VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELAGAVI-590018, KARNATAKA

ASSIGNMENT ON DINING PHILOSOPHERS SYNCHRONIZATION PROBLEM USING MONITORS

Information Science and Engineering

Submitted by :

SRINIVASA V – 1BI17IS048
SHRIYAN ARCOT – 1BI17IS044
SIDDANTH UMESH – 1BI17IS045
SRIHARI N – 1BI17IS047
SUBHRAMANYA H – 1BI17IS049

Under the Guidance of

Mrs. ROOPA H

Assistant Professor

Dept. of ISE, BIT

BANGALORE INSTITUTE OF TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

K.R. Road, V.V. Puram, Bengaluru-560004

2020-2021

1. INTRODUCTION

Dining-Philosophers Problem – N philosophers seated around a circular table

- There is one chopstick between each philosopher
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once

We need an algorithm for allocating these limited resources(chopsticks) among several processes(philosophers) such that solution is free from deadlock and free from starvation.

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

2. DETAILED EXPLANATION

2.1 Monitor-based Solution to Dining Philosophers

We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. Monitor is used to control access to state variables and condition variables. It only tells when to enter and exit the segment. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

THINKING – When philosopher doesn't want to gain access to either fork.
HUNGRY – When philosopher wants to enter the critical section.
EATING – When philosopher has got both the forks, i.e., he has entered the section.

3. Monitors and condition variables

- semaphores can result in deadlock due to programming errors forgot to add a P() or V(), or misordered them, or duplicated them
- to reduce these errors, introduce high-level synchronization primitives, e.g. *monitors with condition variables*, that essentially automates insertion of P and V for you
  – As high-level synchronization constructs, monitors are found in high-level programming languages like Java and C#
  – underneath, the OS may implement monitors using semaphores and mutex locks.

3.1. Declare a monitor as follows (looks somewhat like a C++ class):

monitor *monitor_name* {

// shared local variables

function f1(...) {

...

}

...

function fN(...) {

...

}

init_code(...) {

...

}

}

3.2. Explanation:

A monitor ensures that only 1 process/thread at a time can be active within a monitor

– simplifies programming, no need to explicitly synchronize
Implicitly, the monitor defines a mutex lock

semaphore mutex = 1;

Implicitly, the monitor also defines essentially mutual exclusion around each function

– Each function's critical code is surrounded as follows:

function fj(...) {

P(mutex)

critical code V(mutex)

}

The monitor's local variables can only be accessed by local monitor functions

Each function in the monitor can only access variables declared locally within the monitor and its parameters

Example:

```
monitor sharedcounter {

int counter;

function add() { counter++;}

function sub() { counter--;}

init() { counter=0; }

}
```

If two processes want to access this sharedcounter monitor, then access is mutually exclusive and only one process at a time can modify the value of counter

– if a write process calls sharedcounter.add(), then it has exclusive access to modifying counter until it leaves add(). No other process, e.g. a read process, can come in and call sharedcounter.sub() to decrement counter while the write process is still in the monitor.

In the previous sharedcounter example, a writer process may be interacting with a reader process via a bounded buffer

– like the solution to the bounded buffer producer/consumer problem, the writer should signal blocked reader processes when there are no longer zero elements in the buffer
– monitors alone don't provide this signalling synchronization capability

In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.

– Thus, monitors alone are insufficient.

– Augment monitors with condition variables.

A condition variable x in a monitor allows two main operations on itself:
– x.wait() -- suspends the calling process until another process calls x.signal()
– x.signal() -- resumes exactly 1 suspended process. If none, then no effect.

Note that x.signal() is unlike the semaphore's signalling operation V(), which preserves state in terms of the value of the semaphore.

– Example: if a process Y calls x.signal() on a condition variable x before process Z calls x.wait(), then Z will wait. The condition variable doesn't remember Y's signal.

– Comparison: if a process Y calls V(mutex) on a binary semaphore mutex (initialized to 0) before process Z calls P(mutex), then Z will not wait, because the semaphore remembers Y's V() because its value = 1, not 0.

– the textbook mentions that a third operation can be performed x.queue()

Declare a condition variable with pseudo-code:

condition x,y;

Semantics concerning what happens just after x.signal() is called by a process P in order to wake up a process Q waiting on this CV x

– Hoare semantics, also called signal-and-wait

The signaling process P either waits for the woken up process Q to leave the monitor before resuming, or waits on another CV

– Mesa semantics, also called signal-and-continue

The signaled process Q waits until the signaling process P leaves the monitor or waits on another condition.

