PROCESS_SYNCHRONIZATION_3

BCSE303P Operating Systems

Name: Chillara V L N S Pavana Vamsi

Register Number: 21BCE5095

1. Develop a monitor code for the Dining Philosopher problem using threads and monitor in Java. Discuss how it achieves a deadlock free solution

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Chopstick {
    private boolean availability;

    public Chopstick() {
        availability = true;
    }

    public boolean getAvailability() {
        return availability;
    }

    public void setAvailability(boolean flag) {
        availability = flag;
    }
}

class Helper {
    private Lock mutex = null;
    private Condition[] cond;
    private String[] state;
    private int[] id;

    private void outputState(int id) {
        StringBuffer line = new StringBuffer();
        for (int i = 0; i < 5; i++)
            line.append(state[i] + " ");
        System.out.println(line + "(" + (id + 1) + ")");
    }

    public Helper() {
        id = new int[5];
        mutex = new ReentrantLock();
        state = new String[5];

        cond = new Condition[5];
        for (int i = 0; i < 5; i++) {
            id[i] = i;
            state[i] = "O";
            cond[i] = mutex.newCondition();
        }
    }

    public void setState(int id, String s) {
        state[id] = s;
    }

    public void grabChopsticks(int id, Chopstick l, Chopstick r) {
```

```
        mutex.lock();
        try {
            setState(id, "o");
            while (!l.getAvailability() || !r.getAvailability())
                cond[id].await();
            l.setAvailability(false);
            r.setAvailability(false);
            setState(id, "X");
            outputState(id);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            mutex.unlock();
        }
    }

    public void releaseChopsticks(int id, Chopstick l, Chopstick r) {
        mutex.lock();
        try {
            setState(id, "O");
            l.setAvailability(true);
            r.setAvailability(true);
            cond[(id + 1) % 5].signalAll();
            cond[(id + 4) % 5].signalAll();
            outputState(id);
        } finally {
            mutex.unlock();
        }
    }
}

class Philosopher implements Runnable {
    private Helper hlp;
    private Chopstick l, r;
    private int id;

    public Philosopher(int id, Chopstick l, Chopstick r, Helper i) {
        this.hlp = i;
        this.l = l;
        this.r = r;
        this.id = id;
    }

    private void eat() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
        }

    }

    private void think() {
        try {
            Thread.sleep(2000);
```

```
        } catch (InterruptedException e) {
        }
    }

    public void run() {
        while (true) {
            hlp.grabChopsticks(id, l, r);
            eat();
            hlp.releaseChopsticks(id, l, r);
            think();
        }
    }
}

public class Problem {
    private Chopstick[] s;
    private Philosopher[] f;
    private Helper hlp;

    private void init() {
        s = new Chopstick[5];
        f = new Philosopher[5];
        hlp = new Helper();
        for (int i = 0; i < 5; i++)
            s[i] = new Chopstick();
        for (int i = 0; i < 5; i++) {
            f[i] = new Philosopher(i, s[i], s[(i + 4) % 5], hlp);
            new Thread(f[i]).start();
        }
    }

    public Problem() {
        init();
    }

    public static void main(String[] args) {
        new Problem();
    }
}
```

Or

```
class DiningPhilosophersMonitor {
    private boolean[] forks = new boolean[2];

    public synchronized void pickup(int philosopherId) throws InterruptedException {
        int leftFork = philosopherId;
        int rightFork = (philosopherId + 1) % 5;
        while (forks[leftFork] || forks[rightFork]) {
            wait();
        }
        forks[leftFork] = true;
```

```
        forks[rightFork] = true;
    }

    public synchronized void putdown(int philosopherId) {
        int leftFork = philosopherId;
        int rightFork = (philosopherId + 1) % 5;
        forks[leftFork] = false;
        forks[rightFork] = false;
        notifyAll();
    }
}

class Philosopher implements Runnable {
    private int id;
    private DiningPhilosophersMonitor monitor;

    public Philosopher(int id, DiningPhilosophersMonitor monitor) {
        this.id = id;
        this.monitor = monitor;
    }

