

# Dining Philosophers Deadlock Problem

---

*Parallel Programming*

Duygu Kara - 190315066 / Dilara Ceyda Çetin - 200315079

23.12.2023

---

## Table of Contents

<b>Introduction To The Dining Philosophers Problem.....</b>	<b>2</b>
Rules for the problem include .....	2
<b>The strategy we used to solve the deadlock problem in this problem:</b>	
<b>“Resource Hierarchy” .....</b>	<b>3</b>
How does our chosen strategy of resource hierarchy solve the deadlock problem?.....	3
Advantages .....	4
Disadvantages.....	4
<b>Implementation of the problem with using resource hierarchy strategy.....</b>	<b>4</b>
<b>Output of Implementation.....</b>	<b>14</b>
<b>In Conclusion .....</b>	<b>16</b>
<b>References .....</b>	<b>16</b>

## Introduction To The Dining Philosophers Problem

The "Dining Philosophers" problem was introduced to the literature by Dijkstra and symbolizes concurrent process management. It is a synchronization problem in computer science, serving as an example to understand synchronization issues in parallel and distributed systems. The problem is defined as follows:

- A certain number of philosophers (usually five) are sitting at a table.
- Each philosopher has a plate and a fork in front of them.
- Philosophers can transition between two states: thinking and eating.
- Philosophers need two forks to eat, and the philosophers on both sides are required to share the same fork.

### Rules for the problem include:

- A philosopher must use the fork to the left and right to eat.
- A philosopher starts eating if they can use both forks.
- After eating, the philosopher must put the forks back and transition to thinking.
- If a philosopher is eating, the philosophers to the left and right cannot use the same fork.

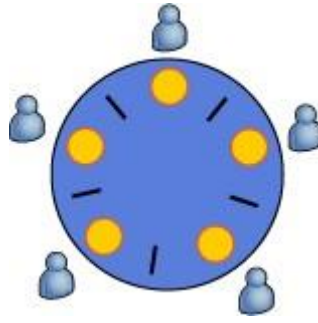
The goal is to regulate the sharing of forks among philosophers, creating a proper sequence for philosophers to eat. Numerous algorithms exist for solving this problem, with the most well-known using semaphore or mutex-like locking mechanisms to ensure synchronization.

In the scenario where all philosophers attempt to take the fork on their right, they will each have one fork, preventing any of them from eating. If they all try to grab both forks at the same time, a racing condition occurs, and the philosopher who acts first (with no guarantee) will be able to eat. If they all release the forks to let the other eat, none of them will be able to eat. These problems are often considered starvation issues, where a philosopher may go hungry, and there is no guarantee of anyone eating.

Another challenge in the problem is the possibility of deadlock. Due to a faulty design, a philosopher waiting for another to release a fork may lock the system. This represents the second risk in the problem.

Lastly, the no communication rule among philosophers is stated in the problem definition, but many solutions violate this rule. Philosophers communicate indirectly through the forks, signaling the availability of forks on their left or right.

Various algorithms have been developed to solve the problem, including resource hierarchy, random solution, odd-even strategy, conductor solution, Chandy-Misra solution, and the "Hold and Wait - Don't grab any chopsticks unless they are both available" approach. We can see a visual representation of the problem in figure 1.1.



**Figure 1.1**

## **The strategy we used to solve the deadlock problem in this problem:**

### **“Resource Hierarchy”**

As I mentioned earlier, the problem was designed to illustrate the challenges of avoiding deadlock, a system state where no progress is possible. In addition to deadlock, resource starvation can occur independently if a specific philosopher cannot obtain both forks due to a timing issue. For instance, there could be a rule where philosophers wait ten minutes for the other fork to become available before releasing the fork and waiting another ten minutes before attempting again. This scheme eliminates the possibility of deadlock (the system can always progress to a different state), but it still suffers from the liveness deadlock problem. If all five philosophers enter the dining room at the same time and each grabs the left fork simultaneously, they will wait ten minutes until all philosophers release their forks. Afterward, they will wait another ten minutes before grabbing the forks again. The failures that philosophers may experience resemble the challenges encountered in real computer programming where multiple programs require exclusive access to shared resources. However, the difficulties illustrated in the Dining Philosophers problem arise more frequently when multiple processes access updated datasets. Complex systems like operating system kernels rely on numerous locks and synchronization protocols to avoid issues such as deadlock, starvation, and data corruption.

