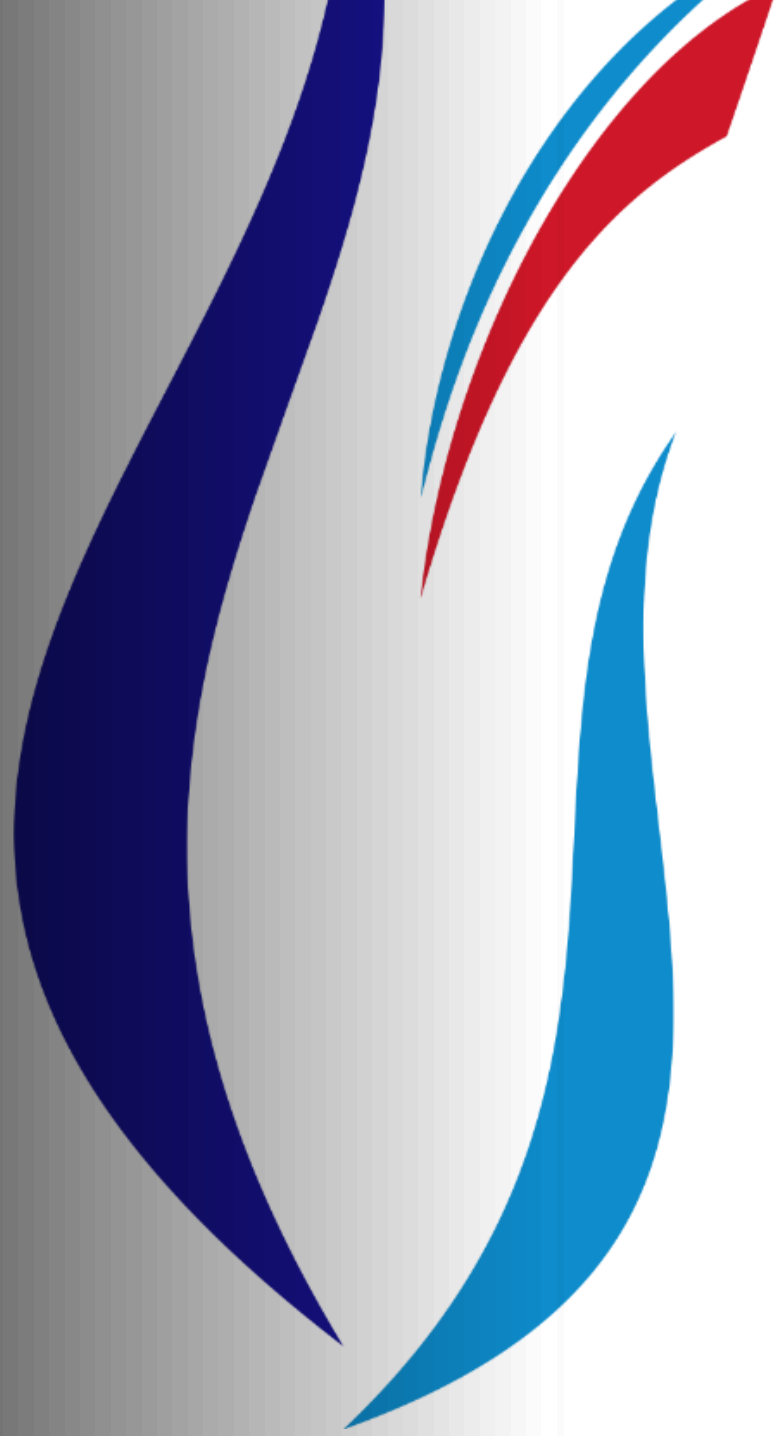
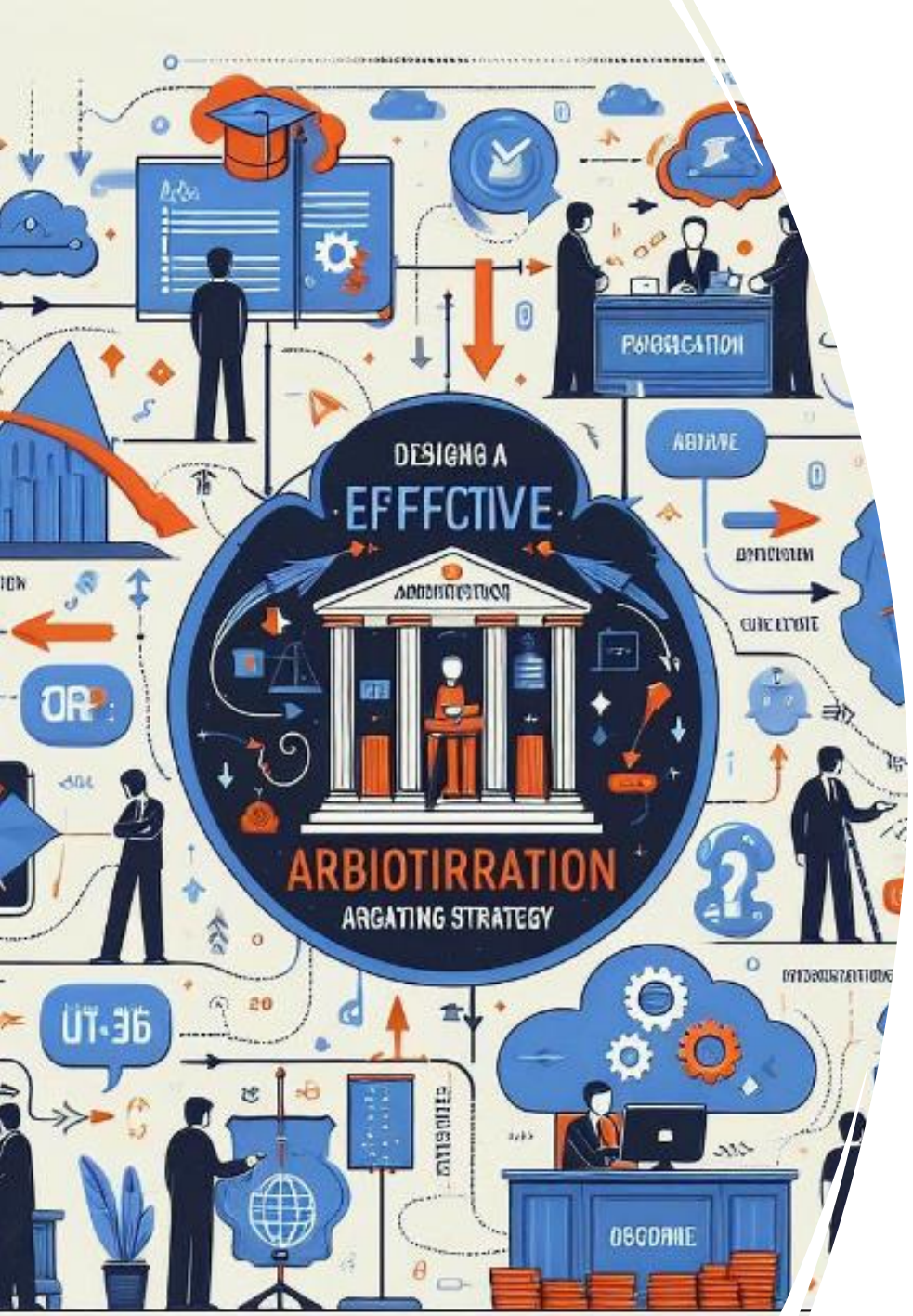
PARALEL PROGRAMMING

