**Experiment No: 06**

**TITLE:** Dining Philosophers Problem (Using Semaphore or mutex)

**OBJECTIVE:**

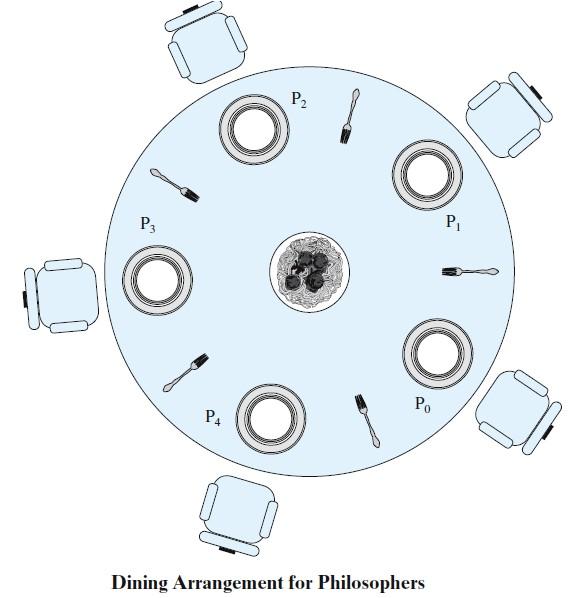
1. To understand the deadlock and starvation problem using programming.
2. To study dining philosophers problem of operating system.

**THEORY**:

**The Dining Philosophers Problem**

The dining philosophers problem was introduced by Dijkstra. Five philosophers live in a house, where a table is laid for them. The life of each philosopher consists principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti. Due to a lack of manual skill, each philosopher requires two forks to eat spaghetti.

The eating arrangements are simple: a round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti.

The problem: devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation. This problem may not seem important or relevant in itself. However, it does illustrate basic

problems in deadlock and starvation. Furthermore, attempts to develop solutions reveal many of the difficulties in concurrent programming. In addition, the dining philosophers problem can be seen as representative of problems dealing with the coordination of shared resources, which may occur when an application includes concurrent threads of execution. Accordingly, this problem is a standard test case for evaluating approaches to synchronization.

**Solution Using Semaphores**

Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table.

semaphore fork [5] = {1}; int i;

void philosopher (int i)

{

while (true)

{

think();

wait (fork[i]);

wait (fork [(i+1) mod 5]); eat();

signal(fork [(i+1) mod 5]); signal(fork[i]);

}

}

void main()

{

parbegin (philosopher (0), philosopher (1), philosopher (2), philosopher (3), philosopher (4));

}

**FAQs**

1. What is the difference between mutex and binary semaphore?
2. How many number of maximum philosophers will this problem have?
3. What is deadlock?what are the possible conditions for deadlock?
4. Explain the strategies for deadlock prevention.
5. Explain deadlock avoidance with with bankers algorithm.
6. Explain deadlock detection algorithm.
7. Explain dining philosophers problem in detail .
8. Explain solution for dining philosophers by using semaphore.

#include<stdio.h>

#include<semaphore.h>

#include<pthread.h>

#define N 5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

#define LEFT (ph\_num+4)%N

#define RIGHT (ph\_num+1)%N

sem\_t room;

sem\_t chopst[N];

void \* philospher(void \*num);

void take\_chopst(int);

void put\_chopst(int);

void test(int);

//int try;

int state[N];

int phil\_num[N]={0,1,2,3,4};

int main()

{

int i;

pthread\_t thread\_id[N];

sem\_init(&room,0,1);

for(i=0;i<N;i++)

sem\_init(&chopst[i],0,0);

for(i=0;i<N;i++)

{

pthread\_create(&thread\_id[i],NULL,philospher,&phil\_num[i]);

printf("Philosopher %d is thinking\n",i+1);

}

for(i=0;i<N;i++)

pthread\_join(thread\_id[i],NULL);

sem\_destroy(&chopst[N]);

sem\_destroy(&room);

pthread\_exit(0);

}

void \*philospher(void \*num)

{int x=1;

while(x<5)

{

int \*i = num;

usleep(1);

take\_chopst(\*i);

usleep(1);

put\_chopst(\*i);

x++;

}

}

void take\_chopst(int ph\_num)

{

sem\_wait(&room);

state[ph\_num] = HUNGRY;

printf("Philosopher %d is Hungry\n",ph\_num+1);

test(ph\_num);

sem\_post(&room);

sem\_wait(&chopst[ph\_num]);

usleep(1);

}

void test(int ph\_num)

{

if (state[ph\_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)

{

state[ph\_num] = EATING;

usleep(1);

printf("Philosopher %d takes left chopst %d and right chopstick %d\n",ph\_num+1,LEFT+1,ph\_num+1);

printf("Philosopher %d is Eating\n",ph\_num+1);

sem\_post(&chopst[ph\_num]);

}

}

void put\_chopst(int ph\_num)

{

sem\_wait(&room);

state[ph\_num] = THINKING;

printf("Philosopher %d putting chopst %d and %d down\n",ph\_num+1,LEFT+1,ph\_num+1);

printf("Philosopher %d is thinking\n",ph\_num+1);

test(LEFT);

test(RIGHT);

sem\_post(&room);

}