### **How does our chosen strategy of resource hierarchy solve the deadlock problem?**

This solution to the problem is the one originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks they plan to use. The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest-numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks.

### Advantages:

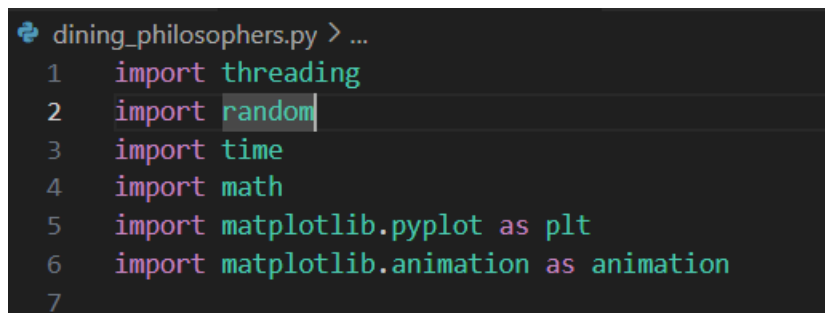
The resource hierarchy strategy reduces the possibility of deadlocks by limiting access to resources in a specific hierarchical order. The requirement for each philosopher to fork in turn prevents all philosophers from forking at the same time. Hierarchical sorting is a very simple and understandable strategy to implement. Having each philosopher fork in a particular order ensures that the system remains in a consistent state.

### Disadvantages:

While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose. The resource hierarchy solution is not fair. If philosopher 1 is slow to take a fork, and if philosopher 2 is quick to think and pick its forks back up, then philosopher 1 will never get to pick up both forks. A fair solution must guarantee that each philosopher will eventually eat, no matter how slowly that philosopher moves relative to the others.

## Implementation Of The Problem With Using Resource Hierarchy Strategy

We can see the import of the necessary libraries in figure 2.1.



```
dining_philosophers.py > ...
1  import threading
2  import random
3  import time
4  import math
5  import matplotlib.pyplot as plt
6  import matplotlib.animation as animation
7
```

Figure 2.1

The fork class is the class in which the fork is represented. We can see in figure 2.2. In this class, `__init__(self, index: int)`: This method is the initializer of the Fork class. It takes a parameter called `index` and initializes the properties of the fork. Each fork has a unique index, priority value (priority), a lock, a flag indicating whether the fork is currently picked up (`picked_up`), and an owner indicating who is using the fork.

`__enter__(self)`: This method can be used with the with statement. The `__enter__` method is called when you enter the with block and returns the self object. This allows the locking mechanism to be used within the with block.

`__call__(self, owner: int)`: This method is called within the with block to determine the owner of the fork. When called with the owner parameter, it unlocks the fork and determines its owner.

`__exit__(self, exc_type, exc_value, traceback)`: This method is called when exiting the with block. Releases the lock and resets the state of the fork.

`__str__(self)`: This method is used to represent the fork object as a string. Returns a string containing the index and owner of the fork.

This class is used to keep track of the status of a fork and regulate the use of the fork. Thanks to the use of the with statement, the fork is automatically locked and unlocked during use.

```
8 class Fork:
9     def __init__(self, index: int):
10         self.index: int = index
11         self.priority: int = index # Added priority based on the index
12         self.lock: threading.Lock = threading.Lock()
13         self.picked_up: bool = False
14         self.owner: int = -1
15
16     def __enter__(self):
17         return self
18
19     def __call__(self, owner: int):
20         if self.lock.acquire():
21             self.owner = owner
22             self.picked_up = True
23         return self
24
25     def __exit__(self, exc_type, exc_value, traceback):
26         self.lock.release()
27         self.picked_up = False
28         self.owner = -1
29
30     def __str__(self):
31         return f"F{self.index:2d} ({self.owner:2d})"
```

**Figure 2.2**

Then the Philosopher class is derived from the Thread class. This structure is used to support multithreading programming. Since the Philosopher class derives from the Thread class, each Philosopher instance represents a thread. This means that each philosopher can work independently and carry out his own operations at the same time. The Philosopher class has an `__init__` (initiator) method that determines the initial state. This initializer method is called when a philosopher object



is created and initializes the philosopher's properties ( fork status, number of spaghetti, etc.). So this method is called when a Philosopher object is created and determines the initial state of the philosopher.

