Project 1 - Dining Philosophers Problem

LCMPE 312

Alpay Özkeskin

May 2017

1 Description of Dining Philosophers Problem

The dining philosphers problem is a synchronization problem posed and solved by **Edsger W. Dijkstra** in 1965, in the paper titled **Hierarchical ordering of sequential processes**.[1] Since then, it has become the standard of testing new synchronization algorithms.[2] In the problem, the life of a philosopher consists of the loop of two different actions; thinking and eating.

```
while (true) {
    think();
    eat();
}
```

Five of these philosophers which are numbered from 0 to 4 are living together in a house where they dine on a round table. Each of the philosophers have their own plate and one fork on their right. However their dinner is a slippery spagetti, which requires two forks to eat (Figure 1). The purpose of the problem is designing an algorithm, so that each philosopher eats his dinner, and none of them starves.

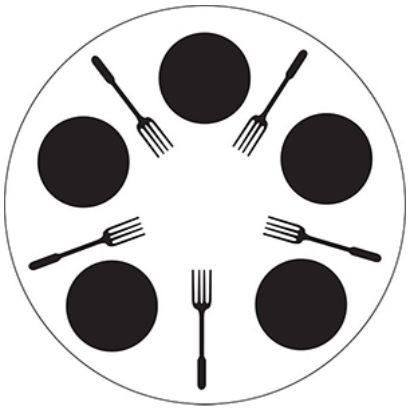

Figure 1: Dining Philosophers Problem

2 Main Problem of the Philosophers and the Solutions

The main problem of the philosophers is that no two neighbouring philosophers can eat simultaneously.Therefore only way for a philosopher to start eating is when both left and right neighbours enter thinking state.[1]
The first and easiest solution is assigning a binary semaphore to each fork, 0 meaning the fork is in use and 1 meaning the fork is available, so that no two

neighbours can eat simultaneously. However this scenario contains the danger of deadlock. If every philosopher grab their left side fork, then no philosopher can eat and they starve. In order to overcome this danger philosophers need to check both left and right handside forks before taking them. To provide this checking, we need to use a variable that describes the state of philosophers. If we call this variable 'C' where:

```
C[i] = 0 // philosopher i is thinking.
C[i] = 1 // philosopher i is hungry.
C[i] = 2 // philosopher i is eating.
```

The hungry state is added to ensure no philosopher ever starves. In the current scenario, for a philosopher p to enter eating state, there exists three conditions that need to be satisfied.

```
1) C[p] = 1           // p must be in hungry state.
2) C[p + 1 % 5] != 2  // p's left neighbour should not be eating.
3) C[p - 1 % 5] != 2  // p's right neighbour should not be eating.
```

When the conditions are satisfied, C[p] becomes 2 and p starts eating. In order to send the hungry philosophers to tables two tests needs to be done. First is the case when a philosopher becomes hungry, he needs to check if he can start eating. Second one is that, when a philosophers stops eating and starts thinking, the neighbours of him should be checked whether they are hungry and whether they can start eating.
To put this scenario into computer code, we need to define a set of variables. These are:

```
A binary Semaphore mutex,initialized as 1.
An integer array C[5], each element initialized as 0.
A semaphore array S[5], each element initialized as 0.
A function test(int p), to test the availability of forks.
```

The test function is as follows:

```
void test(int p) {
    if(C[p - 1 % 5] != 2 && C[p + 1 % 5] != 2 && C[p] == 1) {
        C[p] = 2;
        UP(s[p])[1];
    }
}
```

Now the scenario and the environment is ready to produce the computer code for the solution of **the Dining Philosophers Problem**.

[1] up is one of the two atomic functions that can be used with a semaphore.
DOWN(s) waits until s is greater than 0 and decrements s.
UP(s) increments s.

3 C program That Tries to Solve the Problem

```
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

sem_t mutex;
int C[5] = {0};
sem_t s[5];
int id[5] = {0,1,2,3,4};

void eat(int p) {
        printf("Philosopher %d is eating\n", p);
        sleep(2);
}
void think(int p) {
        printf("Philosopher %d is thinking\n", p);
}

void hungry(int p) {
        printf("Philosopher %d is hungry\n", p);
}
//test function to see if a neighbour can start eating
void test(int p) {
        if(C[(p + 5 - 1) % 5] != 2 && C[(p + 1) % 5] != 2 && C[p] == 1) {
C[p] = 2;
eat(p);
sem_post[2](&s[p]);
        }
}

void * dine(void * arg) {
        while(1) {
                int * w = (int *) arg;
                sleep(1);
                sem_wait(&mutex); //critical section
                        C[*w] = 1;
                        hungry(*w);
                        test(*w);
                sem_post(&mutex); //end of critical section
                sem_wait[3](&s[*w]); //waits until &s[*w] is posted
                sleep(1);
```

[2]Equivalent of UP(s) function in C
[3]Equivalent of DOWN(s) in C

```
                sleep(0);
                sem_wait(&mutex); //critical section
                        C[*w] = 0;
                        think(*w);
                        test((*w + 1) % 5);
                        test((*w + 5 - 1) % 5);
                sem_post(&mutex); //end of critical section
}
}

int main(int argc, char **argv) {
        sem_init(&mutex, 0, 1); //mutex initialization to 1
        pthread_t t[5]; //5 threads representing philosophers
        int i;
        for(i = 0;i < sizeof(C) / sizeof(C[0]);i++) {
                sem_init(&s[i], 0, 0);
        }
        for(i = 0;i < sizeof(C) / sizeof(C[0]);i++) {
                pthread_create(&t[i], NULL, dine, &id[i]);
                printf("Philosopher %d has sat at the table\n", i);
        }
        for(i = 0;i < sizeof(C) / sizeof(C[0]);i++) {
                pthread_join(t[i], NULL);
        }
}
```

4 Evaluation of the Program

References

[1] Dijkstra, E. W. (1965). Hierarchical ordering of sequential processes.

[2] Tanenbaum, A. S., & Bos, H. (2013). Modern operating systems (4th ed.). Harlow: Pearson.