OPERATING SYSTEMS LAB (COM-312)

Simulating the Solution for Dining Philosophers problem that explores concurrent programming with threads and mutexes, processes and semaphores.

CSE, MODEL INSTITUTE OF ENGINEERING AND TECHNOLOGY

BACHELOR OF ENGINEERING
In
COMPUTER SCIENCE & ENGINEERING

SUBMITTED BY

RAGHAV SHARMA(2021a1r045)
SAYEUM MAHAJAN(2021a1r034)
VAIBHAV SABHERWAL(2021a1r048)
RASSIA BHARTI(2022a1l010)

SUBMITTED TO
Department of Computer Science &
Engineering Model Institute of Engineering and Technology
(Autonomous)

ACKNOWLEDGEMENT

We take this opportunity to express our sincere gratitude to all those who helped us in various capacities in undertaking this project and devising the report.

We are privileged to express our sense of gratitude to our faculty guide Asst. Professor Saurabh Sharma whose unparalleled knowledge, moral fiber and judgment along with his know-how , was an immense support in completing this project.

We are also grateful to Dr. Ashok Kumar, Dean Academics for the brainwave and encouragement given.

We are also entitled to Professor Ankur Gupta, the director of MIET, for their consistent support and motivation which leads us to a better future.

We take this opportunity to thank our friends for their cooperation and compliance.

FACULTY AND MEMBER

Dr. Ankur Gupta - He is the Director at the ModeI Institute of Engineering and Technology, Jammu, India, besides being a Professor in the Department of Computer Science and Engineering. Prior to joining academia, he worked as Technical Team Lead at Hewlett Packard, developing software in the network management and e-Commerce domains. He has 2 patents to his name and 20 patents pending. He obtained his B.E, Hons. He is a senior member of the ACM, senior member IEEE and a life-member of the Computer Society of India. He has received competitive grants over Rs 2 crores from various funding agencies and faculty awards from IBM and EMC.

Prof. Ashok Kumar - He has over 21 years of experience in industry, academics, research and academic administration. He did his Doctorate from IKG Punjab Technical University, Jalandhar; Master of Engineering from Punjab Engineering College, Chandigarh and Bachelor’s degree from NIT, Calicut, Kerala. He is Fellow of Institution of Electronics and Telecommunication Engineers (IETE). His research interest areas are Optical Networks, Electronics Product Design, Wireless

Sensor Networks and Digital Signal Processing.. He is reviewer of many reputed national and international journals. He has also coordinated many national and international conferences.

Mr. Saurabh Sharma - He is working as Assistant Professor in the Department of Computer Science at Model Institute of Engineering and Technology, Jammu and has over 10 years of experience in academics and research. He has done his M. Tech CSE in Web Mining from Maharishi Markandeshwar University, Mullana, Ambala (HR) in the year 2011.His research interest areas are Web Mining, Data Mining, Machine Learning, ANN, Pattern Classification and Object Identification. He has, to his credit, 25 research papers published in journals of national and international repute and he has guided 11 M. Tech Dissertations.

ABSTRACT

The Dining Philosophers problem is a classic case study in the synchronization of concurrent processes and this research describes how to avoid deadlock condition in dining philosophers problem. Dining itself is a situation where five philosophers sit at a circular table with a large bowl of spaghetti in the centre. A fork is placed in between each pair of adjacent philosophers, and as such, each philosopher has one fork to his left and one fork to his right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. Each philosopher can only use the forks on his immediate left and immediate right. The philosophers never speak to each other, which creates a dangerous possibility of deadlock when every philosopher holds a left fork and waits perpetually for a right fork (or vice versa).To resolve this condition semaphore variable is used. It is marked as in a circular waiting state. At first, most people wear concepts simple synchronization is supported by the hardware, such as user or user interrupt routines that may have been implemented by hardware. In 1967, Dijkstra proposed a concept wearer an integer variable to count the number of processes that are active or who are inactive. This type of variable is called semaphore. The mostly semaphore also be used to synchronize the communication between devices in the device. In this journal, semaphore used to solve the problem of synchronizing dining philosophers problem.This paper presents the efficient distributed deadlock avoidance scheme using lock and release method that prevents other thread in

the chain to make race condition.

CONTENTS

INTRODUCTION

Five philosopher dine together at the same table. Each philosopher has their own place at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right fork. Thus two forks will only be available when their two nearest neighbours are thinking, not eating. After an individual philosopher finishes eating, they will put down both forks. The problem is how to design a regimen (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think (an issue of incomplete information).