The `super()` function calls the instance method of the Philosopher class's parent class (here `threading.Thread`). That is, it initializes the `__init__` method of the Thread class.

Thread class provides the basic functionality to create a thread.

```
self.index: int = index
```

The `self.index` property holds the index of the philosopher. Each philosopher has a unique index.

```
self.left_fork: Fork = min(left_fork, right_fork, key=lambda x: x.priority)
```

The `self.left_fork` property represents the left fork. When determining the left fork, the `min` function is used and the smallest one is selected according to the fork priorities.

```
self.right_fork: Fork = max(left_fork, right_fork, key=lambda x: x.priority)
```

The `self.right_fork` property represents the right fork. When determining the right fork, the `max` function is used and the larger one is selected according to the fork priorities.

```
self.spaghetti: int = spaghetti
```

The property `self.spaghetti` holds the amount of spaghetti the philosopher has. It is assumed that every philosopher had a certain amount of spaghetti to begin with.

```
self.eating: bool = False
```

The `self.eating` property indicates whether the philosopher is eating or not. Initially the philosopher is not at dinner, so it is set to `False`.

The `run` method represents the life cycle of a philosopher. This method continues the philosopher's thinking and eating until the amount of spaghetti is zero. This is a cycle in which the philosopher is constantly thinking and eating.

The `think` method simulates the philosopher's thinking action.

The `time.sleep` function pauses the process for a specified period of time (with a randomly selected time interval), which represents the philosopher's thinking process.

The `eat` method simulates the philosopher's eating action. It sorts the left and right forks as small and large, then locks and releases them respectively. It pauses the process for a certain period of time (again, with a randomly chosen time interval), which represents the philosopher's eating process. The expression `self.spaghetti -= 1` indicates that the philosopher ate one spaghetti and

reduced the amount of spaghetti by one. The `self.eating` property is a flag that indicates whether the philosopher is currently eating.

The `__str__` method returns a human-friendly string representation of a `Philosopher` instance. This string contains the philosopher's index and the amount of spaghetti he has. This information shows the philosopher's situation more meaningfully. In summary, the `philosopher` class is a class derived from the `Thread` class and is used to support multi-threaded programming. Each `Philosopher` instance represents a thread, which means that each `Philosopher` can run independently and perform its own operations simultaneously. The `init` method of the `Philosopher` class determines the initial state. This initializer method is called when a `Philosopher` object is created and initializes the philosopher's properties (fork state, spaghetti count, etc.). The `super()` function calls the instance method of the superclass of the `Philosopher` class (here `threading.Thread`). That is, it initializes the `init` method of the `Thread` class.

The main features of the `Philosopher` class are:

`self.index`: Keeps the philosopher's index. Each philosopher has a unique index.

`self.left_fork` and `self.right_fork`: Represents left and right fork. In fork selection, `min` and `max` functions are used according to fork priorities.

`self.spaghetti`: Holds the amount of spaghetti the philosopher has. It is initially assumed that every philosopher has a certain amount of spaghetti.

`self.eating`: A flag indicating whether the philosopher is eating or not. In the beginning, the philosopher is not at dinner (`False`).

The `run`, `think` and `eat` methods simulate the philosopher's life cycle and thinking/eating actions. Operations such as fork locking, lock release and time delay (`time.sleep`) are used in the thinking and eating processes. The `str` method provides a human-friendly string representation of a `Philosopher` instance. This structure allows each philosopher to work independently and organize fork sharing by prioritizing forks. We can see these implements in figure 2.3.



```

35     class Philosopher(threading.Thread):
36         def __init__(self, index: int, left_fork: Fork, right_fork: Fork, spaghetti: int):
37             super().__init__()
38             self.index: int = index
39             self.left_fork: Fork = min(left_fork, right_fork, key=lambda x: x.priority)
40             self.right_fork: Fork = max(left_fork, right_fork, key=lambda x: x.priority)
41             self.spaghetti: int = spaghetti
42             self.eating: bool = False
43
44         def run(self):
45             while self.spaghetti > 0:
46                 self.think()
47                 self.eat()
48
49         1 usage
50         def think(self):
51             time.sleep(3 + random.random() * 3)
52
53         1 usage
54         def eat(self):
55             small_fork, big_fork = sorted([self.left_fork, self.right_fork], key=lambda x: x.index)
56
57             with small_fork(self.index):
58                 with big_fork(self.index):
59                     time.sleep(5 + random.random() * 5)
60                     self.spaghetti -= 1
61                     self.eating = True
62                     time.sleep(5 + random.random() * 5)
63                     self.eating = False
64
65         def __str__(self):
66             return f"P{self.index:2d} ({self.spaghetti:2d})"