3.3. Key Insight

Key insight: pick up 2 chopsticks only if both are free
– this avoids deadlock
– reword insight: a philosopher moves to his/her eating state only if both neighbors are not in their eating states
thus, need to define a state for each philosopher
– 2nd insight: if one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
thus, states of each philosopher are: thinking, hungry, eating
thus, need condition variables to signal() waiting hungry philosopher(s)
– Also, need to Pickup() and Putdown() chopsticks.

4. Problem Statement(constrains/rules):

is explained in the introduction itself.
example:
consider 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 spoons/chopsticks.
solution:
lets think like this, what happen if all the philosophers start to eat at a same time and get their right spoon first to the hand.

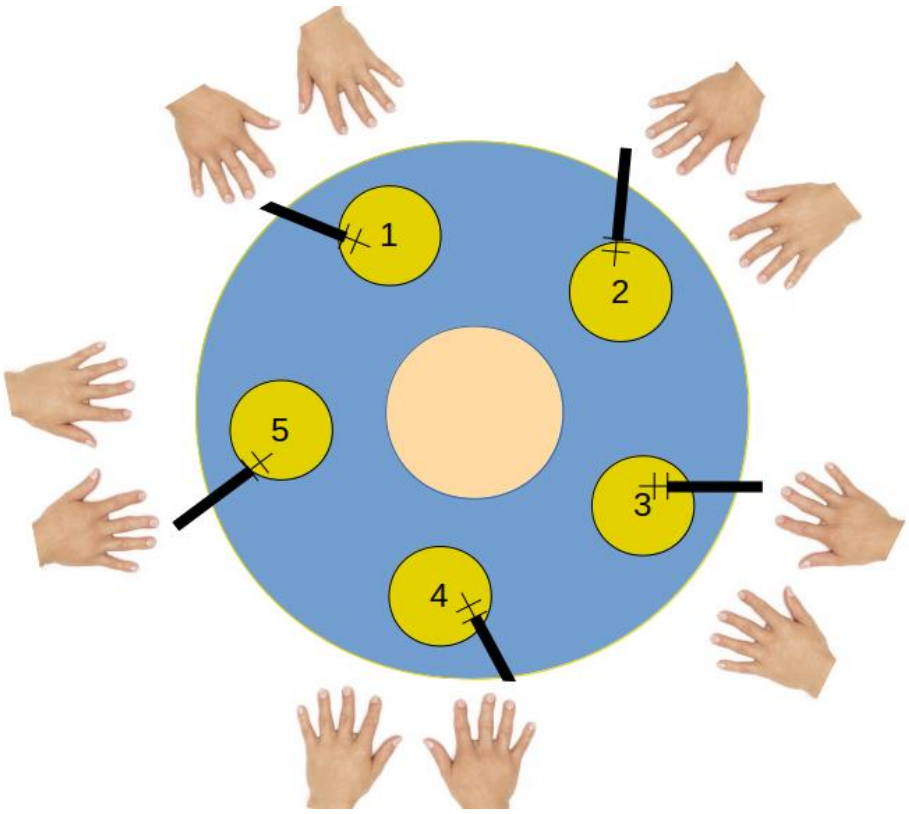

Fig 1: Philosophers thinking state on the dining table

Nothing can do. If both spoons are not available, they can not start eating. No any movement to forward. That means a deadlock occurs. To prevent from that deadlock they should stop doing this again. Then what is the way to do this !.

