&empty_count);
   sem_destroy(&full_count);

   return 0;
}
```

2. Write a C program to implement the Dining Philosopher problem.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM_PHILOSOPHERS 5

sem_t chopsticks[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
    int philosopher_id = (int)arg;

    while (1) {
        sem_wait(&chopsticks[philosopher_id]);

        sem_wait(&chopsticks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);

        sem_post(&chopsticks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);

        sem_post(&chopsticks[philosopher_id]);
    }
}

int main() {
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&chopsticks[i], 0, 1);
    }

    pthread_t philosopher_threads[NUM_PHILOSOPHERS];

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_create(&philosopher_threads[i], NULL, philosopher, (void *)i);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosopher_threads[i], NULL);
    }
```

```
  for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
      sem_destroy(&chopsticks[i]);
  }

  return 0;
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Data and Results

Data

Dining philosophers with chopsticks to avoid deadlocks using semaphores for synchronization, ensuring fairness.

Result

Each philosopher alternates between thinking and eating without causing deadlocks.

Analysis and Inferences:

Analysis

Efficiently demonstrates synchronization, preventing race conditions and deadlocks with semaphore-based mutual exclusion.

Inferences

The solution ensures fair and deadlock-free dining for philosophers.

Post Lab:

1. Write a Program to implement the Sleeping Barber problem in C.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_CUSTOMERS 10
#define NUM_CHAIRS 5

sem_t barber_chair;
sem_t waiting_room;

pthread_mutex_t customer_counter_mutex;

int customer_counter = 0;

void *barber(void *arg) {
  while (1) {
    sem_wait(&customer_counter_mutex);
    while (customer_counter == 0) {
      sem_post(&customer_counter_mutex);
      sem_wait(&barber_chair);
    }
    customer_counter--;
    sem_post(&customer_counter_mutex);

    sem_post(&waiting_room);

    printf("The barber is cutting the customer's hair.\n");
    sleep(1);
  }
}

void *customer(void *arg) {
  sem_wait(&waiting_room);

  sem_wait(&customer_counter_mutex);
  customer_counter++;
  sem_post(&customer_counter_mutex);
```

```
        sem_post(&barber_chair);

    sem_wait(&waiting_room);
    printf("The customer has finished their haircut.\n");
}

int main() {
    sem_init(&barber_chair, 0, 1);
    sem_init(&waiting_room, 0, NUM_CHAIRS);
    pthread_mutex_init(&customer_counter_mutex, NULL);

    pthread_t barber_thread;
    pthread_create(&barber_thread, NULL, barber, NULL);

    pthread_t customer_threads[NUM_CUSTOMERS];
    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        pthread_create(&customer_threads[i], NULL, customer, NULL);
    }

    pthread_join(barber_thread, NULL);
    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        pthread_join(customer_threads[i], NULL);
    }

    sem_destroy(&barber_chair);
    sem_destroy(&waiting_room);
    pthread_mutex_destroy(&customer_counter_mutex);

    return 0;
}
```

2. Write a program to implement the Reader-Writers problem in C

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_READERS 5
#define NUM_WRITERS 2

sem_t read_count;
sem_t write_mutex;

int reader_count = 0;

void *reader(void *arg) {
  while (1) {
    sem_wait(&read_count);
    reader_count++;
    sem_post(&read_count);

    printf("Reader is reading the shared resource.\n");
    sleep(1);

    sem_wait(&read_count);
    reader_count--;
    sem_post(&read_count);

    if (reader_count == 0) {
      sem_post(&write_mutex);
    }
  }
}

void *writer(void *arg) {
  while (1) {
    sem_wait(&write_mutex);

    printf("Writer is writing to the shared resource.\n");
    sleep(1);
```

```
 sem_post(&write_mutex);
    }
}

int main() {
    sem_init(&read_count, 0, 1);
    sem_init(&write_mutex, 0, 1);

    pthread_t reader_threads[NUM_READERS];
    pthread_t writer_threads[NUM_WRITERS];

    for (int i = 0; i < NUM_READERS; i++) {
        pthread_create(&reader_threads[i], NULL, reader, NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_create(&writer_threads[i], NULL, writer, NULL);
    }

    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(reader_threads[i], NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writer_threads[i], NULL);
    }

    sem_destroy(&read_count);
    sem_destroy(&write_mutex);

    return 0;
}
```

Sample VIVA-VOCE Questions (In-Lab):

1. Explain in detail the Dining Philosopher Problem.

Philosophers share chopsticks to eat. Synchronization prevents deadlock, starvation, and contention.

2. Explain in detail Reader’s writer's Problem?

Readers share access; writers need exclusive access. Use semaphores for fairness and synchronization.

3. Explain in detail the principles of Concurrency?

- Synchronization: Coordinate processes.
- Mutual Exclusion: Prevent simultaneous resource access.
- Deadlock Prevention: Avoid circular dependency.

4. Explain in detail the Advantages of Concurrency.

- Better resource utilization.
- Parallel execution.
- Scalability for multi-core.
- Improved responsiveness.

5. Explain in detail the Disadvantages of Concurrency.

- Complex debugging.
- Risk of deadlocks.
- Data inconsistency.
- Increased overhead.

Evaluator Remark (if any):	**Marks Secured_____ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce before signing and posting marks for each experiment.