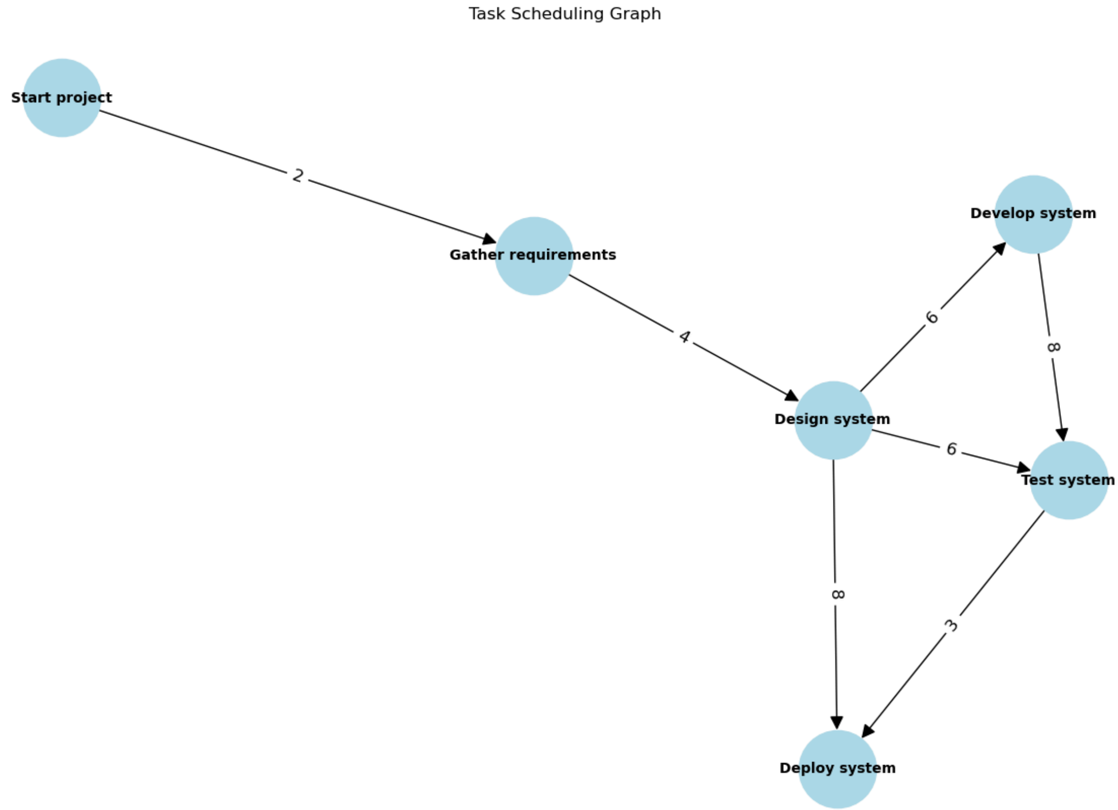
In [22]:
```
import heapq

def dijkstra(graph, start):
    # Initialize distances dictionary with all nodes set to infinity
    distances = {node: float('infinity') for node in graph}
    # Distance to start node is 0
    distances[start] = 0

    # Initialize priority queue with tuple (distance, node)
    priority_queue = [(0, start)]
    # Convert list to heap
    heapq.heapify(priority_queue)

    # Initialize dictionary to store previous nodes in optimal path
    previous_nodes = {node: None for node in graph}

    # Dijkstra's algorithm main loop
    while priority_queue:
        # Pop node with smallest distance from priority queue
        current_distance, current_node = heapq.heappop(priority_queue)

        # If current distance is greater than recorded distance, skip processing
        if current_distance > distances[current_node]:
            continue

        # Iterate over neighbors of current node
        for neighbor, weight in graph[current_node].items():
            # Calculate distance to neighbor via current node
            distance = current_distance + weight

            # If new path to neighbor is shorter, update distance and previous node
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                # Push updated distance and neighbor to priority queue
                heapq.heappush(priority_queue, (distance, neighbor))

    # Return distances and previous nodes for each node
    return distances, previous_nodes

def get_task_order(previous_nodes, start, end):
    path = []
    current_node = end
    # Reconstruct optimal path from end to start node
    while current_node is not None:
        path.append(current_node)
        current_node = previous_nodes[current_node]
    # Reverse path to get correct order from start to end node
    path = path[::-1]
    return path

# Representing tasks and their dependencies
graph = {
    'TaskA': {'TaskB': 2},
    'TaskB': {'TaskC': 4, 'TaskD': 6},
    'TaskC': {'TaskE': 6},
    'TaskD': {'TaskE': 8},
    'TaskE': {'TaskF': 3},
```

```
    'TaskF': {}
}

# Apply Dijkstra's algorithm from 'TaskA' to find distances and optimal paths
distances, previous_nodes = dijkstra(graph, 'TaskA')

# Get the optimal task order from 'TaskA' to 'TaskF'
task_order = get_task_order(previous_nodes, 'TaskA', 'TaskF')

# Print results: optimal task order and distances from 'TaskA'
print("Task Order:", task_order)
print("Distances:", distances)
```

```
Task Order: ['TaskA', 'TaskB', 'TaskC', 'TaskE', 'TaskF']
Distances: {'TaskA': 0, 'TaskB': 2, 'TaskC': 6, 'TaskD': 8, 'TaskE': 12, 'TaskF': 15}
```

In [19]:

```
import matplotlib.pyplot as plt
import networkx as nx

# Create a directed graph
G = nx.DiGraph()

# Define tasks with updated dependencies
tasks = [
    ('Start project', 'Gather requirements', 2),
    ('Gather requirements', 'Design system', 4),
    ('Design system', 'Develop system', 6),
    ('Develop system', 'Test system', 8),
    ('Test system', 'Deploy system', 3)
]

# Add edges with weights (durations)
for task in tasks:
    G.add_edge(task[0], task[1], weight=task[2])

# Update dependencies to match the requirement
G.add_edge('Design system', 'Test system', weight=6)
G.add_edge('Design system', 'Deploy system', weight=8)

# Position nodes using a layout
pos = nx.spring_layout(G)

# Draw the graph
plt.figure(figsize=(12, 8))
nx.draw(G, pos, with_labels=True, node_size=3000, node_color='lightblue', font_size=10

# Draw edge labels
edge_labels = {(u, v): d['weight'] for u, v, d in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=12)

# Display the graph
plt.title("Task Scheduling Graph")
plt.show()
```


Task Scheduling Graph
Start project
2
Gather requirements
4
Design system
6
Develop system
8
6
Test system
8
3
Deploy system