    @Override
    public void run() {
        try {
            while (true) {
                System.out.println("Philosopher " + id + " is thinking");
                Thread.sleep(1000);
                System.out.println("Philosopher " + id + " is hungry");
                monitor.pickup(id);
                System.out.println("Philosopher " + id + " is eating");
                Thread.sleep(1000);
                monitor.putdown(id);
            }
        } catch (InterruptedException e) {
        }
    }
}

public class code2 {
    public static void main(String[] args) throws InterruptedException {
        DiningPhilosophersMonitor dpMon = new DiningPhilosophersMonitor();
        Thread t1 = new Thread(new Philosopher(0, dpMon));
        Thread t2 = new Thread(new Philosopher(1, dpMon));
        Thread t3 = new Thread(new Philosopher(2, dpMon));
        Thread t4 = new Thread(new Philosopher(3, dpMon));
        Thread t5 = new Thread(new Philosopher(4, dpMon));
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
    }
}
```

2. Trace the algorithm for all 5 philosophers had or tries to have noodle successfully with minimum 2 combinations (Ex. 1.P5,P4,P3,P2,P1 (every process makes request after the previous one finishes the eating stage. 2. P3,P4,(P3,P4 finishes eating successfully one after another and P5 enters in eating stage after P4. Then P5 (decides not to putdown the forks) but P1,P2 requests for noodles concurrently. Trace the algorithm manually and upload the written work along with the coded questions

Ans. Initially,
All philosophers are in thinking state.
Ps, P4, P3, P2, P,
(a) P5:
takes the left and right forks state [5]=eating
P5 puts forks down.
(b) P4 takes L and R forks state [4] = eating
P4 puts forks down
(c) P3 takes L and R forks. state (3] = eating
P3 puts forks down
(d) P2 takes 1 and R forks state [2] = eating
P2 puts forks down
(e) P1 takes L and R forks State (1) = eating
P1 puts forks down

➔ P3 takes L and R forks =>state [3] = eating
P3 puts forks down. => stale [3]= thinking
➔ P4 takes L and R. forks. => state [4] = eating
P4 puts forks down => state [4] = thinking.
Now,
-> P5 takes the L and R forks
=> state (5) = eating
Here,
P5 doesn't put down forks.
Now,
P1 and P2 are requesting for noodles for P1,
L fork is unavailable since P5 didn't put forks down.
Thus, P1 has to wait for L fork Also, P2 has to wait for P1 to eat.
There occurs a deadlock.

3. Write a pseudo code in C to achieve a deadlock-free D-P problem using the monitor concept either as a word or written document.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define N 5
typedef enum
{
    THINKING,
    HUNGRY,
    EATING
} state_t;
pthread_mutex_t monitor_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t philo_cond[N] = {PTHREAD_COND_INITIALIZER};
state_t state[N] = {THINKING};
int philo_id[N] = {0, 1, 2, 3, 4};
void pickup_forks(int philo_id)
{
    pthread_mutex_lock(&monitor_mutex);
    state[philo_id] = HUNGRY;
    test(philo_id);
    if (state[philo_id] != EATING)
    {
        pthread_cond_wait(&philo_cond[philo_id], &monitor_mutex);
    }
    pthread_mutex_unlock(&monitor_mutex);
}
void putdown_forks(int philo_id)
{
    pthread_mutex_lock(&monitor_mutex);
    state[philo_id] = THINKING;
    test((philo_id + N - 1) % N);
    test((philo_id + 1) % N);
    pthread_mutex_unlock(&monitor_mutex);
}
void test(int philo_id)
{
    if (state[(philo_id + N - 1) % N] != EATING &&
        state[philo_id] == HUNGRY &&
        state[(philo_id + 1) % N] != EATING)
    {
        state[philo_id] = EATING;
        pthread_cond_signal(&philo_cond[philo_id]);
    }
}
void *philosopher(void *arg)
{
    int *philo_id = (int *)arg;
    while (1)
    {
        printf("Philosopher %d is thinking...\n", *philo_id);
        sleep(1);
```

```
        printf("Philosopher %d is hungry...\n", *philo_id);
        pickup_forks(*philo_id);
        printf("Philosopher %d is eating...\n", *philo_id);
        sleep(1);
        putdown_forks(*philo_id);
    }
}
int main()
{
    pthread_t philo_threads[N];
    int i;
    for (i = 0; i < N; i++)
    {
        pthread_create(&philo_threads[i], NULL, philosopher, &philo_id[i]);
    }
    for (i = 0; i < N; i++)
    {
        pthread_join(philo_threads[i], NULL);
    }
    return 0;
}
```

