IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 18, Issue 6, Ver. IV (Nov.-Dec. 2016), PP 45-50
www.iosrjournals.org

Dining Philosophers Theory and Concept in Operating System Scheduling

Zuhri Ramadhan[1], Andysah Putera Utama Siahaan[2]
Faculty of Computer Science
Universitas Pembangunan Panca Budi
Jl. Jend. Gatot Subroto Km. 4,5 Sei Sikambing, 20122, Medan, Sumatera Utara, Indonesia

***Abstract**: This research describes how to avoid deadlock condition in dining philosophers problem. It is the undesirable condition of concurrent systems. 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. Dining itself is a situation where five philosophers are sitting at the dinner table to eat spaghetti, every philosopher is given a plate of spaghetti and one chopstick to eat spaghetti the two chopsticks are needed to resolve the issue semaphore variable is then applied to each chopstick chopsticks that can be shared all the other philosopher. 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

The operating system is a program that links the user and the computer system. This operating system must be capable of controlling resource usage. In the process of designing the operating system, there is a common foundation 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 [1].

Processes that require synchronization to interact properly controlled. However, the concurrent processes that interact, there are some problems to be solved such as deadlock and synchronization. One of the classic problems that can illustrate the problem is the Dining Philosophers Problem.Dining Philosophers Problem can be illustrated as follows; there are five philosophers who would eat. On the table was reserved five chopsticks. If philosophers really hungry, then it will take two chopsticks, which is in the right and left hands. However, sometimes only one course takes chopsticks. 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, a condition in which two or more processes can not continue execution [3][7].

II. Theories

In literature, there are many different operating systems interesting issues that have been discussed and analyzed. Here is a classic example of the problems in the field of inter-process communication that should be resolved by a system operate.salah one example of modeling processes that are competing for exclusive access to a limited number of resources, such as input / output device. In 1965, Djikstra completing a synchronization problem that he calls the dining philosophers problem. Dining Philosophers Problem is one of the classic problems in the synchronization [4][5].

Dining Philosophers problem can be illustrated as follows; there are five philosophers sitting around a table. There are five bowls of noodles in front of each philosopher and one chopstick in between each philosopher. The philosophers spend time thinking (when full) and eating (when hungry). When hungry, the philosopher will take two chopsticks (in the left hand and right hand) and eat. However, sometimes, just taken one chopstick alone. If there are philosophers who took two chopsticks, then the next two philosophers philosopher who was eating must wait until chopsticks put back. It can be implemented with the wait and signal.Figure 1 show the problem of dining philosophers [6].

Fig. 1 : The Dining Philosophers Problem

We can modify the program so that after taking chopsticks left, the program checks whether the right chopstick allows being taken. If the right chopstick is not likely to be taken, the philosopher left chopsticks laid back, waiting for some time, later repeating the same process. The proposal also wrong, even if for different reasons. With a bit of bad luck, all the philosophers can start algorithms simultaneously, take chopsticks their left, look right chopsticks they were not likely to be taken, laid back their left chopsticks, wait, take chopsticks simultaneously left them again, and so on. Situations like this where all the programs continue to run in unlimited but no changes/advancements that called starvation.

Although this solution ensures that no two neighbors are eating together, but still possible deadlock, i.e., if each philosopher hungry and took the chopsticks left, all grades chopsticks = 0, and then every philosopher will take chopsticks right, there will be a deadlock. There are several ways to avoid deadlock, among others:

- Allows at most four philosophers who sit together at one table.
- Allows a philosopher take chopsticks only if both chopsticks were there.
- Using asymmetric solutions, namely philosophers in odd numbers took a left chopstick first and right chopsticks. While sitting on a chair philosopher even take the right chopstick first, and chopsticks left.

The process is called deadlock if the process of waiting for a particular event that will never happen. A set of processes unconditioned deadlock when every process that is in the collection of waiting for an event that can only be done other processes are also in the collection. The process of waiting for events that will never happen. Deadlock occurs when processes to access exclusive resources. All the deadlock that occurred involving an exclusive competition to secure resources by two or more processes.

