YAŞAR UNIVERSITY FACULTY OF ENGINEERING

COMP 3323 Operating Systems Laboratory Sheet

LAB 7: Two Classical IPC Problems

DINING PHILOSOPHERS

The Problem

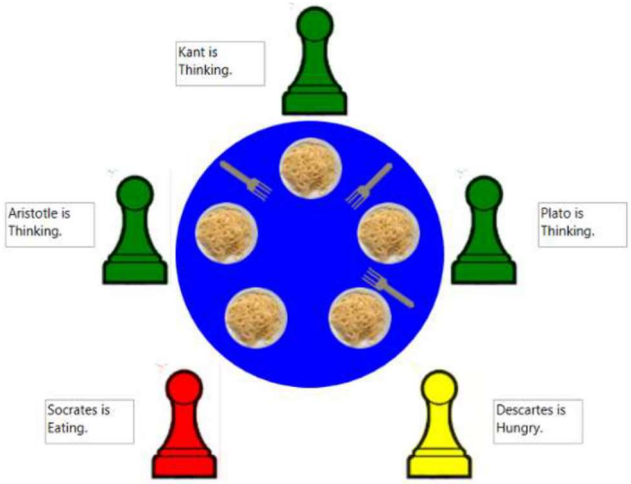
The problem is to design a simulation that enables all philosophers to continue to eat and think without any form of communication between participants. It follows from this that nobody is allowed to die of ***starvation*** and that no ***deadlocks*** can occur.

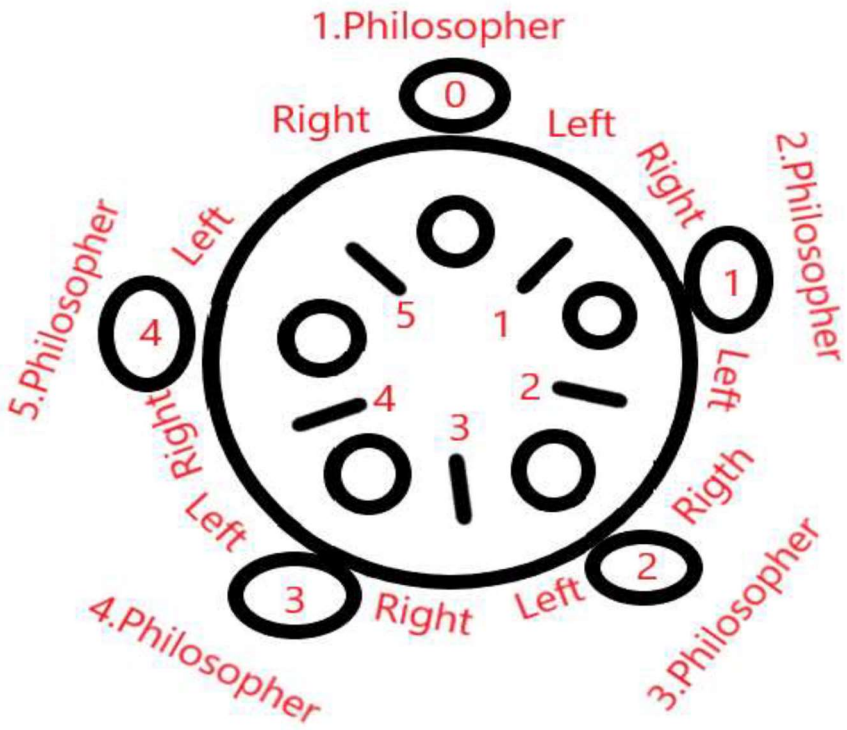
Analysis of the Problem

Although the scenario mentions just two states that a philosopher can be in, namely, eating or thinking, there is a third state and that is, thinking but looking for an opportunity to eat. In other words, the philosopher is hungry. So the philosopher's activities can be broken down into three consecutive phases, Thinking, Hungry and Eating.

<u>There are a few conditions:</u>

- ✓ Philosophers can only fetch chopsticks placed between them and their neighbors.
- ✓ Philosophers can not take their neighbors' chopsticks away while they are eating.
- ✓ Hopefully no philosophers should starve to death (i.e. wait over a certain amount of time before he acquires both chopsticks).





Dining Philosophers (semaphores)

```
#define N               5                   /* number of philosophers */
#define LEFT            (i+N−1)%N           /* number of i's left neighbor */
#define RIGHT           (i+1)%N             /* number of i's right neighbor */
#define THINKING        0                   /* philosopher is thinking */
#define HUNGRY          1                   /* philosopher is trying to get forks */
#define EATING          2                   /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                          /* repeat forever */
        think( );                           /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat( );                             /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}
```

Dining Philosophers (contd)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}
void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

READERS WRITERS PROBLEM

The Problem

A data object is shared among several concurrent processes. ***Readers*** Processes that only want to read the shared object. ***Writers*** Processes that want to update (read and write) the shared object. Any number of readers may access the data at one time, but writers must have exclusive access.

Write a program for these two processes in such a way that no process shall be allowed to starve; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.

Solution Pseudocode

```
binary semaphores in_line =1;
binary semaphores db=1;
binary semaphores mutex=1;
int readcount =0;

void writer () {
      down(in_line);
      down( db);
      CRITICAL REGION to write
      up(in_line);
      up(db);
}
```

```
void reader () {
      int count1, count2;
      down(in_line);
      down(mutex);
      count1 = readcount ;
      readcount = readcount + 1;
      up(mutex);
      if count1 ==0  then down(db);
      up(in_line);
      CRITICAL REGION to read
      down(mutex);
      readcount = readcount - 1;
      count2 = nreaders;
      up(mutex);
      if count2==0 then up(db);
}
```