Centilmenler grubu proje sunumu





What is arbitrator solution?

- The term "Arbitrator solution" refers to the process of making an appropriate decision between various events or demands that usually occur in a system or application. This type of solution is used to resolve conflicts between different components or modules and to manage resources effectively.Arbitration solution can be prominent in the following situations and usage scenarios:Management of Competing Demands:When multiple components or modules in a system try to use the same resource (database, network connection, etc.), there may be conflicts between these demands. An arbitration solution can handle such situations and decide which demand takes priority.Timing and Sequence Control:If there are operations that need to be performed simultaneously or sequentially, the arbitration solution can ensure that these operations are scheduled appropriately. It can be especially useful in multitasking systems or parallel programming.Resource Sharing:If there are multiple components in the system that share resources (memory, processor time, file access, etc.), an arbitration solution can be applied to share these resources fairly.

```python
class Arbitrator:
    def __init__(self, max_allowed: int):
        self.max_allowed = max_allowed
        self.lock = threading.Lock()
        self.condition = threading.Condition(self.lock)
        self.allowed = 0
```

- **max_allowed Parameter:**
  - The **__init__** method takes an integer parameter **max_allowed**, which represents the maximum number of philosophers allowed to eat simultaneously. This parameter determines how many philosophers can acquire permission from the arbitrator at the same time.
- **Lock and Condition Variables:**
  - **self.lock**: This is an instance of **threading.Lock** and is used to provide mutual exclusion when accessing shared data. It ensures that only one thread can acquire the lock at a time, preventing race conditions.
  - **self.condition**: This is an instance of **threading.Condition** associated with the lock. It is used to coordinate communication between threads. Specifically, it allows threads to wait until a certain condition is met and notifies other threads when the condition changes.
- **allowed Attribute:**
  - **self.allowed**: This attribute keeps track of the number of philosophers currently allowed to eat. Initially set to 0, it will be incremented when a philosopher acquires permission and decremented when a philosopher releases permission.

- The **acquire** method is responsible for allowing a philosopher to acquire permission to eat.
- It uses the **with self.lock** statement to acquire the lock associated with the arbitrator. This ensures that only one thread (philosopher) can execute the critical section of code at a time.
- The method enters a **while** loop that checks whether the number of currently allowed philosophers (**self.allowed**) is less than the maximum allowed (**self.max_allowed**). If the condition is not met, it means the maximum number of philosophers are currently eating, so the current thread (philosopher) needs to wait.
- Inside the loop, the **self.condition.wait()** statement is used to release the lock and put the current thread to sleep until another thread notifies it. This is an efficient way to avoid busy-waiting.
- When the condition becomes true (i.e., the number of allowed philosophers is less than the maximum allowed), the loop exits, and the philosopher increments the **self.allowed** count, indicating that it has acquired permission to eat.

```
# It represents permission
1 usage
def acquire(self):
    with self.lock:
        while self.allowed >= self.max_allowed:
            self.condition.wait()
        self.allowed += 1
```

```
1 usage
def release(self):
    with self.lock:
        self.allowed -= 1
        self.condition.notify()
```

- The **release** method is responsible for releasing the permission acquired by a philosopher after finishing eating.
- Similar to **acquire**, it uses the **with self.lock** statement to acquire the lock associated with the arbitrator, ensuring exclusive access to the shared data.
- It decrements the **self.allowed** count, indicating that a philosopher has finished eating and released permission.
- The **self.condition.notify()** statement is then used to notify one waiting thread (philosopher) that the condition may have changed. This is important to wake up a waiting philosopher in the **acquire** method.