Laboratory 6

Title of the Laboratory Exercise: Solution to Dining Philosopher problem using Semaphore

1. Introduction and Purpose of Experiment

   In multitasking systems, simultaneous use of critical section by multiple processes leads to data inconsistency and several other concurrency issues. By solving this problem students will be able to use semaphore for synchronisation purpose in concurrent programs.

2. Aim and

   Objectives Aim

   - To develop concurrent programs using semaphores

   Objectives

   At the end of this lab, the student will be able to

   - Use semaphore
   - Apply appropriate semaphores in different contexts
   - Develop concurrent programs using semaphores

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Question

Implement the Dining Philosopher problem using POSIX threads

Calculations/Computations/Algorithm:

```
Function philosopher(var I : integer) : void
begin
   while(j++ < 10)
   begin
      take_forks(i);
      put_forks(i);
 end
end

Function take_forks(var I : integer) : void
begin
   sem_wait(&mutex);
   state[i] : = hungry;
   print “HUNGRY”
   test(i);
   sem_post(&mutex);
   sem_wait(&s[i]);
end

function put_forks(var I  : integer) : void
begin
   sem_wait(&mutex);
   state[i] = thinking;
   print “thinking”
   test(left);
   test(right);
   sem_post(&mutex);
end

function test(var I : integer)
begin
   if(state[i] == hungry && state[left]!=eating && state[right]!=eating)
    begin
      state[i] = eating;
      sem_post(&s[i]);
   end
end
```

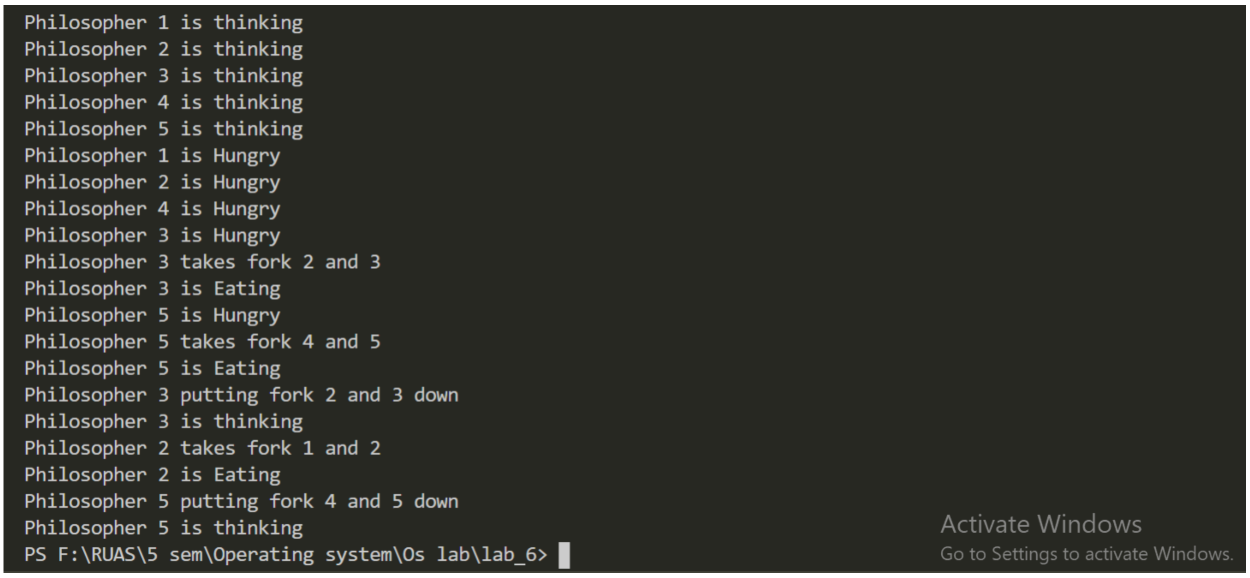
Implementation in c:-

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

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

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
   if (state[phnum] == HUNGRY
      && state[LEFT] != EATING
      && state[RIGHT] != EATING) {
      // state that eating
      state[phnum] = EATING;

      sleep(2);

      printf("Philosopher %d takes fork %d and %d\n",
               phnum + 1, LEFT + 1, phnum + 1);

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

      // sem_post(&S[phnum]) has no effect
      // during takefork
      // used to wake up hungry philosophers
      // during putfork
      sem_post(&S[phnum]);
   }
}

// take up chopsticks
void take_fork(int phnum)
{

   sem_wait(&mutex);

   // state that hungry
   state[phnum] = HUNGRY;

   printf("Philosopher %d is Hungry\n", phnum + 1);
```

```
    // eat if neighbours are not eating
    test(phnum);

    sem_post(&mutex);

    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);

    sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
        phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philospher(void* num)
{

    while (1) {

        int* i = num;

        sleep(1);

        take_fork(*i);

        sleep(0);

        put_fork(*i);
    }
}

int main()
{

    int i;
    pthread_t thread_id[N];
```

```
  // initialize the semaphores
  sem_init(&mutex, 0, 1);

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

    sem_init(&S[i], 0, 0);

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

    // create philosopher processes
    pthread_create(&thread_id[i], NULL,
            philospher, &phil[i]);

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

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

    pthread_join(thread_id[i], NULL);
}
```

5. Presentation of Results:



```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 4 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
PS F:\RUAS\5 sem\Operating system\Os lab\lab_6>
```

6. Analysis and Discussions

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available.Otherwise a philosopher puts down their chopstick and begin thinking again. The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Solution of Dining Philosophers Problem:

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.
The structure of the chopstick is shown below −
semaphore chopstick [5];
Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows –

```
do {
  wait( chopstick[i] );
  wait( chopstick[ (i+1) % 5] );
  . .
  . EATING THE RICE
  .
  signal( chopstick[i] );
  signal( chopstick[ (i+1) % 5] );
  .
  . THINKING
  .
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.
After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

But this approach can lead to a deadlock if all the philosophers pick up their left forks at the same time. This can be solved by changing the order of taking the forks for any one of the philosophers. This approach has been implemented in the program.

7. Conclusions

The dining and drinking philosopher's problem are a very old and important problem in the distributed computing field. It was first introduced by Dijkstra and then used by many other researches as a general problem for illustrating mutual exclusion and resource sharing and allocation problem. A lot of algorithms have been introduced to resolve this problem with many options and assumptions which makes each proposed algorithm suitable for specific applications. It was introduced the problem with some of the fundamental and very old solutions for it in the Introduction section. The problem is implemented in C. It was executed and verified.

8. Comments

1. Limitations of Experiments

The perfectness of the implemented solution cannot be experimentally verified.

2. Limitations of Results

The results do not show the current state of the other philosophers in code.

3. Learning happened

The method to develop the Dining Philosopher problem using Semaphore and POSIX threads was learned. The applications and contexts of semaphores to develop concurrent programs was also learned.