COMP306 - Dining Philosophers Problem

Eren TÜRKAY
Department of Computer Science,
Istanbul Bilgi University,
Istanbul,
Turkey
turkay.eren@gmail.com

May 17, 2013

1 Introduction

Dining philosophers is an example problem used to describe the problems, and techniques for solving these problems in a concurrent programming environment. The problem was originally formulated in 1965 by Edsger W. Dijsktra. It was proposed as an example to illustrate concurrency and synchronization problems for his students.

The problem is stated with n number of philosophers sitting around the dinner table. On the table, there are n number of forks and n number of tables. Usually, the problem is stated with n=*5*. Accordingly to Siblerschatz, the problem is defined as follows:

> Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. 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 neighbors). 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 neighbor. 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. [1]

In Dining Philosophers problem, chopsticks represent the shared resource, and the philosophers represent the processes that want to access these resources. The problem is important because it allows us to intuitively understand concurrency problems.

2 Main Problem

The main problem in *Dining Philosophers* is either that they starve to death or that they hold the stick at the same time so they cannot think or eat.

Imagine that each philosopher sits on the table at the same time and holds one of the sticks. In this case, all chopsticks are occupied and philosophers wait each other for another stick. No philosopher can eat or think. Hence, a deadlock occurs.

When two philosophers are fast thinkers and eaters, they can occupy 4 chopsticks so fast that other philosophers have no chance of getting chopsticks. This leads to ”starvation”. It is different from a deadlock in that 2 philosophers think&eat, others starve.

3 Techniques For Solving

There are different ways of solving Dining Philosophers Problem. In this section, possible solutions to the problem are given. In each section, the way, and the problems that the solution posses are explained.

3.1 Semaphores

Using semaphore is one of the simple solutions to the problem. Each chopstick is represented with a semaphore.

A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1. So, the algorithm when using semaphores is:

1. Lock left chopstick
2. Lock right chopstick
3. Eat
4. Release left chopstick
5. Release right chopstick
6. Think

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. [1]

There are a few ways for encountering the deadlock problem presented by using semaphores:

1. Allow at most four philosophers to be sitting simultaneously at the table

2. Allow a philosopher to pick up her chopstick only if both chopsticks are available. (to do this, she must pick them up in a critical section)

3. Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

3.2 Monitors

This method of solving *Dining Philosophers Problem* arises from the fact that using semaphores is prone to programming errors. When a programmer incorrectly use semaphores, deadlock or starvation could easily occur. With monitors, synchronization and concurrency are achieved through more abstract classes which are not error prone.

A monitor is an *ADT - Abstract Data Type* which encapsulates private data with public methods to operate on that data. A monitor type is an ADT that presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly. This model of monitoring is present in most of the Object Oriented Languages including *Java, Smalltalk, C++, etc.*

The solution to *Dining Philosophers Problem* using monitors imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. For this purpose, a new data structure is created:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher i can set the variable `state[i] = EATING` only if her two neighbors are not eating: `(state[(i+4) % 5] != EATING) AND (state[(i+1) % 5] != EATING)`

4 My Solution

My solution involves using semaphores, and checking if both chopsticks are available to avoid deadlock. The code is self-documented, which can be easily understood.

4.1 Downside

Currently, the code does not have a deadlock as a thread starts eating only if both chopsticks are available. However, the code is prone to *starvation*. If two threads are fast thinkers and fast eaters, other threads will not be able to acquire a chopstick, and resources will be only available to fast thinking&eating threads.

There are ways to overcome this problem. One way is to count the number of eating time and compare this time with other threads. If one thread is eating too many times, it is not given an opportunity to get chopsticks. Chopsticks are given to other threads. The program controls this eating time and balance it between threads.

References

[1] Silberschatz, Abraham (2011) Operating system concepts essentials. *Process Synchronization* (p. 234). Hoboken, NJ: John Wiley & Sons Inc.