**PARALLEL PROGRAMMING**

**HOME WORK**

**GROUP X with RESOURCE HIERARCHY SOLUTION**

| **Ad Soyad** | **Numara** |
| --- | --- |
| Ali Duman | 200315036 |
| Dilara Akdeniz | 200315067 |
| Esra Zeynep Başoğlu | 200315058 |
| Małgorzata Lura | 230315094 |

[**1. WHAT IS DEADLOCK? 2**](#_czxyj41dxbf7)

[**2. DEADLOCK PROBLEM OF DINING PHILOSOPHERS 3**](#_udalpvmvpxvd)

[**3. RESOURCE HIERARCHY SOLUTION 4**](#_1t92rgiw4tx)

[**4. FIXING DEADLOCK PROBLEM 5**](#_1ixp0ffsuau8)

[**5. Full Code and Explanation of Code 7**](#_fiecomiwm1my)

[**6. Working Of Code 12**](#_3c4dc8vcx9ux)

# **WHAT IS DEADLOCK?**

Deadlock, in computer science, refers to a situation where multiple processes or threads are waiting for each other to release resources, resulting in none of the processes being able to proceed. In a deadlock situation, all involved processes are perpetually waiting without the ability to access the resources they need. The occurrence of deadlocks typically requires the simultaneous presence of four conditions:

***1. Mutual Exclusion:*** Resources are non-shareable and can only be used by one process at a time.

***2. Hold and Wait:*** A process can hold onto resources it already has while waiting for additional resources.

***3. Circular Wait:*** There is a cycle of processes where each process is waiting for a resource held by the next process in the cycle.

***4. No Preemption:*** A resource can only be released voluntarily by the process holding it; it cannot be forcibly taken away by another process.

For example, consider two processes, A and B. If A is waiting for a resource that B holds, and B is waiting for a resource that A holds, and these resources are subject to mutual exclusion, this leads to a deadlock. Both processes cannot proceed as they are each waiting for the other to release the resource.

Various methods exist to prevent or resolve deadlocks, including allocating resources in a specific order, careful management of resource requests, and forcibly reclaiming resources (preemption). Classic problems like the Dining Philosophers problem are often used to illustrate how deadlocks can occur and how they can be resolved.

# **DEADLOCK PROBLEM OF DINING PHILOSOPHERS**

The Dining Philosophers problem is a classic synchronization issue in computer science that illustrates how a deadlock can occur and the challenges involved in resolving it. The problem was formulated by E. W. Dijkstra in 1965 and is stated as follows:

- Problem Setup: Imagine five philosophers who spend their lives just thinking and eating. They sit at a round table with five chairs. The table has five forks (or chopsticks), one between each pair of adjacent philosophers.

- Rules: Each philosopher thinks for a while, then gets hungry and tries to eat. To eat, a philosopher requires both forks to their left and right. After eating, they put down both forks and start thinking again.

- Challenge: The challenge is to write a set of rules (a protocol) for the philosophers that will ensure no philosopher will starve; that is, each can eventually eat.

**Deadlock in this Context**

A deadlock can easily occur in this scenario:

- Imagine each philosopher picks up the fork on their left at the same time.

- Then, each of them tries to pick up the fork on their right, which is already held by another philosopher.

- None of the philosophers can eat because each is waiting for a fork that is held by someone else.

- As a result, all philosophers are stuck waiting indefinitely, leading to a deadlock.

**Conditions for Deadlock**

This problem neatly illustrates the four conditions necessary for a deadlock to occur:

***1. Mutual Exclusion:*** Each fork can be held by only one philosopher at a time.

***2. Hold and Wait:*** Each philosopher may hold one fork (the left one) and wait for the other (the right one).

***3. No Preemption:*** A fork can only be released voluntarily by the philosopher holding it. It cannot be forcibly taken away from them.

***4. Circular Wait:*** There exists a set of waiting processes (philosophers), each of which is waiting for a resource (fork) held by another process in the set.

**SOLUTIONS**

To resolve this deadlock issue, several strategies can be employed:

- Resource Hierarchy Solution: Assign an order to resources (forks) and ensure philosophers pick them up in a specified order to prevent a circular wait condition.

- Arbiter Solution: Introduce a waiter or arbiter to control when philosophers can attempt to pick up forks.

- Chandy/Misra Solution: Philosophers request forks from their neighbors and use a system of messaging to avoid deadlock.

- Time-Outs: Philosophers put down a fork if they cannot acquire both within a certain timeframe.

Each solution aims to break one of the deadlock conditions, typically the circular wait, to ensure that the system can continue functioning without any process (philosopher) being indefinitely blocked.

# **RESOURCE HIERARCHY SOLUTION**

The Resource Hierarchy Solution is a method used to solve situations at risk of deadlock, such as the Dining Philosophers problem. This solution involves assigning a hierarchy or ordered arrangement to resources (in this case, forks) to prevent deadlock. Here are the key principles of this approach:

**Fundamental Principles**

***1. Assigning Order to Resources:*** First, an ordered arrangement is assigned to the resources (forks). For example, forks can be numbered from 1 to 5.