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. The process is said to be experiencing starvation when the processes are waiting for the allocation of resources to infinity, while the other processes can obtain resource allocation. Starvation caused bias in policy or strategy of resource allocation. This condition should be avoided because it is unfair, but the desired avoidance is done as efficiently as possible.

Before start taking chopsticks, a philosopher did DOWN on the mutex. After replacing chopsticks, he had to do a UP in the mutex. Regarding theory, this solution is sufficient. However, regarding practice, this solution has fixed the problem. There is only one philosopher can eat the noodles in a variety of occasions. With five chopsticks, we should be able to watch two philosophers eating noodles at the same time.Solutions provided above right and also to allow the maximum number of parallel activities for some philosophers changing uses an array, state, to record the status of a philosopher whether eating, thinking or being hungry because trying to take chopsticks. A philosopher can only status eating (eating) if no neighbor was eating well. Neighbors philosopher defined ole LEFT and RIGHT.

In other words, if i = 2, then the neighbor left (LEFT) = 1 and the neighboring right (RIGHT) = 3. This program uses an array of semaphore hungry (hungry) can be detained if chopsticks left or right of it being worn neighbors. Note that each process is running philosophers procedures as the main code, but other procedures such as take-forks, and the test is the usual procedure and not separate processes. One possible solution is to use a directly visible semaphore. Each chopstick represents a semaphore. Then, when a hungry philosopher, then he

will try to take the chopsticks on the left and the right, or in other words, he will wait until both chopsticks can show he used. When finished eating, chopsticks laid back, and the signal is given to the semaphore so that other philosophers who need can use chopsticks.

Semaphore is a computer data structure that is used for the synchronization process, which is to solve the problem where more than one process or thread run concurrently and must be arranged the sequence of works. Semaphore coined by Edsger Dijkstra and first used in operating systems. Semaphore value is initialized with the number of resource control. In the special case where there is a shared resource called "binary semaphore." Semaphore is a classic solution to the dining philosophers' problem, although it does not prevent deadlock.

III. Methodology

The philosophers are sitting around a round table, and there is a big bowl of spaghetti at the center of the table. There are five forks placed around the table in between the philosophers. When a philosopher, who is mostly in the thinking business gets hungry, he grabs the two forks to his immediate left and right and dead-set on getting a meal; he gorges on the spaghetti with them. Once he is full, the forks are placed back, and he goes into his mental world again. The problem usually omits an important fact that a philosopher never talks to another philosopher. The typically projected scenario is that if all the philosophers grab their fork on their left simultaneously 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.

The basic idea behind the scenario is that if a concurrent activity always does what seems best for itself or what seems to be the right thing for itself in a shared resources scenario, the result can be chaos. Is there a solution to the Dining Philosopher Problem? The scenario was posed not for a solution but to illustrate a basic problem if the traditional programming approach is applied to concurrent systems. The problem itself crops up in the concurrent systems, and the design decisions should be aware of this, and that is what we have to solve. Any set of concurrent programming techniques that we use is expected at the basic level to offer us features that can be used to deal with the Dining Philosophers problem in some way.

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 update the 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, if another 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 the other task waits indefinitely to lock A. This is a simple scenario, and easy to find out. However, you can have a bit more involved case where task A can wait for a lock held by task B which is waiting for a lock held by task C which is waiting for a lock held by task A. A circular wait a deadlock results. This is a Dining Philosophers model.

In the above code fragment, one could resort to locking the files one at a time for modification. Then the problem would disappear. However, there are times when requirements dictate that it has to be locked more than one resource before updating them.