```

**Figure 2.3**

In this section, we create a graphical interface using the matplotlib library to visualize eating philosophers and forks. Here, taking a set of philosopher and fork objects, various matplotlib objects are created to represent them in a plot. `fig, ax = plt.subplots()`: A figure and an axis (axes) are created. A figure represents a drawing area that can contain one or more axes. The axis refers to the area where the drawings are made.

`ax.set_xlim(-1, 1)`, `ax.set_ylim(-1, 1)`: The limits of the X and Y axes are determined. In this case, the x and y axes are both set to be on a scale between -1 and 1.

`ax.set_aspect("equal")`: Keeps the ratio of the axis equal so circles are displayed as circles.

`ax.axis("off")`: Turns off the labels and lines of the axis, showing only the drawings.

`ax.set_title("Dining Philosophers")`: Sets the title of the chart. `philosopher_circles`: A list of circle objects representing philosophers is created. A circle is created for each philosopher, and initially the center of these circles is set to (0,0), their radius is 0.2, and their color is set to black.

`philosopher_texts`: A list of text objects containing the position numbers of philosophers is created. A text object is created for each philosopher and initially these texts are filled with position numbers, with the position as (0,0). Texts are set to center aligned and vertically aligned. `fork_lines`:

A list of line objects representing forks is created. A line is created for each fork and initially the start and end points of these lines are set as (0,0) and its color is set to black. `fork_texts`: A list of text objects containing fork indices is created. For each fork, a text object is created and initially these texts are filled with fork indices, with position (0,0). Texts are set to center aligned and vertically aligned. The created circles, lines, texts and fork index texts are added to the axis. This allows the creation of a chart that visually represents philosophers, forks, and fork indices. The purpose of this visualization is to make it possible to follow the status of philosophers and forks live. We can see these implementations on figure 2.4 and figure 2.5.

```

77 def animated_table(philosophers: list[Philosopher], forks: list[Fork], m: int):
78     fig, ax = plt.subplots()
79     ax.set_xlim(-1, 1)
80     ax.set_ylim(-1, 1)
81     ax.set_aspect("equal")
82     ax.axis("off")
83     ax.set_title("Dining Philosophers")
84     philosopher_circles: list[plt.Circle] = [
85         plt.Circle((0, 0), 0.2, color="black") for _ in range(len(philosophers))
86     ]
87     philosopher_texts: list[plt.Text] = [
88         plt.Text(
89             0,
90             0,
91             str(philosopher.position + 1),
92             horizontalalignment="center",
93             verticalalignment="center",
94         )
95         for philosopher in philosophers
96     ]
97     fork_lines: list[plt.Line2D] = [
98         plt.Line2D((0, 0), (0, 0), color="black") for _ in range(len(forks))
99     ]
100     fork_texts: list[plt.Text] = [
101         plt.Text(
102             0,
103             0,
104             str(fork.index),
105             horizontalalignment="center",
106             verticalalignment="center",
107         )
108         for fork in forks
109     ]

```

**Figure 2.4**

```

110     for philosopher_circle in philosopher_circles:
111         ax.add_patch(philosopher_circle)
112     for fork_line in fork_lines:
113         ax.add_line(fork_line)
114     for philosopher_text in philosopher_texts:
115         ax.add_artist(philosopher_text)
116     for fork_text in fork_texts:
117         ax.add_artist(fork_text)
118

```

**Figure 2.5**

It allows updating the philosophers and fork indexes in the graph using the animation module of the matplotlib library. The update function contains the updates to be performed at each animation step (frame). with `forks[i](i)::` When receiving the philosopher's fork, the fork is unlocked and the philosopher is assigned the position of owner of the fork. This action visually represents the receipt of the fork.

`philosopher_circles[i].center:` Updates the center of the philosopher's circle. This represents the philosopher's new position.

`philosopher_texts[i].set_position:` Updates the position of the philosopher's position text.

`philosopher_texts[i].set_text:` Updates the philosopher's position text. If the philosopher's spaghetti count is 0, it is shown as "X".

