Name-Abhi Mehta
Reg No.-221080001

Experiment-05

Aim-Write a program to solve dining philosophers problem using odd and even Philosopher with left and right chopstick pickup approach.

Theory-

In operating systems, semaphores are a synchronization mechanism used to control access to shared resources in a concurrent environment, such as multiple processes or threads.Semaphores can be thought of as counters with associated atomic operations. They maintain a non-negative integer value and support two fundamental operations: **wait (also known as P or down)** and **signal (also known as V or up).**

- **Wait Operation (wait, P, or down):** When a process or thread wants to access a shared resource, it executes a wait operation on the semaphore associated with that resource. If the semaphore's value is greater than zero, it decrements the value and continues its execution. If the value is zero, indicating that the resource is currently being used, the process or thread is blocked until the semaphore's value becomes greater than zero.

- **Signal Operation (signal, V, or up)**: When a process or thread finishes using a shared resource, it executes a signal operation on the semaphore associated with that resource. This operation increments the semaphore's value. If there are other processes or threads waiting for the resource (i.e., blocked on a wait operation), one of them is unblocked, allowing it to proceed.

- **Semaphores can be categorized into two types based on their value range:**

  Binary Semaphore: A binary semaphore can only take two integer values, typically 0 and 1. It is often used as a mutex (mutual exclusion) to control access to a single resource shared among multiple processes or threads. When initialized to 1, it acts as a lock, allowing only one process or thread to access the resource at a time. When initialized to 0, it indicates that the resource is currently being used and blocks other processes or threads until it becomes available.

Counting Semaphore: A counting semaphore can take any non-negative integer value. It can be used to control access to multiple instances of a resource or to limit the number of concurrent processes or threads accessing a resource. Counting semaphores are typically used to solve synchronization problems involving multiple resources, such as the producer-consumer problem or the dining philosophers problem.

Semaphores provide a flexible and powerful mechanism for coordinating the activities of multiple processes or threads in a concurrent system, helping to prevent race conditions, deadlocks, and other synchronization issues.

Dining Philosophers Problem-

The dining philosophers problem is a classic synchronization problem used to illustrate challenges in concurrent programming, particularly in managing shared resources and preventing deadlock.

In the dining philosophers problem, a number of philosophers sit at a round dining table with a bowl of spaghetti in front of each of them. Between each pair of adjacent philosophers, there is a single chopstick. To eat, a philosopher must pick up the two chopsticks adjacent to them. However, due to the limited number of chopsticks (equal to the number of philosophers), the philosophers must share the chopsticks and coordinate their actions to avoid deadlock and ensure that everyone gets a chance to eat.The problem can be summarized as follows:

- Each philosopher alternates between thinking and eating. While thinking, they do not require any resources. However, to eat, they must acquire two chopsticks (one from their left and one from their right).
- Philosophers are not allowed to speak to each other; they can only communicate by taking or releasing chopsticks.
- A deadlock can occur if all philosophers pick up the chopstick on their right simultaneously and then wait indefinitely for the chopstick on their left (since it's being held by their neighbor). In this case, no philosopher can proceed with eating, leading to a deadlock.

To solve the dining philosophers problem, various synchronization techniques can be employed, such as semaphores, mutexes, or monitors. One common solution is to introduce a rule that philosophers can only pick up both chopsticks simultaneously. This prevents deadlock by ensuring that a philosopher can only eat if both chopsticks are available. Additionally, strategies like resource ordering or limiting the number of

philosophers allowed to pick up chopsticks concurrently can be employed to prevent deadlock and starvation.
The dining philosophers problem serves as an essential illustration of concurrency issues in distributed systems and the need for proper synchronization mechanisms to ensure correct and efficient operation. It's often used in computer science courses and textbooks to introduce concepts such as deadlock, livelock, mutual exclusion, and synchronization.

Code-

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

#define Philosopher 5

sem_t chopsticks[Philosopher];

void *philosopher(void *arg) {
   int phil_id = *(int *)arg;
   int left_chop, right_chop;

   if ((phil_id & 1) == 0) {
       left_chop = (phil_id + 1) % Philosopher;
       right_chop = phil_id;
   } else {
       left_chop = phil_id;
       right_chop = (phil_id + 1) % Philosopher;
   }

   while (1) {
       printf("Philosopher %d is thinking.\n", phil_id);
       sleep(2);

       printf("Philosopher %d is hungry.\n", phil_id);

       sem_wait(&chopsticks[left_chop]);
       printf("Philosopher %d  picked up left chopstick. \n", phil_id);
```

```
        sem_wait(&chopsticks[right_chop]);
        printf("Philosopher %d  picked up right chopstick. \n", phil_id);

        printf("Philosopher %d is eating. \n", phil_id);
        sleep(2);

        sem_post(&chopsticks[right_chop]);
        printf("Philosopher %d put down right chopstick.\n", phil_id);

        sem_post(&chopsticks[left_chop]);
        printf("Philosopher %d put down left chopstick.\n", phil_id);
    }
}

int main() {
    pthread_t philosophers[Philosopher];
    int phil_ids[Philosopher];

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

    for (int i = 0; i < Philosopher; i++) {
        phil_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &phil_ids[i]);
    }

    for (int i = 0; i < Philosopher; i++) {
        pthread_join(philosophers[i], NULL);
    }

