Parallel Programming Project Report

Fall 2023

Senem Ürkmez- 200316014

Barış Akın- 200316061

Ragıp Günay- 200316007

Emir Sercan Korkmazgil- 200316055

1. Introduction

Five philosophers dine together at the same table. Each philosopher has his own plate at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat his spaghetti when he has both a left and right fork. Thus two forks will only be available when his two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, he will put down both forks. The problem is how to design a regimen (a concurrent algorithm) such that any philosopher will not starve; *i.e.*, each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think (an issue of incomplete information).

1.1 Problem

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows:

- think unless the left fork is available; when it is, pick it up;
- think unless the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- put the left fork down;
- put the right fork down;
- repeat from the beginning.

However, each philosopher will think for an undetermined amount of time and may end up holding a left fork thinking, staring at the right side of the plate, unable to eat because there is no right fork, until he starves.

Resource starvation, mutual exclusion and livelock are other types of sequence and access problems. There are a lot of dtrategy for solve this problem. We focus on the Chandy/Misra Solution.

1.2 Chandy/Misra Solution

In 1984, K. Mani Chandy and J. Misra proposed a different solution to the dining philosophers problem to allow for arbitrary agents (numbered P_1, ..., P_n) to contend for an arbitrary number of resources, unlike Dijkstra's solution. It is also completely distributed and requires no central authority after initialization. However, it violates the requirement that "the philosophers do not speak to each other" (due to the request messages).

1. For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID (*n* for agent P_n). Each fork can either be *dirty* or *clean.* Initially, all forks are dirty.
2. When a philosopher wants to use a set of resources (*i.e.*, eat), said philosopher must obtain the forks from his contending neighbors. For all such forks the philosopher does not have, he sends a request message.
3. When a philosopher with a fork receives a request message, he keeps the fork if it is clean, but give it up when it is dirty. If the philosopher sends the fork over, he cleans the fork before doing so.
4. After a philosopher is done eating, all his forks become dirty. If another philosopher had previously requested one of the forks, the philosopher that has just finished eating cleans the fork and sends it.

This solution also allows for a large degree of concurrency, and will solve an arbitrarily large problem.

It also solves the starvation problem. The clean/dirty labels act as a way of giving preference to the most "starved" processes, and a disadvantage to processes that have just "eaten". One could compare their solution to one where philosophers are not allowed to eat twice in a row without letting others use the forks in between. Chandy and Misra's solution is more flexible than that, but has an element tending in that direction.

In their analysis, they derive a system of preference levels from the distribution of the forks and their clean/dirty states. They show that this system may describe a directed acyclic graph, and if so, the operations in their protocol cannot turn that graph into a cyclic one. This guarantees that deadlock cannot occur. However, if the system is initialized to a perfectly symmetric state, like all philosophers holding their left side forks, then the graph is cyclic at the outset, and their solution cannot prevent a deadlock. Initializing the system so that philosophers with lower IDs have dirty forks ensures the graph is initially acyclic.

2. Project Implementation

In this solution, each fork is represented by a **Fork** class, and each philosopher is represented by a **Philosopher** class. The key features of the **Fork** class include:

```
2 usages
def acquire(self, philosopher_id, priority):
    with self.lock:
        if self.dirty or (self.owner == philosopher_id and self.priority <= priority):
            self.owner = philosopher_id
            self.dirty = False#Catalı temize çeker
            self.priority = priority
            return self
        return None
```

- **'acquire'**: Allows a philosopher to acquire the fork. If the philosopher can acquire it, it updates the dirty state and returns the fork; otherwise, it returns '**None**'.

```
4 usages
def release(self):
    with self.lock:
        self.dirty = True
        self.owner = -1
        self.priority = 0
```

- **'release'**: Resets the dirty state and owner of the fork.

```
1 usage
def request_forks(self):
    while True:
        # Acquire left fork
        left_acquired = self.left_fork.acquire(self.index, self.index)
        # Acquire right fork
        right_acquired = self.right_fork.acquire(self.index, self.index)

        if left_acquired and right_acquired:
            # Both forks are clean, break the loop
            break
        else:
            # Release the acquired fork(s) and try again
            if left_acquired:
                self.left_fork.release()
            if right_acquired:
                self.right_fork.release()
```

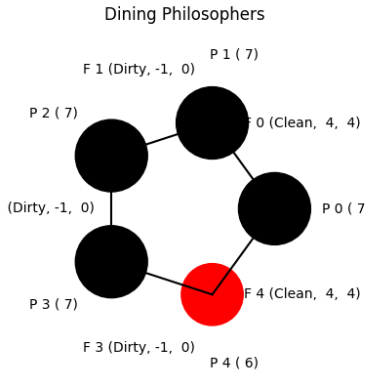
- The '`request_forks`' method ensures that a philosopher acquires both the left and right forks before proceeding to eat. If acquisition of either fork fails, the philosopher releases any acquired fork and retries until successful. This process helps prevent deadlocks by ensuring that a philosopher only eats when it can acquire both necessary forks.

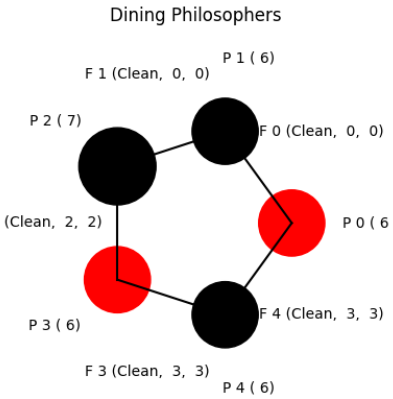
```
# After eating, mark the forks as dirty
self.left_fork.release()
self.right_fork.release()
```

The **Philosopher** class has a **run** method that performs thinking and eating actions. The **think** method simulates thinking for a specific duration, while the **eat** method simulates the process of requesting forks and eating. After eating, the philosopher releases the forks, allowing other philosophers to use them.

For visualization, the **table** and **animated_table** functions are used. The **table** function provides a simple console table output, and the **animated_table** function uses the matplotlib library to create an animation. These functions allow us to visually track the state of philosophers and forks.

The overall purpose of the code is to enable philosophers to eat and think concurrently by coordinating fork requests appropriately and, in the process, correctly sharing forks. The animation and table output visually depict the progression of actions and fork sharing among philosophers.





3. Conclusion

The Dining Philosophers problem encapsulates the challenges of concurrent resource sharing and synchronization, providing an illustrative scenario for studying deadlock, contention, and the importance of proper coordination in a multi-process environment. The provided Python code offers a thoughtful solution to this classic problem, demonstrating effective synchronization mechanisms and visualization techniques.

In conclusion, the provided code not only presents an effective solution to the Dining Philosophers problem but also incorporates visualization tools that facilitate a deeper understanding of the complexities associated with concurrent resource sharing. We also gained a new perspective on the problem with the chandy/misra strategy.