`philosopher_circles[i].set_color:` Updates the color of the circle depending on whether the philosopher has eaten or not.

`philosopher_circles[i].radius:` Updates the radius of the circle based on the philosopher's spaghetti count.

`fork_lines[i].set_data:` Updates the position of the fork's line.

`fork_texts[i].set_position:` Updates the position of the fork's index text.

`fork_texts[i].set_text:` Updates the index text of the fork. Finally, an animation is created by calling this update function at a certain interval with `animation.FuncAnimation` and this animation is shown on the screen with `plt.show()`. Animation is a loop in which philosophers, fork indices, circles and lines are updated over a period of time. We can see in these implementations on figure 2.6 and figure 2.7.

```

119 def update(frame):
120     nonlocal philosophers, forks
121     for i in range(len(philosophers)):
122         with forks[i](i): # Use the Fork context manager for visualization
123             pass
124         philosopher_circles[i].center = (
125             0.5 * math.cos(2 * math.pi * i / len(philosophers)),
126             0.5 * math.sin(2 * math.pi * i / len(philosophers)),
127         )
128         philosopher_texts[i].set_position(
129             (
130                 0.9 * math.cos(2 * math.pi * i / len(philosophers)),
131                 0.9 * math.sin(2 * math.pi * i / len(philosophers)),
132             )
133         )
134         philosopher_texts[i].set_text(
135             str(philosophers[i]) if philosophers[i].spaghetti > 0 else "x"
136         )
137         if philosophers[i].eating:
138             philosopher_circles[i].set_color("red")
139         else:
140             philosopher_circles[i].set_color("black")
141         philosopher_circles[i].radius = 0.2 * philosophers[i].spaghetti / m
142         fork_lines[i].set_data(
143             (
144                 0.5 * math.cos(2 * math.pi * i / len(philosophers)),
145                 0.5 * math.cos(2 * math.pi * (i + 1) / len(philosophers)),
146             ),
147             (
148                 0.5 * math.sin(2 * math.pi * i / len(philosophers)),
149                 0.5 * math.sin(2 * math.pi * (i + 1) / len(philosophers)),
150             ),
151         )

```

Figure 2.6

```

152         fork_texts[i].set_position(
153             (
154                 0.5 * math.cos(2 * math.pi * i / len(philosophers))
155                 + 0.5 * math.cos(2 * math.pi * (i + 1) / len(philosophers)),
156                 0.5 * math.sin(2 * math.pi * i / len(philosophers))
157                 + 0.5 * math.sin(2 * math.pi * (i + 1) / len(philosophers)),
158             )
159         )
160         fork_texts[i].set_text(str(forks[i]))
161     return philosopher_circles + fork_lines + philosopher_texts + fork_texts
162
163 ani = animation.FuncAnimation(
164     fig, update, frames=range(100000), interval=10, blit=False
165 )
166 plt.show()
167
168

```

Figure 2.7

In this part, the function represents a loop in which philosophers eating, fork indexes and general philosopher situations are shown live on the screen.

while sum(philosopher.spaghetti for philosopher in philosophers) > 0:: The cycle continues as long as the sum of all philosophers' remaining spaghetti numbers is greater than 0. So, the cycle ends when any philosopher runs out of spaghetti.

eating\_philosophers: int = sum(philosopher.eating for philosopher in philosophers): Calculates how many philosophers are currently eating.

print("\033[H\033[J"): Clears the console screen. This prevents updated information from overlapping by clearing the screen with each iteration.

print("=" \* (len(philosophers) \* 16)): Creates a title line, causing a separator to appear on the screen.

"E" if philosopher.eating else "T" for philosopher in philosophers: Prints the character "E" or "T" on the screen depending on the eating situation of each philosopher. These states designate philosophers who eat as "E" and philosophers who think as "T".

print(" ".join(map(str, forks)), " ", forks[0]): Prints the fork indexes and the first fork to the screen. Formats with spaces between fork indexes.

print(" ", " ".join(map(str, philosophers)), " : ", str(eating\_philosophers)): Prints the philosophers' positions and meal statuses on the screen. Formats with gaps between philosophers' positions.

time.sleep(0.1): Waits for a short time. This ensures that the screen updates quickly and is readable.