***2. Ordered Resource Request:*** Each philosopher must acquire forks according to this order. That is, a philosopher should pick up the lower-numbered fork first and then the higher-numbered one. This prevents all philosophers from trying to reach for the same forks simultaneously.

***3. Preventing Circular Wait:*** This approach avoids a circular waiting condition because each philosopher attempts to pick up the forks in a different order. Hence, it prevents any philosopher from waiting for both the fork to their left and right at the same time.

**Implementation**

To apply this solution to the Dining Philosophers problem, assume that a philosopher must not pick up the fork to their right before picking up the one to their left. For example, the first philosopher picks up Fork 1 (on their left) first, then Fork 2 (on their right). However, the last philosopher (Philosopher 5) must first pick up Fork 5 and then reach for Fork 1.

**Advantages and Disadvantages**

- Advantages: The Resource Hierarchy Solution is simple and straightforward to implement. The ordered use of resources prevents the occurrence of circular wait conditions.

- Disadvantages: This solution can lead to delays in resource utilization in some cases. For instance, the chance for a philosopher to use both forks simultaneously decreases, which can slow down the eating process.

In conclusion, the Resource Hierarchy Solution offers an effective method for solving deadlock problems, but it may not always be the most efficient approach. Especially in systems where resource utilization and efficiency are important, potential delays introduced by this method should be considered.

# **FIXING DEADLOCK PROBLEM**

Let's look at a more detailed explanation of the code, specifically focusing on the 'eating' method of the 'Philosopher' class, as this is where the Resource Hierarchy Solution is implemented.

**`Philosopher` Class's `eat` Method**

This method simulates the eating behavior of philosophers. It's crucial that the philosophers pick up the forks in a specific order to prevent deadlocks. Here's a detailed breakdown of the `eat` method:

| def eat(self):  first\_fork, second\_fork = sorted([self.left\_fork, self.right\_fork], key=lambda fork: fork.index)    with first\_fork(self.index), second\_fork(self.index):  print(f"Philosopher {self.index} starts eating.")  time.sleep(random.uniform(1, 2))  print(f"Philosopher {self.index} finishes eating.")  self.spaghetti -= 1 |
| --- |

**Sorting the Forks**

The `first\_fork` and `second\_fork` variables represent the sorted forks. The `sorted` function arranges the forks (`self.left\_fork` and `self.right\_fork`) based on the `index` attribute of the `Fork` objects. This ensures that each philosopher picks up the lowest-numbered fork first.

**Context Manager (`with` Block)**

The `with` block utilizes Python's context managers. This block ensures that the forks are safely acquired and released. `first\_fork(self.index)` and `second\_fork(self.index)` statements acquire the locks of the respective forks. Within this block, the philosopher eats (simulated by `time.sleep`) and upon exiting the block, the locks on the forks are automatically released.

**Eating and Decreasing Spaghetti**

When the philosopher starts eating, a message is printed, followed by `time.sleep(random.uniform(1, 2))` function to simulate the duration of eating. After eating, the philosopher prints a message indicating they have finished eating and decrements the `self.spaghetti` value by one, representing the remaining number of meals.

**Summary**

The implementation of this `eat` method reflects the core principles of the Resource Hierarchy Solution. By picking up the forks in a specific order, each philosopher prevents deadlocks. This approach is effective in solving concurrency issues like the Dining Philosophers problem and simply avoids the formation of deadlocks.

# **Full Code and Explanation of Code**

Here is the complete and modified code for the Dining Philosophers problem using the Resource Hierarchy Solution:

