International Journal of Advances in Engineering and Management (IJAEM)
Volume 2, Issue 9, pp: 381-385www.ijaem.net ISSN: 2395-5252

Dining Philosophers Theory and Concept in Operating System Scheduling

Phavya.J,Pratheeksha.K.S, Tissyashri.S, Dr.M.SujithraM.C.A M.Phil., Phd.,Dr.A.D.Chitra M.C.A M.Phil., Phd.

2nd Year, M.Sc. Software Systems (Integrated),Coimbatore Institute of Technology,Coimbatore
Assistant Professor, Department of Data Science Coimbatore Institute of Technology, Coimbatore
Assistant Professor, Department of Software Systems, Coimbatore Institute of Technology, Coimbatore

Date of Submission: 10-11-2020 Date of Acceptance: 25-11-2020

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 center. 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.

KEYWORDS-Dining Philosophers Problem, Race Condition, Concurrent**,** Deadlock, Starvation

I. INTRODUCTION

An Operating System (OS) is an interface between a computer user and computer hardware. In the process of designing the operating system, there is a common base called concurrency. Concurrent processes are when the processes work at the same time. This is called the multitasking operating system. Concurrent processes can be completely independent of the other but can also interact with each other. the concurrent processes that interact, there are some problems to be solved such as deadlock and synchronization.

The illustration of dining philosopher problem is as follows:

The five philosophers sit at a circular table with a large bowl of spaghetti in the center. 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). In between there may be the possibility of deadlock, which occurs due to starvation. Deadlock is the condition in which two or more processes cannot continue execution at the same time.

II. METHODOLOGY

The philosophers are sitting around a round table, and there is a big bowl of spaghetti at the center ofthe table. There are five forks placed around the table in between the philosophers. When a philosopher, who ismostly in the thinking business gets hungry, he grabs the two forks to his immediate left and right and dead-seton getting a meal; he gorges on the spaghetti with them. Once he is full, the forks are placed back, and he goesinto his mental world again. The problem usually omits an important fact that a philosopher never talks toanother philosopher. The typically projected scenario is that if all the philosophers grab their fork on their leftsimultaneously none of them will be able to grab the fork on their right. Moreover, with their one-track mindset,they will forever er keep waiting for the fork on their right to come back on the table.

Assume that we have the simple task of writing some important information into two files on the disk.However, these files are shared by other programs as well. Therefore we use the following strategy to updatethe files:

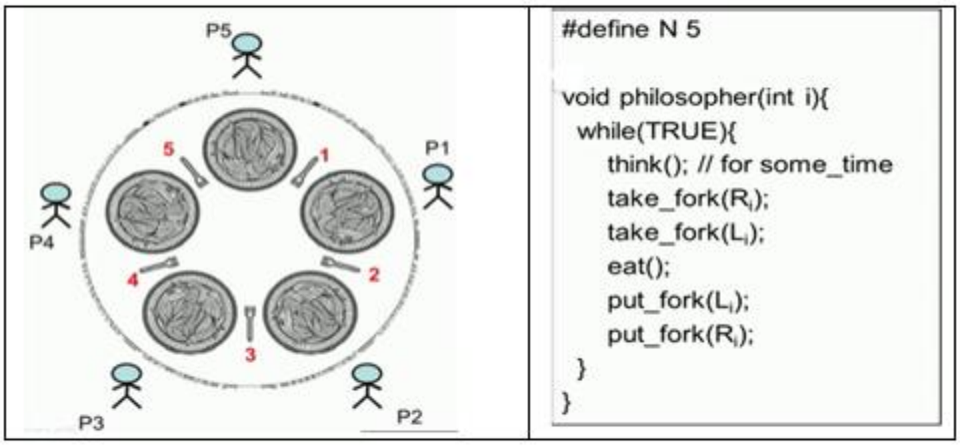
Lock A
Lock B

Write information to A and B
Release the locks

This obvious coding can result in deadlocks if other tasks are also writing to these files. For example, ifanother task locks B first, then locks A, and if both tasks try to do their job at the same time – dead-lock occurs.My task would lock A, the other task would lock B, then my task would wait indefinitely to lock B while theother task waits indefinitely to lock A. This is a simple scenario, and easy to find out. However, you can have abit more involved case where task A can wait for a lock held by task B which is waiting for a lock held by taskC which is waiting for a lock held by task A. A circular wait a deadlock results. This is a Dining Philosophersmodel.