This loop constantly updates the status of the philosophers and fork indexes and provides a live display on the screen. The eating state is denoted as "E" and the thinking state is denoted as "T". Additionally, each philosopher's position and eating status are indicated on the screen. We can see in these implementations on figure 2.8.

```
169 def table(philosophers: list[Philosopher], forks: list[Fork], m: int):
170     while sum(philosopher.spaghetti for philosopher in philosophers) > 0:
171         eating_philosophers: int = sum(
172             philosopher.eating for philosopher in philosophers
173         )
174         print("\033[H\033[J")
175         print("=" * (len(philosophers) * 16))
176         print(
177             "
178             " ".join(
179                 ["E" if philosopher.eating else "T" for philosopher in philosophers]
180             ),
181         )
182         print(" ".join(map(str, forks)), " ", forks[0])
183         print(
184             "
185             " ".join(map(str, philosophers)),
186             " : ",
187             str(eating_philosophers),
188         )
189         time.sleep(0.1)
190
```

**Figure 2.8**

Finally, the main function is a main program where philosophers and forks are created, philosopher threads are started, the table view and animation are displayed. Additionally, it waits for all philosopher threads to complete.

```
n: int = 5 and m: int = 7:
```

n represents the number of philosophers, and m represents the amount of spaghetti each philosopher initially had.

```
forks: list[Fork] = [Fork(i) for i in range(n)]:
```

Forks creates a list containing n forks. Each fork is a Fork instance and its indices are numbers from 0 to n-1.

```
philosophers: list[Philosopher] = [...]:
```

philosophers is a list of examples of Philosophers, each representing a philosopher.

Each philosopher's left fork, right fork, and initial amount of spaghetti are determined. The left fork takes the current index in the fork list, and the right fork takes the index determined by the formula  $(i + 1) \% n$ . This allows forks to be shared in a circular manner.

```
for philosophers in philosophers: philosopher.start():
```

Each created philosopher thread is started (the start method is called).

Each philosopher will perform his own operations (thinking and eating).

```
threading.Thread(target=table, args=(philosophers, forks, m), daemon=True).start():
```

A background thread is started (with daemon=True). This thread routinely prints the status of philosophers and forks by calling the table function. The table function clears the screen at regular intervals in a loop and shows the philosophers' status.

```
animated_table(philosophers, forks, m):
```

The animated\_table function is called to show the animated table view. This function provides an animated graphical representation of philosophers and forks.

```
for philosopher in philosophers: philosopher.join():
```

The join method is used to wait for all philosopher threads to complete. This allows the main thread to wait for other threads to complete.

```
if __name__ == "__main__":
```

This check allows the Python interpreter to check whether a file is being executed directly or is a module. If this file is executed directly (that is, it is not a module), it calls the main function and initializes the main logic of the program. We can see in these implementations on figure 2.9.



```

210
211 def main() -> None:
212     n: int = 5
213     m: int = 7
214     forks: list[Fork] = [Fork(i) for i in range(n)]
215     philosophers: list[Philosopher] = [
216         Philosopher(i, forks[i], forks[(i + 1) % n], m) for i in range(n)
217     ]
218     for philosopher in philosophers:
219         philosopher.start()
220     threading.Thread(target=table, args=(philosophers, forks, m), daemon=True).start()
221     animated_table(philosophers, forks, m)
222     for philosopher in philosophers:
223         philosopher.join()
224
225 if __name__ == "__main__":
226     main()

```

Figure 2.9

## Output of Implementation

According to this implementation, we show some of the output in figure 3.1, figure 3.2, figure 3.3 and figure 3.4. Since it is very long, we added a part of the beginning to this file.