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

    return 0;
}
```

Screenshots of Code-

os5.2.c > philosopher(void *)

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

#define Philosopher 5

sem_t chopsticks[Philosopher];

void *philosopher(void *arg) {
    int phil_id = *(int *)arg;
    int left_chop, right_chop;

    if ((phil_id & 1) == 0) {
        left_chop = (phil_id + 1) % Philosopher;
        right_chop = phil_id;
    } else {
        left_chop = phil_id;
        right_chop = (phil_id + 1) % Philosopher;
    }

    while (1) {
        printf("Philosopher %d is thinking.\n", phil_id);
        sleep(2);

        printf("Philosopher %d is hungry.\n", phil_id);

        sem_wait(&chopsticks[left_chop]);
        printf("Philosopher %d  picked up left chopstick. \n", phil_id);

        sem_wait(&chopsticks[right_chop]);
        printf("Philosopher %d  picked up right chopstick. \n", phil_id);

        printf("Philosopher %d is eating. \n", phil_id);
        sleep(2);

        sem_post(&chopsticks[right_chop]);
        printf("Philosopher %d put down right chopstick.\n", phil_id);

        sem_post(&chopsticks[left_chop]);
        printf("Philosopher %d put down left chopstick.\n", phil_id);
    }
}
```

```c
int main() {
    pthread_t philosophers[Philosopher];
    int phil_ids[Philosopher];

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

    for (int i = 0; i < Philosopher; i++) {
        phil_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &phil_ids[i]);
    }

    for (int i = 0; i < Philosopher; i++) {
        pthread_join(philosophers[i], NULL);
    }

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

    return 0;
}
```

Output-

```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 0 is hungry.
Philosopher 0  picked up left chopstick.
Philosopher 0  picked up right chopstick.
Philosopher 0 is eating.
Philosopher 1 is hungry.
Philosopher 2 is hungry.
Philosopher 2  picked up left chopstick.
Philosopher 2  picked up right chopstick.
Philosopher 2 is eating.
Philosopher 3 is hungry.
Philosopher 4 is hungry.
Philosopher 0 put down right chopstick.
Philosopher 0 put down left chopstick.
Philosopher 0 is thinking.
Philosopher 4  picked up left chopstick.
Philosopher 4  picked up right chopstick.
Philosopher 4 is eating.
Philosopher 1  picked up left chopstick.
Philosopher 2 put down right chopstick.
Philosopher 2 put down left chopstick.
Philosopher 2 is thinking.
Philosopher 1  picked up right chopstick.
Philosopher 1 is eating.
Philosopher 3  picked up left chopstick.
Philosopher 0 is hungry.
Philosopher 4 put down right chopstick.
Philosopher 4 put down left chopstick.
Philosopher 4 is thinking.
Philosopher 3  picked up right chopstick.
Philosopher 3 is eating.
Philosopher 2 is hungry.
Philosopher 1 put down right chopstick.
Philosopher 1 put down left chopstick.
Philosopher 1 is thinking.
Philosopher 0  picked up left chopstick.
Philosopher 0  picked up right chopstick.
Philosopher 0 is eating.
Philosopher 4 is hungry.
Philosopher 3 put down right chopstick.
Philosopher 3 put down left chopstick.
Philosopher 3 is thinking.
Philosopher 1 is hungry.
Philosopher 0 put down right chopstick.
Philosopher 0 put down left chopstick.
Philosopher 0 is thinking.
Philosopher 2  picked up left chopstick.
Philosopher 2  picked up right chopstick.
```

```
Philosopher 2 is eating.
Philosopher 4  picked up left chopstick.
Philosopher 4  picked up right chopstick.
Philosopher 4 is eating.
Philosopher 1  picked up left chopstick.
Philosopher 3 is hungry.
Philosopher 0 is hungry.
Philosopher 4 put down right chopstick.
Philosopher 4 put down left chopstick.
Philosopher 4 is thinking.
Philosopher 2 put down right chopstick.
Philosopher 2 put down left chopstick.
Philosopher 2 is thinking.
Philosopher 3  picked up left chopstick.
Philosopher 3  picked up right chopstick.
Philosopher 3 is eating.
Philosopher 1  picked up right chopstick.
Philosopher 1 is eating.
Philosopher 4 is hungry.
Philosopher 4  picked up left chopstick.
Philosopher 2 is hungry.
Philosopher 1 put down right chopstick.
Philosopher 1 put down left chopstick.
Philosopher 3 put down right chopstick.
Philosopher 4  picked up right chopstick.
Philosopher 4 is eating.
Philosopher 0  picked up left chopstick.
Philosopher 2  picked up left chopstick.
Philosopher 2  picked up right chopstick.
Philosopher 2 is eating.
Philosopher 1 is thinking.
Philosopher 3 put down left chopstick.
Philosopher 3 is thinking.
Philosopher 4 put down right chopstick.
Philosopher 4 put down left chopstick.
Philosopher 4 is thinking.
Philosopher 0  picked up right chopstick.
Philosopher 0 is eating.
```

So on…..

Conclusion-

In conclusion, the dining philosophers problem tackled using the odd-even approach and semaphores presents an effective strategy for ensuring fairness and preventing deadlock among philosophers vying for access to chopsticks. By categorizing philosophers into two groups odd or even, and employing semaphores to control access to chopsticks, this approach helps to address the concurrency challenges inherent in the dining philosophers problem.

Overall, the provided code offers a practical and efficient solution to the dining philosophers problem, demonstrating proper concurrency control and synchronization techniques using pthreads and semaphores. It effectively addresses the challenges of coordinating multiple processes in a concurrent environment and ensures that the dining philosophers can dine peacefully without encountering deadlock or starvation.