CMPE 312 FINAL PROJECT

Simge Erek

The Dining Philosophers' Problem

May 10, 2019

A dining philosophers problem is a classic synchronization problem. It is evaluate situations where there is a need of allocating multiple resources to multiple processes and provides a significant learning value , particularly in process synchronization.It was orginally formulated in 1965 by Edger Dijkstra. Presented as a student exam exercise, the problem illustrates a number of computers competing for Access to tape drive peripherals.The formulation know today was a later version by Tony Hoare.

There are 5 philosophers who sit at the circular table and each has his/her own chair.There is a rice in the center of the table and one chopstick on left and right to each philosophers.Philosophers spend their time just thinking or eating, they never speak to each other.They cannot eat with one chopstick, they need two chopstick to eat.Each philosopher thinks until the becomes hungry.When philosopher is hungry, he/she gets two chopsticks on his/her left and his/her right sides and eats for a while, then he/she puts the chopstick down and starts thinking again.

In the code, there is a five philosopher who are numbered 0 and 4.Each philosopher is represented by a thread that executes the function philosopher(i), where i is the number of that philosopher.A philosopher can be in one of the three state; HUNGRY,EATING or THINKING.If philosopher is hungry, then he/she will be waiting for a chopstick.If philosopher is eating, he/she has two chopstick and is eating.

Semaphore s[i] is assigned to philosopher i. The role of this semaphore is to enable philosopher i to eat only when the conditions for eating are satisfied. Semaphore s[i] is set to 1, there by enabling philosopher i to eat, only when philosopher i wants to eat and both forks are available.

Mutex is used such that no two philosophers may access the pickup or put down at the same time.The array is used to control the behavior of each philosopher.But, semaphores can result in deadlock due to programming errors.

Methods in the code:

void test(int i) : Philosopher i calls test(i) in takeChopstick() to check wether the left and right chopstick are available.If both forks are available,the left and right philosophers are not eating, then sem_post(s[i]) will be called and the call to sem_wait(s[i]) in takechopstick() will not block.So, philosopher i will be able to change the state eating.If the left and right chopstick is unavailable, the corresponding philosopher is eating, then test(i) does not call sem_wait(s[i]) in takechopstick() blocks philosopher i.

Philosopher i also calls test(left(i)) and test(right(i)) when he/she is done eating to unblock the philosophers to his/her left and right if they are waiting to eat and can now eat.

left(int i): number of the philosopher in the left side of i.

right(int i): number of the philosopher in the right side of i.

takechopstick(int i) : If neighbours of i are not eating, i take up the chopsticks.If not available to eat, i waits to be signalled.

putchopstick(int i): i put down the chopsticks, her/his state will be hungry and method tests the right and left philosopher of i for they are hungry.

Problems

Deadlock : If every philosopher pick up the their left chopstick, there is no available chopstick anymore.If philosopher's right chopstick is available he/she pick it up, otherwise keep thinking again.In this case philosophers are stuck.

Starvation : Philosopher 3 and Philosopher 5 starts eating, and philosopher 4 wants to eat, calls takechopstick(4) and gets blocked.Philosopher 3 finishes eating but cannot call sem_post(s[4]) to unblock philosopher 4 because philosopher 5 is still eating.Philosopher 3 starts eating again.Philosopher 5 finished eating but cannot call sem_post(s[4]) to unlock philosopher 4 because philosopher 3 is still eating.Philosopher 5 starts eating again and Philosopher 3 finishes eating but cannot call sem_post(s[4]) to unblock philosopher 4 because philosopher 5 is still eating.Philosopher 4 never gets a chance to eat.