4. Develop a Readers- Writers Problem using the monitor in Java:

```
import java.util.*;

class Monitor {
    private volatile int readers; // specifies number of readers reading
    private volatile boolean writing; // specifies if someone is writing
    private volatile Condition OK_to_Read, OK_to_Write;

    public Monitor() {
        readers = 0;
        writing = false;
        OK_to_Read = new Condition();
        OK_to_Write = new Condition();
    }

    public synchronized void Start_Read(int n) {
        System.out.println("wants to read " + n);
        if (writing || OK_to_Write.is_non_empty()) {
            try {
                System.out.println("reader is waiting " + n);
                OK_to_Read.sleep_();
            } catch (InterruptedException e) {
                }
        }
        readers += 1;
        OK_to_Read.release_all();
    }

    public synchronized void End_Read(int n) {
        System.out.println("finished reading " + n);
```

```
        readers -= 1;
        if (OK_to_Write.is_non_empty()) {
            OK_to_Write.release_one();
        } else if (OK_to_Read.is_non_empty()) {
            OK_to_Read.release_one();
        } else {
            OK_to_Write.release_all();
        }
    }

    public synchronized void Start_Write(int n) {
        System.out.println("wants to write " + n);
        if (readers != 0 || writing) {

            try {
                System.out.println("Writer is waiting " + n);
                OK_to_Write.sleep_();
            } catch (InterruptedException e) {
            }
        }
        writing = true;
    }

    public synchronized void End_Write(int n) {
        System.out.println("finished writing " + n);
        writing = false;
        if (OK_to_Read.is_non_empty()) {
            OK_to_Read.release_one();
        } else if (OK_to_Write.is_non_empty()) {
            OK_to_Write.release_one();
        } else {
            OK_to_Read.release_all();
        }
    }
}

class Condition {
    private int number;// specifies the number of readers/writers waiting

    public Condition() {
        number = 0;
    }

    public synchronized boolean is_non_empty() {
        if (number == 0)
            return false;
        else
            return true;
    }

    public synchronized void release_all() {
        number = 0;
        notifyAll();
    }
```

```
    public synchronized void release_one() {
        number -= 1;
        notify();
    }

    public synchronized void wait_() throws InterruptedException {
        number++;
        wait();
    }

    public synchronized void sleep_() throws InterruptedException {
        Thread.sleep(1000);
    }
}

class Reader extends Thread {
    private Monitor M;
    private String value;

    public Reader(String name, Monitor c) {
        super(name);
        M = c;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            M.Start_Read(i);
            // System.out.println("Reader "+getName()+" is retreiving data...");
            System.out.println("Reader is reading " + i);
            M.End_Read(i);
        }
    }
}

class Writer extends Thread {
    private Monitor M;
    private int value;

    public Writer(String name, Monitor d) {
        super(name);
        M = d;
    }

    public void run() {
        for (int j = 0; j < 10; j++) {
            M.Start_Write(j);
            // System.out.println("Writer "+getName()+" is writing data...");
            System.out.println("Writer is writing " + j);
            M.End_Write(j);
        }
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Monitor M = new Monitor();
        Reader reader = new Reader("1", M);
        Writer writer = new Writer("1", M);
        writer.start();
        reader.start();
    }
}
```

Output:

wants to write 0

Writer is writing 0

finished writing 0

wants to write 1

Writer is writing 1

finished writing 1

wants to write 2

Writer is writing 2

finished writing 2

wants to write 3

Writer is writing 3

finished writing 3

wants to write 4

Writer is writing 4

finished writing 4

wants to write 5

Writer is writing 5

finished writing 5

wants to write 6

Writer is writing 6

finished writing 6

wants to write 7

Writer is writing 7

finished writing 7

wants to write 8

Writer is writing 8

finished writing 8

wants to write 9

Writer is writing 9

finished writing 9

wants to read 0

Reader is reading 0

finished reading 0

wants to read 1

Reader is reading 1

finished reading 1

wants to read 2

Reader is reading 2

finished reading 2

wants to read 3

Reader is reading 3

finished reading 3

wants to read 4

Reader is reading 4

finished reading 4

wants to read 5

Reader is reading 5

finished reading 5

wants to read 6

Reader is reading 6

finished reading 6

wants to read 7

Reader is reading 7

finished reading 7

wants to read 8

Reader is reading 8

finished reading 8

wants to read 9

Reader is reading 9

finished reading 9