OS Lab Test

U20CS110

Krishna Pandey

**Problem Statement**

**Even Numbers**

1**. The Dining Philosopher Problem states that 5 philosophers seated around a circular table with one chopstick between each pair of philosophers. ● 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 Solve this problem by using Semaphore so that at least one can eat at a time.**

import sys

import threading

import time

class Semaphore():

    def \_\_init\_\_(self, initial):

        self.lock = threading.Condition(threading.Lock())

        self.value = initial

    def up(self):

        with self.lock:

            self.value += 1

            self.lock.notify()

    def down(self):

        with self.lock:

            while self.value == 0:

                self.lock.wait()

            self.value -= 1

class ChopStick():

    def \_\_init\_\_(self, number):

        self.number = number           # chop stick ID

        self.user = -1                 # keep track of philosopher using it

        self.lock = threading.Condition(threading.Lock())

        self.taken = False

    def take(self, user):         # used for synchronization

        with self.lock:

            while self.taken == True:

                self.lock.wait()

            self.user = user

            self.taken = True

            sys.stdout.write(

                "Philosopher[%s] took ChopStick[%s]\n" % (user, self.number))

            self.lock.notify\_all()

    def drop(self, user):         # used for synchronization

        with self.lock:

            while self.taken == False:

                self.lock.wait()

            self.user = -1

            self.taken = False

            sys.stdout.write(

                "Philosopher[%s] dropped ChopStick[%s]\n" % (user, self.number))

            sys.stdout.write("Philosopher[%s] THINKING\n" % (user))

            self.lock.notify\_all()

class Philosopher (threading.Thread):

    def \_\_init\_\_(self, number, left, right, butler):

        threading.Thread.\_\_init\_\_(self)

        self.number = number            # philosopher number

        self.left = left

        self.right = right

        self.butler = butler            # philosopher as butler

    def run(self):

        for i in range(1):

            time.sleep(1)

            self.butler.down()              # start service by butler

            time.sleep(1)                 # think

            self.left.take(self.number)     # pickup left chopstick

            time.sleep(1)      # (yield makes deadlock more likely)

            self.right.take(self.number)    # pickup right chopstick

            time.sleep(1)                # eat

            sys.stdout.write("Philosopher[%s] EATING\n" % (self.number))

            self.right.drop(self.number)    # drop right chopstick

            time.sleep(1)

            self.left.drop(self.number)     # drop left chopstick

            time.sleep(1)

            self.butler.up()              # end service by butler

            time.sleep(1)

        sys.stdout.write(

            "Philosopher[%s] finished THINKING & EATING\n" % self.number)

def dinner():

    # number of philosophers / chop sticks

    n = int(input('Enter Total Philosophers : '))

    sys.stdout.write("\n")

    # butler for deadlock avoidance (n-1 available)

    butler = Semaphore(n-1)

    # list of chopsticks

    c = [ChopStick(i) for i in range(n)]

    # list of philsophers

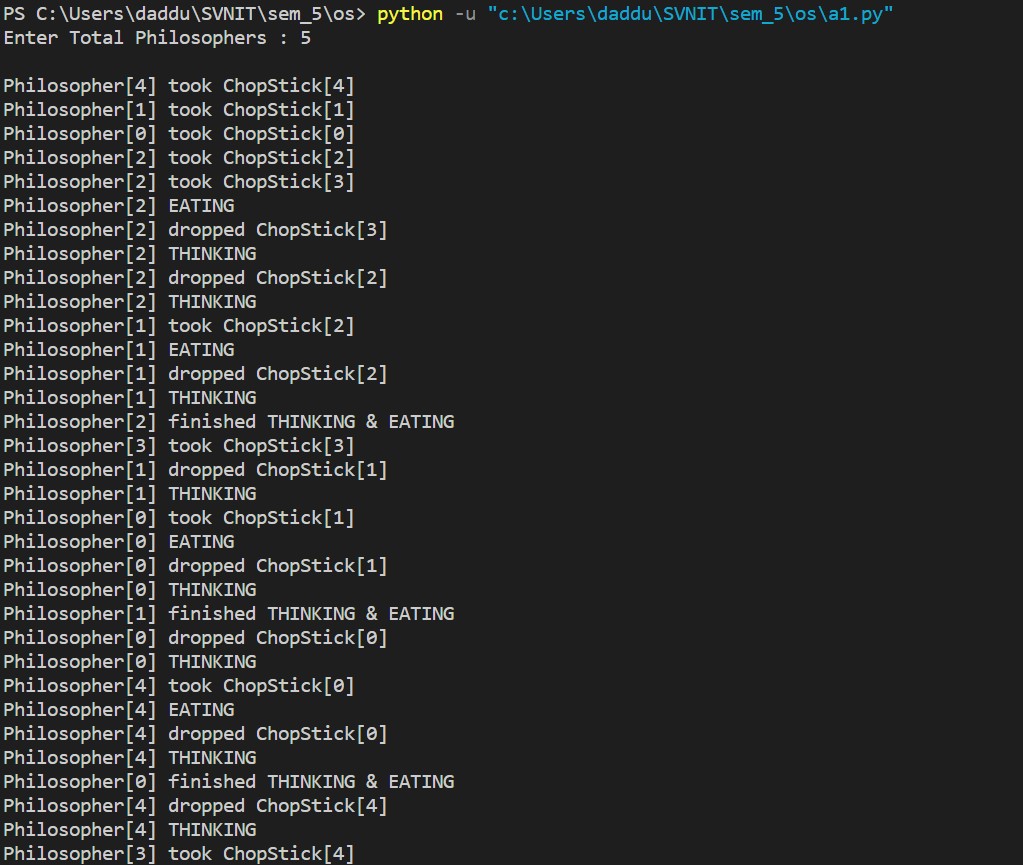
    p = [Philosopher(i, c[i], c[(i+1) % n], butler) for i in range(n)]