| import threading  import random  import time  import math  import matplotlib.pyplot as plt  import matplotlib.animation as animation  class Fork:  def \_\_init\_\_(self, index: int):  self.index: int = index  self.lock: threading.Lock = threading.Lock()  self.picked\_up: bool = False  self.owner: int = -1  def \_\_enter\_\_(self):  return self  def \_\_call\_\_(self, owner: int):  if self.lock.acquire():  self.owner = owner  self.picked\_up = True  return self  def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  self.lock.release()  self.picked\_up = False  self.owner = -1  def \_\_str\_\_(self):  return f"F{self.index:2d} ({self.owner:2d})"  class Philosopher(threading.Thread):  def \_\_init\_\_(self, index: int, left\_fork: Fork, right\_fork: Fork, spaghetti: int):  super().\_\_init\_\_()  self.index: int = index  self.left\_fork: Fork = left\_fork  self.right\_fork: Fork = right\_fork  self.spaghetti: int = spaghetti  def run(self):  while self.spaghetti > 0:  self.think()  self.eat()  def think(self):  update\_visual(self.index, 'thinking', self.spaghetti)  print(f"Philosopher {self.index} is thinking.")  time.sleep(random.uniform(1, 3))  def eat(self):  first\_fork, second\_fork = sorted([self.left\_fork, self.right\_fork], key=lambda fork: fork.index)  with first\_fork(self.index), second\_fork(self.index):  update\_visual(self.index, 'eating', self.spaghetti)  print(f"Philosopher {self.index} starts eating.")  time.sleep(random.uniform(1, 2))  print(f"Philosopher {self.index} finishes eating.")  self.spaghetti -= 1  update\_visual(self.index, 'thinking', self.spaghetti)  # Görselleştirme için yeni fonksiyonlar  states = ['thinking'] \* 5  spaghetti\_remaining = [3] \* 5 # Başlangıçta her filozof için 3 spaghetti  positions = [(0.5 \* math.cos(2 \* math.pi \* i / 5), 0.5 \* math.sin(2 \* math.pi \* i / 5)) for i in range(5)]  fig, ax = plt.subplots()  def update\_visual(philosopher, status, spaghetti):  global states, spaghetti\_remaining  states[philosopher] = status  spaghetti\_remaining[philosopher] = spaghetti  def draw\_circle(ax, pos, radius, color):  circle = plt.Circle(pos, radius, color=color)  ax.add\_artist(circle)  def animate(i):  ax.clear()  ax.set\_xlim(-1.5, 1.5)  ax.set\_ylim(-1.5, 1.5)  ax.set\_aspect('equal', 'box')  ax.axis('off')  for i, (state, spaghetti) in enumerate(zip(states, spaghetti\_remaining)):  color = 'yellow' if state == 'eating' else 'gray'  radius = 0.1 \* spaghetti # Yemek miktarına göre dairenin büyüklüğü  draw\_circle(ax, positions[i], radius, color)  ax.text(positions[i][0], positions[i][1], str(i), ha='center', va='center')  def main():  forks = [Fork(i) for i in range(5)]  philosophers = [Philosopher(i, forks[i], forks[(i + 1) % 5], 3) for i in range(5)]  for philosopher in philosophers:  philosopher.start()  ani = animation.FuncAnimation(fig, animate, interval=1000)  plt.show()  for philosopher in philosophers:  philosopher.join()  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

**Explanation of Codes**

1. **Fork Class**

-The Fork class represents a fork.

-The **\_*\_init*\_\_** method sets the fork's index and initializes a **threading.Lock** object. The **picked\_up** and **owner** variables track the status and the owner of the fork.

-The **\_*\_enter*\_\_** method is called when the context manager starts and returns the fork itself.

-The **\_*\_call*\_\_** method is invoked when a philosopher attempts to use the fork. If the lock can be acquired, it updates **owner** and sets **picked\_up** to **True**.

-The **\_*\_exit*\_\_** method is called when the context manager ends. It releases the lock of the fork and resets its status.

-The **\_*\_str*\_\_** method creates a string representation of the fork, typically used for debugging.

1. **Philosopher Class**

-The Philosopher class represents a philosopher and is derived from **threading.Thread.**

-The **\_*\_init*\_\_** method sets the philosopher's index, left and right forks, and the amount of spaghetti they can eat.

- The **run** method defines the main loop of the philosopher. It alternates between thinking and eating as long as there is spaghetti left.

-The **think** method simulates the philosopher thinking. It calls **update\_visual** for visualization and waits for a random duration.

-The **eat** method simulates the process of the philosopher eating. Forks are picked up and released in order. The philosopher's eating process is visualized, and the spaghetti count is decreased.

**->Sorting Forks**

* This line sorts the philosopher's left and right forks based on their indices. This is done to prevent deadlock. Philosophers always pick up the lowest indexed fork first, ensuring that all philosophers follow the same order for picking and putting down forks.

**->Acquiring Forks and Eating**

* The **with** statement is used to manage the locks of the forks via Python's context manager protocol. Within this block, the philosopher first acquires **first\_fork** and then **second\_fork**. If both fork locks are successfully acquired, the code within the block is executed. When the block ends (either after the block of code is completed or an exception occurs), the **\_*\_exit*\_\_** methods are called, and the fork locks are released.

**->Visualization and Eating Process**

* The **update\_visual** function is called to update the philosopher's state to 'eating', which is used for visualization.
* The philosopher's start of eating is printed to the console.
* **time.sleep(random.uniform(1, 2)):** The philosopher eats for a random duration between 1 and 2 seconds, simulating real-time action.

**->Updating the Amount of Spaghetti**

* The amount of spaghetti the philosopher has eaten is decreased by one.
* The philosopher's state is updated to 'thinking', and the remaining amount of spaghetti is sent for visualization again.

**3. Visualization Function**

-The **states**, **spaghetti\_remaining,** and **positions** lists hold the states of the philosophers, the remaining spaghetti, and their positions in the visualization.

-**fig** and **ax** are figure and axes objects needed for Matplotlib visualization.

-The **update\_visual** function updates a specific philosopher's state and remaining spaghetti.

-The **draw\_circle** function draws a circle at a given position, radius, and color.

-The **animate** function is called at each step of the animation and updates the visual representations of the philosophers. The size of the circles is adjusted according to the remaining spaghetti.

**4. Main Function**

-The **main** function creates forks and philosophers, starts the philosophers, and initiates the Matplotlib animation. It waits until all philosophers have finished.

**SUMMARY**

This code ensures parallel execution of each philosopher as a thread. Philosophers alternate between thinking and eating actions, and this process is visualized. The code provides a solution to the dining philosophers problem and a visual representation of it.

# **Working Of Code**