III. IMPLEMENTATION

Dining Philosophers Problem is one of the classic problems in the synchronization. Dining Philosophers problem can be illustrated as follows;we have five philosophers P 1, P 2, P 3, P 4, P 5 who are sitting around a table. One for each of the philosophers and five forks 1, 2, 3, 4, 5.Now each of these philosophers could do just one of two things. Each philosopher could either think or eat. Now, in order to eat, a philosopher needs to hold both forks. If P1 wants to eat then he needs to have the fork 1 and fork 2. Similarly, if P3 wants to eat then forks 4 and forks 3 are required. Now the problem is or the problem what we are trying to solve is to develop an algorithm, where no philosopher starves that is every philosopher should eventually get a chance to eat.



```
#define N 5

void philosopher(int i){
  while(TRUE){
     think(); // for some_time
     take_fork(R_i);
     take_fork(L_i);
     eat();
     put_fork(L_i);
     put_fork(R_i);
  }
}
```

Fig 1. First Try Algorithm

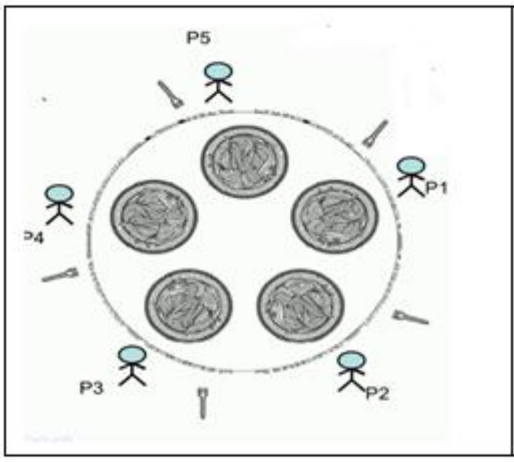
- **First try:** Let us say we have a solution over here, where we define N as 5 corresponding to each philosopher. And we have a function for philosopher. So, this function takes a integer value ‘i' and this ‘i' could be values of 1 to 5 corresponding to each philosopher that is P 1, P 2, P 3, P 4 or P 5.Philosopher will think for some time and then after some time he begins to feel hungry. So, he will take the fork on his right and then left, then he is going to eat for some time, and after that he is going to put down the left fork, and then right fork, and this continues in a loop infinitely. So, for instance philosopher P 1 will think for some time, then feel hungry, then he would pick up the fork 1, fork 2, then eat for some time and put down both the forks.So, this seems like a very simple solution but it cause issues.
- P1 and P3 have higher priority. So P1 and P3 eats whenever they wants, while others have

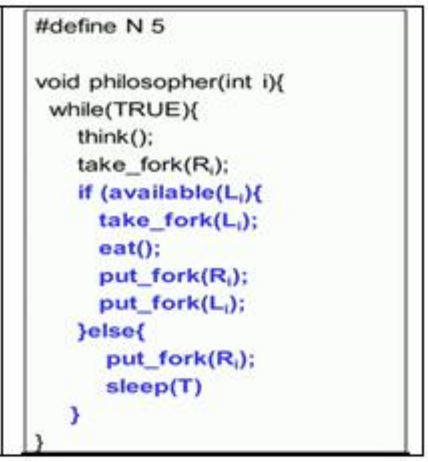
less priority. For instance, P2 neither could pick up right or left fork and therefore, P2 cannot eat. Same for P4 and P5 follows. Thus P 2, P 4 and P 5 will starve; and this is not the ideal solution for our problem.

- All philosophers pickup their right simultaneously. Now, in order to eat each philosophers have to pick up their left fork and this could lead to starvation. So, each of the philosophers is waiting for nearer philosopher to put down the fork. So, there is a circular wait and this leads to a deadlock, there by starvation.

❖ **Second try:** We have the same function over here. The philosopher takes the right fork, then he would determine if the leftfork is available; if the left fork is available the philosopher would take the left fork,eat for sometime then put down both the forks and theloop continues as usual.However, if the left fork is not available, then we go to the else part and the philosopherwill put back the right fork.So this will allow another philosopher to probably eat.And after this is done there is a sleep for some fixed interval T before the philosophertries again. Let us see what issue may cause.

- Let us consider where all philosophers start at exactly the same time, they run simultaneously and think for exactly the same time. Here all philosophers find out that their forks are not available, so they put down their fork simultaneously, then they sleep for some time and then they repeat the process. A slightly better solution to this case is where instead of sleeping for a fixed time,the philosopher would put down the right fork and sleep for some random amount of time.While this does not guarantee that starvation will not occur, it reduces the possibilityof starvation.