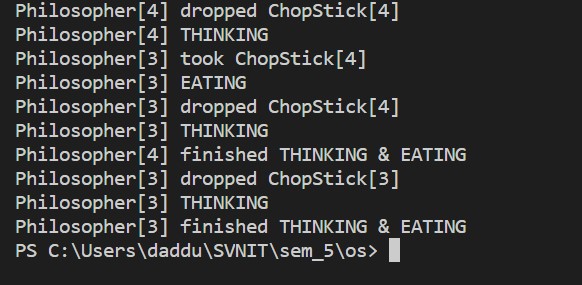
    for i in range(n):

        p[i].start()

dinner()

**output:**





**2. Implement an algorithm for deadlock detection. It may contain given datasets: Available Vector of length m Indicates a number of available resources of each type. Allocation Matrix of size n\*m A[i,j] indicates the number of jth resource type allocated to the ith process. Request Matrix of size n\*m Indicates request of each process. Request[i,j] tells the number of instances the Pi process is requesting of jth resource type. The output should indicate whether there is any deadlock or not.**

**Ans:**

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int n, m, i, j, k;

    // n = 5

    cout << "enter the Number of processes and resources: ";

    cin >> n >> m;

    // m = 3

    int alloc[n][m], max[n][m];

    cout << "enter the allocation matrix";

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < m; j++)

            cin >> alloc[i][j];

    }

    cout << "enter the max matrix";

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < m; j++)

            cin >> max[i][j];

    }

    // int avail[3] = {3, 3, 2}; // Available Resources

    int avail[m];

    cout << "Number of available Resources";

    for (int i = 0; i < m; i++)

    {

        cin >> avail[i];

    }

    int f[n], ans[n], ind = 0;

    for (k = 0; k < n; k++)

    {

        f[k] = 0;

    }

    int need[n][m];

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < m; j++)

            need[i][j] = max[i][j] - alloc[i][j];

    }

    int y = 0;

    for (k = 0; k < n; k++)

    {

        for (i = 0; i < n; i++)

        {

            if (f[i] == 0)

            {

                int flag = 0;

                for (j = 0; j < m; j++)

                {

                    if (need[i][j] > avail[j])

                    {

                        flag = 1;

                        break;

                    }

                }

                if (flag == 0)

                {

                    ans[ind++] = i;

                    for (y = 0; y < m; y++)

                        avail[y] += alloc[i][y];

                    f[i] = 1;

                }

            }

        }

    }

    int flag = 1;

    // To check if sequence is safe or not

    for (int i = 0; i < n; i++)

    {

        if (f[i] == 0)

        {

            flag = 0;

            cout << "Deadlock occurs";

            break;

        }

    }

    if (flag == 1)

    {

        cout << "Deadlock will not occur and process sequence will be Sequence" << endl;

        for (i = 0; i < n - 1; i++)

            cout << " P" << ans[i] << " ->";

        cout << " P" << ans[n - 1] << endl;

    }

    return 0;

}

**Output:**