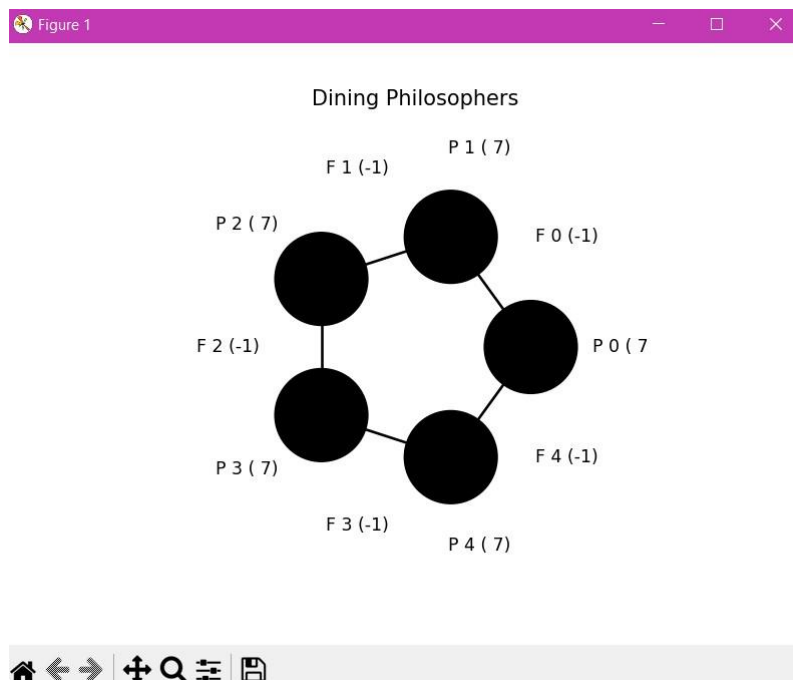
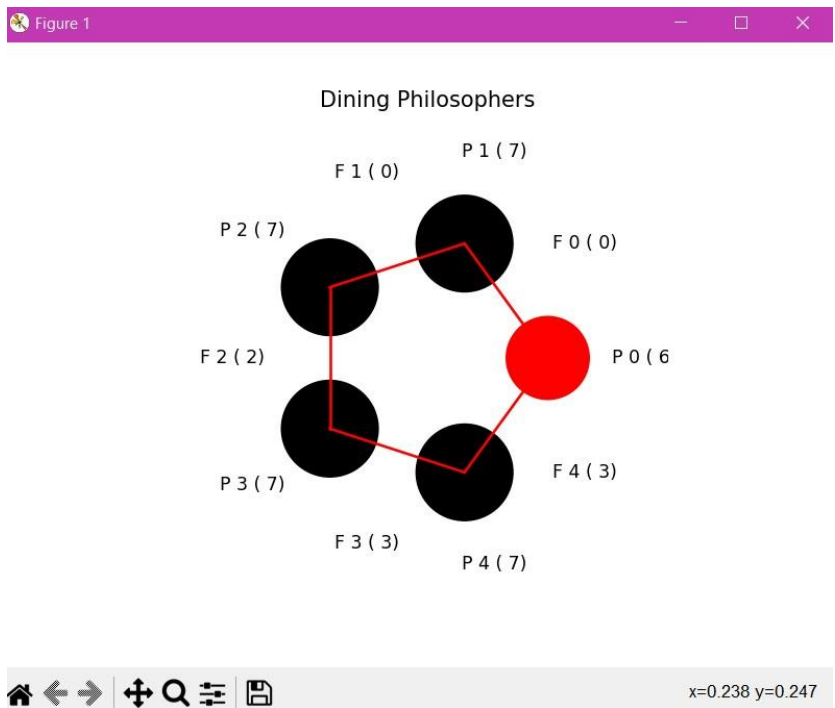
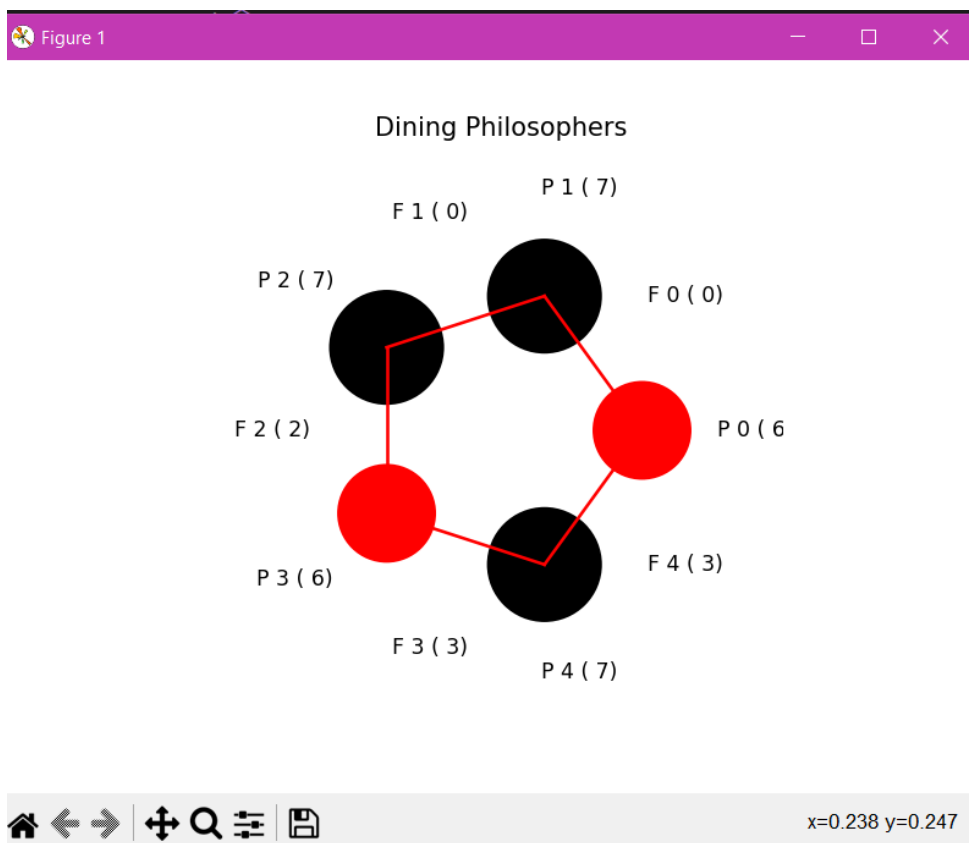