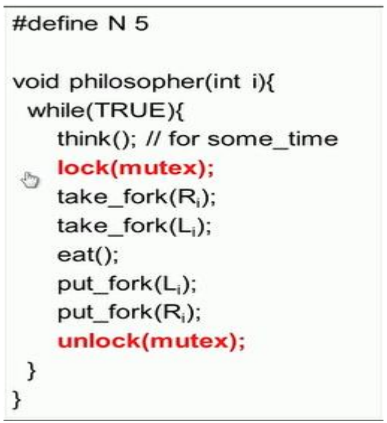


```
#define N 5

void philosopher(int i){
  while(TRUE){
     think();
     take_fork(R_i);
     if (available(L_i)){
       take_fork(L_i);
       eat();
       put_fork(R_i);
       put_fork(L_i);
     }else{
        put_fork(R_i);
        sleep(T)
     }
}
```

Fig 2. Second Try Algorithm

❖ **Third try using Mutex:** This particular solution uses a mutex.Essentially before taking the right or the left fork, the philosopher needs to lock amutex.And the mutex is unlocked only after eating and the forks are put back on to the table. So, this solution essentially ensures that starvation will not occur, it prevents deadlocks.However, the problem here is that because we are using a mutex, so at most one philosophercan enter into this critical section.In other words, at most one philosopher could eat at any particular instant.So, while this solution works, it is not the most efficient solution.So, we would want something which does much better than this.



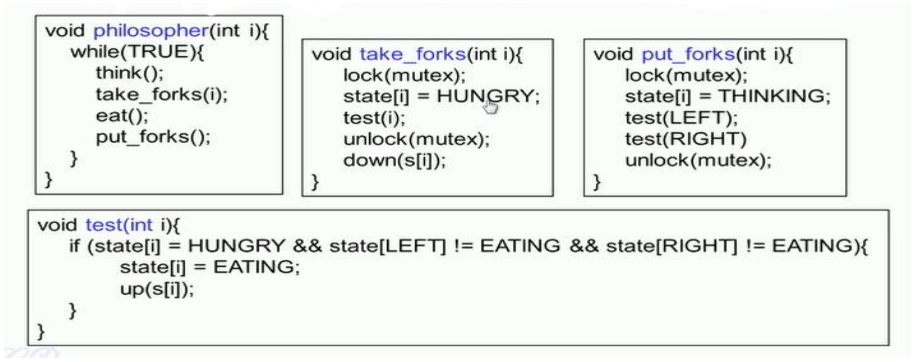
```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      lock(mutex);
      take_fork(R_i);
      take_fork(L_i);
      eat();
      put_fork(L_i);
      put_fork(R_i);
      unlock(mutex);
  }
}
```

Fig 3. Using Mutex

❖ **Forth try using semaphores:** This particular solution uses a semaphores. So let us say that we have N semaphores; so the semaphores s[1] to s[n] and we have onesemaphore per philosopher.So all these semaphores are initialized to 0; in addition the philosopher can be in oneof 3 states – hungry, eating or thinking.For instance when the philosopher is thinking, the state will be thinking; then the philosopherbecomes hungry, so it goes to the hungry state then eating, and then back to thinking, andthis process continuous till the eternity.So, the general solution that we will be seeing here is that a philosopher can only move to the eating state if neither neighbor is eating.That is a philosopher can eat only if its left neighbor as well as its right neighboris not eating.So, in order to implement this particular solution, we have four functions. It has four functions.