IV. Implementation

Dining Philosophers Problem is one of the classic problems in the synchronization. Dining Philosophers problem can be illustrated as follows; there are five philosophers sitting around a table. There are five bowls of noodles in front of each philosopher and one chopstick in between each philosopher. The philosophers spend time thinking and eating. When hungry, the philosopher will take two chopsticks and eat. However, sometimes, just taken one chopstick alone. If there are philosophers who took two chopsticks, then the next two philosophers philosopher who was eating must wait until chopsticks put back. It can be implemented with the wait and signal. Although this solution ensures that no two neighbors are eating together, but still possible deadlock, i.e., if each philosopher hungry and took the chopsticks left, all grades chopsticks = 0, and then every philosopher will take chopsticks right, there will be a deadlock [2]. Figure 2 shows the diagram of the dining philosophers problem.

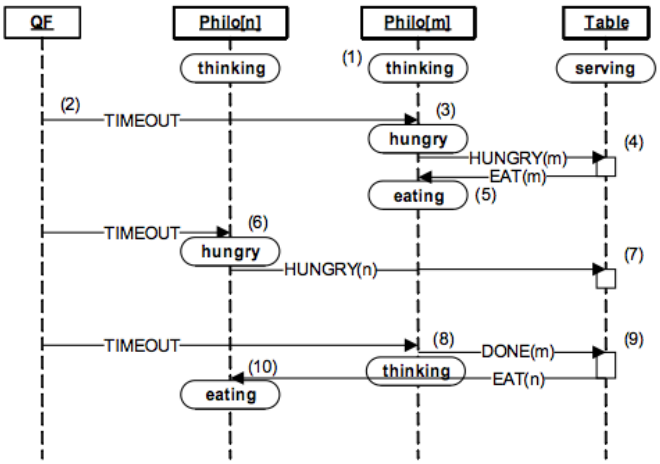

Fig. 2 The sequence diagram of the DPP application

- Each Philosopher active object starts in the "thinking" state. Upon the entry to this state, the Philosopher arms a one-shot time event to terminate the thinking.
- The QF framework posts the time event (timer) to Philosopher[m].
- Upon receiving the TIMEOUT event, Philosopher[m] transitions to "hungry" state and posts the HUNGRY(m) event to the Table active object. The parameter of the event tells the Table which Philosopher is getting hungry.
- The Table active object finds out that the forks for Philosopher[m] are available and grants it the permission to eat by publishing the EAT(m) event.
- The permission to eat triggers the transition to "eating" in Philosopher[m]. Also, upon the entry to "eating", the Philosopher arms its one-shot time event to terminate the eating.
- The Philosopher[n] receives the TIMEOUT event, and behaves exactly as Philosopher[m], that is, transitions to "hungry" and posts HUNGRY(n) event to the Table active object.
- This time, the Table active object finds out that the forks for Philosopher[n] are not available, and so it does not grant the permission to eat. Philosopher[n] remains in the "hungry" state.
- The QF framework delivers the timeout for terminating the eating arrives to Philosopher[m]. Upon the exit from "eating", Philosopher[m] publishes event DONE(m), to inform the application that it is no longer eating.
- The Table active object accounts for free forks and checks whether any direct neighbors of Philosopher[m] are hungry. Table posts event EAT(n) to Philosopher[n].
- The permission to eat triggers the transition to "eating" in Philosopher[n].

Table 1 Normal Philosopher Properties

	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

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 of

10 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 philosopher-3 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.

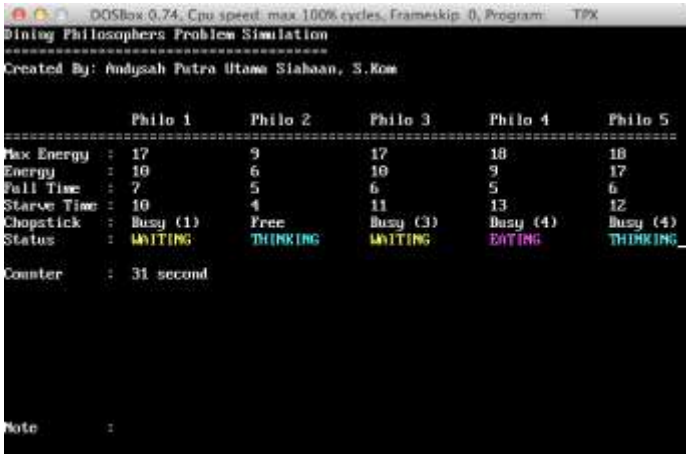

Fig. 5 Normal Running Program

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:

Table 2 Deadlock Philosopher Properties

	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

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.