Figure 3.1

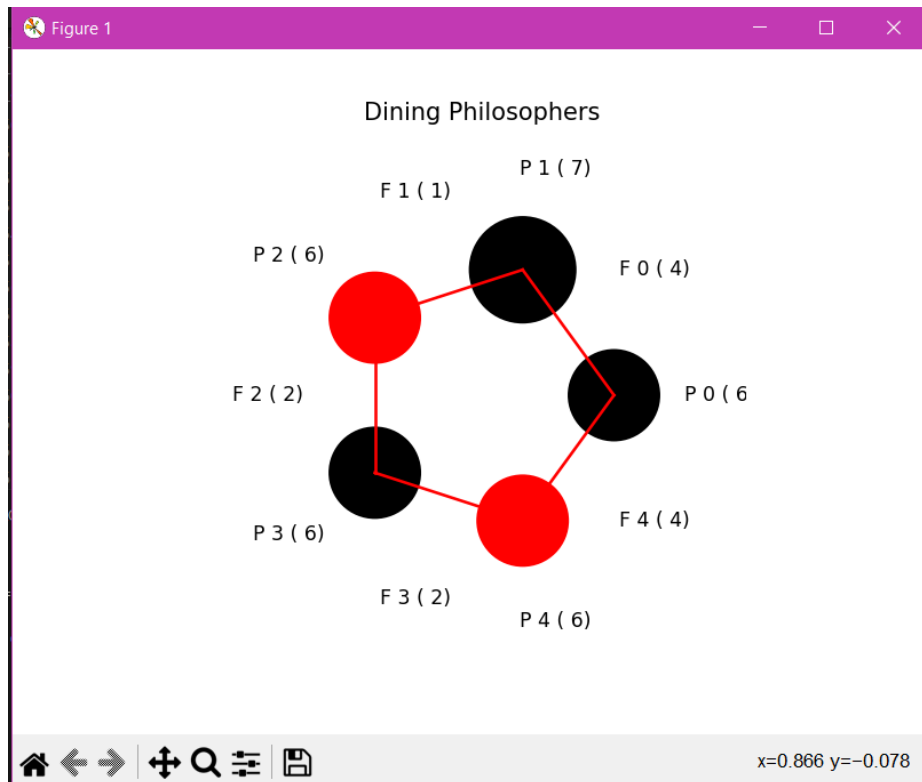




**Figure 3.2**



**Figure 3.3**



**Figure 3.4**

## In C onclusion

As a result, a philosopher's left and right forks are determined by their order of priority. Thus, fork locks cannot be captured by two different philosophers at the same time. Fork locks are made in order of priority. Min and max functions are used when determining the left and right fork.

## References

<https://bilgisayarkavramlari.com/2012/01/22/filozoflarin-aksam-yemegi-dining-philosophers/>

[https://eng.libretexts.org/Courses/Delta\\_College/Operating\\_System%3A\\_The\\_Basics/06%3A\\_Deaddlock/6.4%3A\\_Dining\\_Philosopher\\_Problem](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/06%3A_Deaddlock/6.4%3A_Dining_Philosopher_Problem)