Name: K Srikanth Registration Number: 17ETCS002124

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

5. Calculations/Computations/Algorithms

Algorithm

Function test(I):

If (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {

1. *Record state that philosopher is eating*
2. *eat for a while*
3. *if blocked signal the philosopher I to eat*

Function take_fork(I):

1. *lock the critical section*
2. *change the state to hungry*
3. *try to acquire two forks and eat*
4. *exit critical region*
5. *wait for a while*

Function put_fork(I):

1. *lock the critical section*
2. *change the state to thinking*
3. *put down the forks*
4. *let philosopher I think*
5. *check if left philosopher can now eat*
6. *check if right philosopher can now eat*
7. *unlock critical section*

Funtion philosopher(I) :

While(true)

1. *let philosopher think*
2. *try to acquire forks*

3. *eat*
4. *put down the forks*

Main function():

1. *start*
2. *declare N threads: philosophers*
3. *initialize first philosopher to be hungry*
4. *for i=1 to N-1:*

   initialize philosopher I to be thinking

5. *for i=0 to N-1:*

   create thread I with parameter I

6. *for i=0 to N-1:*

   Join thread I

7. *end*

6. Presentation of Results

Code

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
// 17ETCS00124 K Srikanth
// All the Global Explaination
#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];

```

Figure 1 C Code for the given problem statement

Function test

```
20  void test(int phnum)
21  {
22      if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
23          state[phnum] = EATING;
24          sleep(2);
25          printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1);
26          printf("Philosopher %d is Eating\n", phnum + 1);
27          sem_post(&S[phnum]);
28      }
29  }
```

Figure 2 C Code for the given problem statement continued

Function take_fork

```
30  void take_fork(int phnum)
31  {
32      sem_wait(&mutex);
33      state[phnum] = HUNGRY;
34      printf("Philosopher %d is Hungry\n", phnum + 1);
35      test(phnum);
36      sem_post(&mutex);
37      sem_wait(&S[phnum]);
38      sleep(1);
39  }
```

Figure 3 C Code for the given problem statement continued

Function put_fork

```
41  void put_fork(int phnum)
42  {
43      sem_wait(&mutex);
44      state[phnum] = THINKING;
45      printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEFT + 1, phnum + 1);
46      printf("Philosopher %d is thinking\n", phnum + 1);
47      test(LEFT);
48      test(RIGHT);
49      sem_post(&mutex);
50  }
```

Figure 4 C Code for the given problem statement continued

Function philosopher

```
52  void* philospher(void* num)
53  {
54      while (1) {
55          int* i = num;
56          sleep(1);
57          take_fork(*i);
58          sleep(0);
59          put_fork(*i);
60      }
61  }
```

Figure 5 C Code for the given problem statement continued

Main function

```
63  int main()
64  {
65      int i;
66      pthread_t thread_id[N];
67      sem_init(&mutex, 0, 1);
68      for (i = 0; i < N; i++)
69          sem_init(&S[i], 0, 0);
70      for (i = 0; i < N; i++) {
71          pthread_create(&thread_id[i], NULL,
72                         philospher, &phil[i]);
73          printf("Philosopher %d is thinking\n", i + 1);
74      }
75      for (i = 0; i < N; i++)
76          pthread_join(thread_id[i], NULL);
77  }
```

Figure 6 C Code for the given problem statement continued

Output

To Run this C Program

```
>> gcc -W -Wall filename.c -o filename -pthread
```

Now to run the executable file,

```
>> ./filename
```

Result

```
2 warnings generated.
❯ ./Lab_6
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 1 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 1 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
```

Figure 7 C Program Output for the given problem statement

7. Analysis and Discussions

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

The Dining Philosopher Problem states that K philosophers seated around a circular table with one fork between each pair of philosophers. There is one fork for each philosopher. A philosopher may eat if he can pick up the two forks adjacent to him. One fork may be picked up by any one of its adjacent followers but not both.

It can be implemented with the wait and signal mechanism in semaphores. Although this solution ensures that no two neighbours are eating together, but still possible deadlock, i.e., if

each philosopher hungry and took the forks left, all grades forks = 0, and then every philosopher will take forks right, there will be a deadlock

8. Conclusions

Dining Philosophers Problem is one of the classic issues in the operating systems. Dining Philosophers Problem can be described as follows there are k philosophers sharing a circular table and they eat and think alternatively. There is a bowl of spaghetti for each of the philosophers and k forks. A philosopher needs both their right and left fork to eat. A hungry philosopher may only eat if there are both forks are available, otherwise a philosopher puts down their fork and begin thinking again.

9. Comments

1. Limitations of Experiments and Results

- A deadlock may occur where the system cannot progress anymore due to a formation of a waiting cycle.
- Mutual exclusion ensures that no two neighbouring philosophers can eat at the same time so a philosopher has to wait for its neighbour to complete eating.
- A live lock may occur where some philosopher has no chance to eat anymore from a certain point and thus starves to death.

2. Learning happened

In this lab implementation of dinning philosopher problem in C using POSIX threads and semaphore is learnt and executed.