The deadlock condition that occurs can be avoided by these three following solutions:

1. There are four philosophers who sit together at one table. Deadlock condition will not occur if there are less than four philosophers who sit together around the table with five seats.
2. A philosopher take chopsticks only if both chopsticks were there. If all philosophers hunger simultaneously, then only two philosophers who can eat, because philosophers took two chopsticks at the same time.
3. Philosopher on odd numbered take the first left new chopsticks right, while the even-numbered philosophers take the right chopstick first and chopsticks left. If all philosophers hunger simultaneously, making way for asymmetric solution will prevent all philosophers take the left chopstick simultaneously so that a deadlock condition can be avoided.

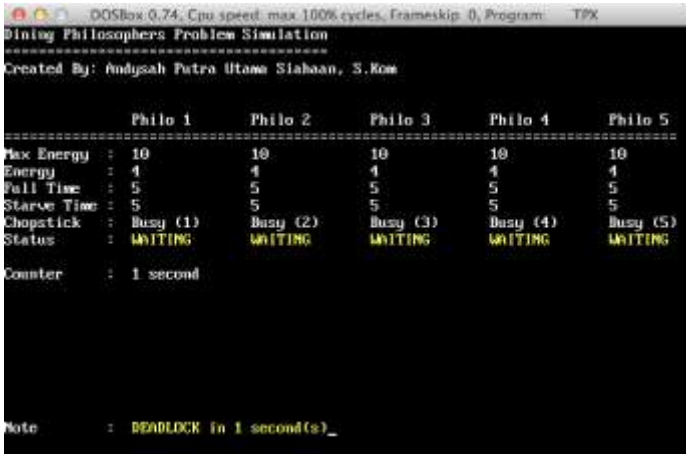

Fig. 6 Deadlock Running Program

Figure 5 show the program running well. There are various states in there. Two of the philosophers are waiting, two are thinking while the another one is still eating. There is no deadlock condition in here. After running the second program, it demonstrated the wrong algorithm. Figure 6 shows the five philosophers are all waiting. No way out has been found. It

V. 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]. A. P. U. Siahaan, "Penyelarasan Pada Masalah Dining Philosophers Menggunakan Algoritma Lock & Release," Techsi, vol. 6, no. 1, pp. 14-18, 2015.
[2]. M. Samek, Application Note Dining Philosophers Problem (DPP) Example, USA: Quantum Leaps, 2012.
[3]. V. R. Raojillelamudi, S. Mukherjee, R. S. Ray and U. K. Ray, "Lock-Free Dining Philosopher," International Journal of Computer & Communication Technology, vol. 4, no. 3, pp. 54-58, 2013.
[4]. E. Styer and G. Peterson, "Improved Algorithms for Distributed Resource Allocation," in Proc. 7th ACM Symposium on Principles of Distributed Computing, Toronto, Canada, 1988.
[5]. R. Alur, H. Attiya and G. Taubenfeld, "Time-Adaptive Algorithms for Synchronization," in Proc. 26th ACM Symposium on Theory of Computing, Canada, 1994.
[6]. T. A. S. K. Ishwarya and R. C. A. Naidu, "A New Methodology to Avoid Deadlock with Dining Philospher Problem in Rust and Go System Programming Languages," International Journal of Advanced Research in Computer and Communication Engineering, vol. 4, no. 9, pp. 418-420, 2015.
[7]. J. G. Vaughan, "The Dining Philosophers Problem and Its Decentralisation," Microprocessing and Microprogramming, vol. 35, no. 1, pp. 455-462, 1992.