OS – Common Task

1.Process Management:

Waiting Time (WT):

Definition: Waiting time refers to the total time a process spends waiting in the ready queue before it gets a chance to execute on the CPU.

Calculation: It is the difference between the time a process arrives (enters the system) and the time it starts executing.

Importance: Waiting time impacts system performance. Minimizing waiting time ensures efficient utilization of resources.

Turnaround Time (TAT):

Definition: Turnaround time represents the total time taken by a process from submission (arrival) to its completion (termination).

Calculation: It is the sum of waiting time and execution time (burst time):TAT=Waiting Time+Burst Time

Importance: Turnaround time reflects the overall efficiency of the system. Lower turnaround time indicates faster job completion.

In summary, waiting time focuses on the time spent waiting in the queue, while turnaround time considers the entire lifecycle of a process.

Python code:

```
def calculate_fifo(processes):
    n = len(processes)
    wt = [0] * n
    tat = [0] * n

    # Calculate waiting time
    for i in range(1, n):
        wt[i] = processes[i - 1]["bt"] + wt[i - 1]
```

```
    # Calculate turnaround time
    for i in range(n):
        tat[i] = processes[i]["bt"] + wt[i]

    # Calculate averages
    avg_wt = sum(wt) / n
    avg_tat = sum(tat) / n

    return avg_wt, avg_tat

# Example usage
if _name_ == "_main_":
    processes = [
        {"at": 0, "bt": 10},
        {"at": 0, "bt": 5},
        {"at": 0, "bt": 8},
    ]
    avg_waiting_time, avg_turnaround_time = calculate_fifo(processes)
    print(f"Average Waiting Time: {avg_waiting_time:.2f}")
    print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")
```

2.Deadlock Handling:

Python code that demonstrates deadlock detection using a simple resource allocation graph (RAG)

```
class Resource:
    def _init_(self, name):
        self.name = name
        self.available = 0
        self.max_instances = 0
```

```
class Process:
    def _init_(self, name):
        self.name = name
        self.allocated = {}
        self.requested = {}

def detect_deadlock(resources, processes):
    # Initialize the resource allocation graph (RAG)
    for p in processes:
        for r in resources:
            p.allocated[r.name] = 0
            p.requested[r.name] = 0

    # Simulate resource requests and allocations (you'd replace this with actual logic)
    # For demonstration purposes, let's assume Process P1 requests R1 and P2 requests R2
    processes[0].requested["R1"] = 1
    processes[1].requested["R2"] = 1

    # Check for cycles in the RAG (simple cycle detection)
    visited = set()
    stack = []

    def dfs(process):
        visited.add(process)
        stack.append(process)

        for r in resources:
            if process.requested[r.name] > 0:
                if process.allocated[r.name] + process.requested[r.name] > r.max_instances:
                    return True
                if processes[r.allocated[process.name]].name not in visited:
```

```
                if dfs(processes[r.allocated[process.name]]):
                    return True
        stack.pop()
        return False

    for p in processes:
        if p not in visited:
            if dfs(p):
                return True

    return False

# Example usage
if _name_ == "_main_":
    R1 = Resource("R1")
    R1.available = 2
    R1.max_instances = 3

    R2 = Resource("R2")
    R2.available = 1
    R2.max_instances = 2

    P1 = Process("P1")
    P2 = Process("P2")

    resources = [R1, R2]
    processes = [P1, P2]

    if detect_deadlock(resources, processes):
        print("Deadlock detected!")
    else:
```

```
print("No deadlock detected.")
```