OBJECTIVES

From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbours).
A philosopher may pick up only one chopstick at a time.
Obviously, she cannot pick up a chopstick that is already in the hand of a neighbour.
When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
When she is finished eating, she puts down both of her chopsticks and starts thinking again.
It is a simple representation of the need to allocate several resources among several processes in a deadlock- and starvation free manner.
One simple solution is to represent each chopstick by a semaphore.
A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores.

TERMINOLOGY

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.

ALGORITHM

```
philosopher int P[5] ;
While ( TRUE )
{
.....!?!?!.....; /*Thinking*/
P ( fork [j] ) ; /*Pick up left fork*/
P ( fork [i+1] mod 5 ) ; /*Pick up right fork */
eat ( ) ;
V ( fork [i] ) ;
}
}
Philosopher 4 ( ) {
While ( TRUE ) {
...../*Thinking*/
P ( fork [0] ) ; /*Pick up right fork*/
P ( fork [4] ) ; /*Pick up left fork*/
eat( ) ;
V ( fork [4] ) ;
V ( fork [0] ) ;
}
}
Semaphore fork [5] = {1, 1, 1, 1, 1};
fork (philosopher), 1, 0) ;
fork (philosopher), 1, 1) ;
fork (philosopher), 1, 2) ;
fork (philosopher), 1, 3) ;
fork (philosopher), 4, 0) ;
```

METHODOLOGY (FLOWCHART)

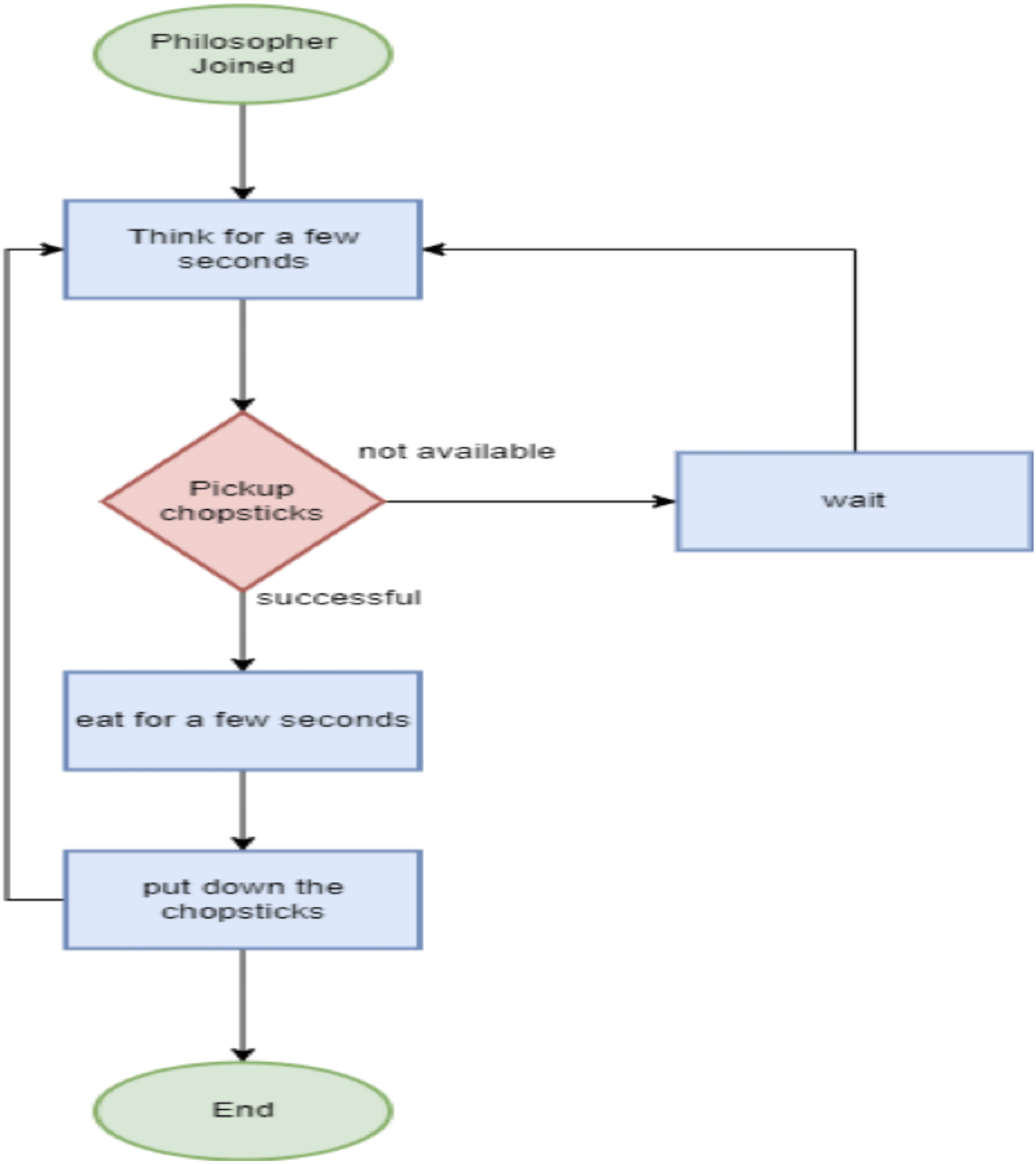

Fig 1.