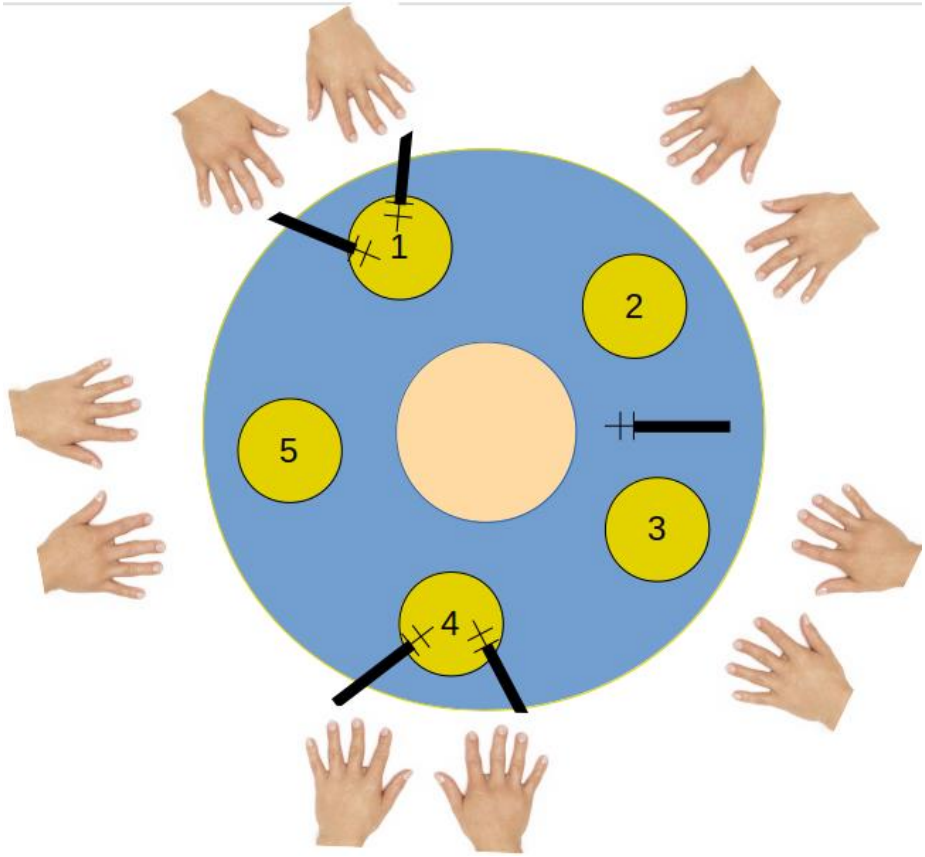

Fig 2: Philosopher eating state on a Dining Table

Think like this, some of them start to eat with both right and left spoons of them and rest of the philosophers keep thinking till others stop eating. This is what should be done by them if all should eat. Continuously this should be done, then all can eat according to this picture no 1 and no 4 are eating, after some time they release that spoons and start thinking again by letting others to eat. Likewise all should eat.

5. SOME DEADLOCK-FREE SOLUTIONS:

– allow at most 4 philosophers at the same table when there are 5 resources
– odd philosophers pick first left then right, while even philosophers pick first right then left
– allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.

6. IMPLEMENTATION AND CODE

6.1 Implementation

We have used JAVA language to write the code. Java is General purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible. It is intended to let application developers write once, run anywhere. Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture.

6.2 Code

```
import java.util.concurrent.locks.Lock;
import java.util.*;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
class Monitor {
        private int numOfPhilosophers;
        final Lock lock;
        private enum States {hungry, thinking, eating};
        private States [] state;
        final Condition [] cond;
        Monitor(int numOfPhil){
                this.numOfPhilosophers = numOfPhil;
                lock = new ReentrantLock();
                state = new States[numOfPhilosophers];
                cond = new Condition[numOfPhilosophers];
                for(int i = 0; i < numOfPhilosophers; i++){
```

```
                state[i] = States.thinking;
                cond[i] = lock.newCondition();
            }
        }

        public void PickUp(int i){
            lock.lock();
            try{
                state[i] = States.hungry;
                if( ( state[(i-1+numOfPhilosophers)%numOfPhilosophers] !=
States.eating ) &&
                   (state[(i+1)%numOfPhilosophers] != States.eating) ){
                    System.out.format("Philosopher %d picks up left chopstick\n",
i+1);
                    System.out.format("Philosopher %d picks up right chopstick\n",
i+1);
                    state[i] = States.eating;
                }
                else {
                    try {
                        cond[i].await();
                        // Eat after waiting.
                        System.out.format("Philosopher %d picks up left
chopstick\n", i+1);
                        System.out.format("Philosopher %d picks up right
chopstick\n", i+1);
                        state[i] = States.eating;
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                }
            }
            finally{
                lock.unlock();
            }
        }

        public void PutDown(int i){
            lock.lock();
            try{
                System.out.format("Philosopher %d puts down right chopstick\n", i+1);
                System.out.format("Philosopher %d puts down left chopstick\n", i+1);
                state[i] = States.thinking;
                int left = (i - 1 + numOfPhilosophers)%numOfPhilosophers;
                int left2 = (i - 2 + numOfPhilosophers)%numOfPhilosophers;
```

```
                        if( (state[left] == States.hungry) &&
                                (state[left2] != States.eating) ){
                                cond[left].signal();
                        }
                        if( (state[(i+1)%numOfPhilosophers] == States.hungry) &&
                                (state[(i+2)%numOfPhilosophers] != States.eating) ){
                                cond[(i+1)%numOfPhilosophers].signal();
                        }
                }
                finally {
                        lock.unlock();
                }
        }
}

public class Philosophers implements Runnable {
        // Private data.
        private int myId;
        private int timesToEat;
        private Monitor mon;
        private Thread t;
        private int sleepLength;
        Philosophers(int id, int numToEat, Monitor m){
                this.myId = id;
                this.timesToEat = numToEat;
                this.mon = m;
                sleepLength = 2000;
                t = new Thread(this);
                t.start();
        }
        @Override
        public void run() {
                int count = 1;
                while(count <= timesToEat ){
                        mon.PickUp(myId);
                        mon.PutDown(myId);
                        eat(count);
                        ++count;
                }
        }
        void eat(int count){
                System.out.format("Philosopher %d eats (%d times)\n", myId+1, count);
                try {
                   Thread.sleep(sleepLength);
                }
```

```
            catch (InterruptedException e) {}
        }

        public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            int numOfPhilosophers=0,timesToEat=0;
            System.out.println("Enter the number of philosophers");
            numOfPhilosophers=sc.nextInt();
            System.out.println("Enter the number of times to eat");
            timesToEat=sc.nextInt();

            Monitor mon = new Monitor(numOfPhilosophers);
            Philosophers [] p = new Philosophers[numOfPhilosophers];

            System.out.println("Dinner is starting!");
            System.out.println("");

            for(int i = 0; i < numOfPhilosophers; i++)
                p[i] = new Philosophers(i, timesToEat, mon);

            for(int i = 0; i < numOfPhilosophers; i++)
                try {
                    p[i].t.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            System.out.println("");
            System.out.println("Dinner is over!");
        }
        }
}

}
```