- First is the philosophers, which is the infinite loop and corresponds to the philosopher 'i'. The philosopher will think, then it will take forks, then eat for some time and put down the forks and this repeats continuously.
- Now in the take fork function, first we set that the philosopher in a hungry state. The state of the philosopher is set to hungry then the function called test is invoked. So, what test will do is that it's going to check whether the state of the philosopher is hungry and as well as the state of the philosopher to the left as well as to the right is not in the eating state. If this indeed is true then the philosopher can eat. And at the end, after eating, the forks are put down and the state of the philosopher goes to thinking.



```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
      lock(mutex);
      state[i] = HUNGRY;
      test(i);
      unlock(mutex);
      down(s[i]);
}
```

```
void put_forks(int i){
      lock(mutex);
      state[i] = THINKING;
      test(LEFT);
      test(RIGHT)
      unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

Fig 4. Using Semaphore Algorithm

	Time A	Time B	State
Philosopher-1	7	10	15
Philosopher-2	5	4	4
Philosopher-3	6	11	14
Philosopher-4	5	13	11
Philosopher-5	6	12	18

Table 1 Normal Philosopher Properties

From Table 1, it can be seen the conditions of each philosopher, the philosopher-1 are in a state of satiety as initial conditions are above the 15-seconds and that only 10 seconds. The philosopher-2 in a state of hunger because of the initial conditions = 4 seconds of the time-B, philosopher-3 are in a condition to be satisfied, the philosopher-4 in a state of hunger and philosopher-5 in a state of satiety.The initial condition dining philosophers problem can be illustrated by the following illustration: At the time t = 1 second, the philosopher-1, 3-philosophers and philosopher-5 full and thinking, while philosophers and philosopher-2-4 hungry and get the chopsticks in his left hand. At time t = 2 seconds, philosophers and philosopher-2-4 got two chopsticks and began eating, while the philosopher-1, 3-philosophers, and philosopher-5 are still satisfied and thinking. At time t = 3 second, the philosopher-3 was hungry (for the lifetime of the philosopher-3 now = time-B is 11 second) and started looking for chopsticks, but did not get chopsticks for chopsticks on the left is used by the philosopher-4 and chopsticks on the right is used by the philosopher-2. At time t = 5 seconds, the philosopher-1 was hungry (for the lifetime of the current philosopher-1-B = time of10 seconds) and look for chopsticks. Philosopher-1 to get the chopsticks in the right hand. At time t = 6 second, philosopher-5 was hungry (for the lifetime of the current philosopher-5-B = time is 12 seconds) and look for chopsticks. The philosopher-5 did not get chopsticks. At the time t = 9 seconds, philosopher-2 full (because his life has reached its maximum value, which is 9-second = time A + time-B) and start thinking. The philosopher3 to get the chopsticks in the right hand. At time t = 10 second, philosopher-1 got two chopsticks and began eating. At time t = 11 second, philosopher-4 satiety and start thinking. The philosopher-5 to get the chopsticks in the right hand. At time t = 12 second, philosopher-3 got two chopsticks and began eating.

The simulation process will continue by the procedure. The simulation will only stop if there is a deadlock condition. A deadlock condition in the simulation dining philosophers problem occurs when at one time; all the philosophers get hungry simultaneously, and all philosophers take the chopsticks in his left hand. By the time the philosopher will take the chopsticks in the right hand, then there was a deadlock condition since all philosophers will both waiting for chopsticks on the right (a condition that will never happen). For the case of deadlock, consider the following:

	Time A	Time B	State
Philosopher-1	17	12	27
Philosopher-2	5	3	2
Philosopher-3	15	10	25
Philosopher-4	6	5	5
Philosopher-5	20	5	20

Table 2 Deadlock Philosopher Properties

From Table 2, at the time t = 1 second, philosophers and philosopher-2-4 hungry and get the chopsticks in his left hand, while the philosopher-1, philosopher-3 and philosopher-5 full and thinking. At time t = 2 seconds, philosophers and philosopher-2-4 got two chopsticks and began eating. At time t = 10 second, philosophers and philosopher-2-4 satiety and start thinking. At time t = 15 seconds, all philosophers simultaneously hungry and took the chopsticks in his left hand. At this time, there has been a deadlock condition, because all the philosophers who were holding the chopsticks in hand chopsticks left waiting on the right. All philosophers will wait for each other.

IV. CONCLUSION

Dining Philosophers Problem is one of the classic issues in the operating systems. Dining Philosophers Problem can be described as follows; there are five philosophers who want to eat. There are five chopsticks on the table. Each philosopher must use two chopsticks if he would like to eat the spaghetti. If philosophers really hungry, then it will take two chopsticks, which is in the right and left hands. If there are philosophers who took two chopsticks, then there are philosophers who have to wait until the chopsticks are placed back. Inside this problem there is the possibility of deadlock.

REFERENCES

[1]. R. Alur, H. Attiya and G. Taubenfeld, "Time-Adaptive Algorithms for Synchronization," in Proc. 26th ACM Symposium on Theory of Computing, Canada, 1994.
[2]. J. G. Vaughan, "The Dining Philosophers Problem and Its Decentralisation," Microprocessing and Microprogramming, vol. 35, no. 1, pp. 455-462, 1992.
[3]. Dijkstra, E. W.: Hierarchical Ordering of Sequential Processes, Acts Informatica I, 115 – 138 (1971)
[4]. Brinch Hansen, P. Operating Systems Principles. Prentice-Hall 1973.
[5]. Hoare, C.A.R. Towards a theory of parallel programming, Operating Systems Techniques, quoted above.