IMPLEMENTATION

A) CODE:

```
#include<stdio.h>
#include<semaphore.h>
#include<pthread.h>
#include<unistd.h>
#define N 5 //Number of philosopher are 5
#define THINKING 0 //Three States Thinking, Hungry and Eating
#define HUNGRY 1
#define EATING 2
#define LEFT (ph_num+4)%N //Two conditions for picking the
fork/chopstick
#define RIGHT (ph_num+1)%N
sem_t mutex;
sem_t S[N];
int count[5];
int FOOD = 0;
void * philospher(void *num);
void take_fork(int);
void put_fork(int);
void test(int);
int state[N]; //Checks the state of the philosopher
int phil_num[N]={0,1,2,3,4}; //Sequence for the philosopher's
int main() //main function
{
int i;
pthread_t thread_id[N]; //declaration of threads
sem_init(&mutex,0,1); //Use of semaphores(binary)
for(i=0;i<N;i++)
sem_init(&S[i],0,0);
for(i=0;i<N;i++) //Creation of threads for all philosophers
```

```
{
pthread_create(&thread_id[i],NULL,philospher,&phil_num[i]);
}
for(i=0;i<N;i++)
pthread_join(thread_id[i],NULL); // waits for the thread to exit
for(i=0;i<N;i++)
printf("Philospher %d ate %d \n",i,count[i]);
// outputs food count for each philosophers
printf("\n");
}
void *philospher(void *num)
{
while(FOOD <= 20) //use of while condition
{
int *i = num; //picking up and picking down fork condition
usleep(10000);
take_fork(*i);
put_fork(*i);
}
}
void take_fork(int ph_num) //hungry state condition
{
sem_wait(&mutex);
state[ph_num] = HUNGRY;
test(ph_num);
sem_post(&mutex);
sem_wait(&S[ph_num]);
usleep(10000);
}
void test(int ph_num)
{
if (state[ph_num] == HUNGRY && state[LEFT] != EATING &&
state[RIGHT] !=
EATING)
{ //eating state condition
```

```
state[ph_num] = EATING; //condition for checking the availability of
forks(right&left)
usleep(20000);
sem_post(&S[ph_num]);
}
}
void put_fork(int ph_num) //thinking state condition
{
sem_wait(&mutex);
state[ph_num] = THINKING;
count[ph_num]++;
FOOD++;
test(LEFT);
test(RIGHT);
printf("#Eating Count = %d \n", FOOD);
int i;
for(i=0;i<5;i++){ //setting the states for the philospher
if(state[i]==EATING)
printf("Philosopher %d is eating\n", i);
else if(state[i]==HUNGRY)
printf("Philosopher %d is waiting and calling pickup()\n", i);
else if(state[i]==THINKING)
printf("Philosopher %d is thinking\n", i);
}
sem_post(&mutex);
```

B) **OUTPUTS (TEST):**

```
smarty@Raghavs-MacBook-Air ~ % ./a.out
#Eating Count = 1
Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 2 is waiting and calling pickup()
Philosopher 3 is eating
Philosopher 4 is waiting and calling pickup()
#Eating Count = 2
Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 2 is thinking
Philosopher 3 is eating
Philosopher 4 is thinking
#Eating Count = 3
Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 2 is thinking
Philosopher 3 is eating
Philosopher 4 is thinking
#Eating Count = 5
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is waiting and calling pickup()
Philosopher 3 is eating
Philosopher 4 is waiting and calling pickup()
#Eating Count = 7
Philosopher 0 is thinking
```

Fig 2.

REFERENCES

1. **JAVA POINT**

   https://www.javatpoint.com/os-dining-philosophers-problem

2. **GEEKS FOR GEEKS**

   https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/

3. **WIKIPEDIA**
   https://en.wikipedia.org/wiki/Dining_philosophers_problem#:~:text=The%20problem%20is%20how%20to,an%20issue%20of%20incomplete%20information).

4. **TUTORIALS POINT**

   https://www.tutorialspoint.com/dining-philosophers-problem-dpp

5. **RESEARCH GATE**

   https://www.researchgate.net/publication/221644832_The_Dining_Philosophers_Problem_Revisited