7. SNAPSHOTS

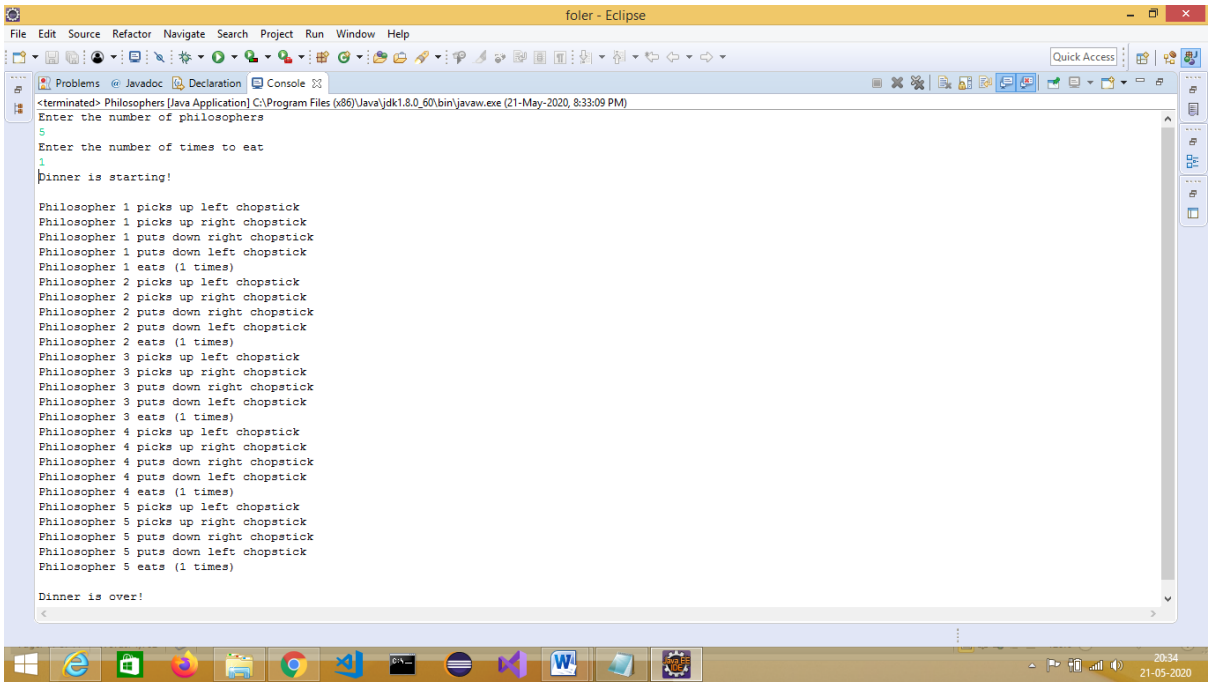

Fig 3: Output for the Manual Input