KIET Group of Institutions, Ghaziabad

Computer Science and Information Technology

Project Report

on

Task Scheduler

SUBJECT: Design and Analysis of Algorithms

(KCS-503)

Submitted By:

SAUBHAGYA CHAUDHARI (2100290110146)

SAURABH KASHYAP (2100290110147)

ACKNOWLEDGEMENT

I've got this golden opportunity to express my kind gratitude and sincere thanks to my Head of Institution, KIET Group of Institutions of Engineering and Technology, and Head of Department of "**Computer Science and Information Technology"** for their kind support and necessary counselling in the preparation of this project report. I'm also indebted to each and every person responsible for the making up of this project directly or indirectly.

I must also acknowledge or deep debt of gratitude each one of my colleague who led this project come out in the way it is. It's my hard work and untiring sincere efforts and mutual cooperation to bring out the project work. Last but not the least, I would like to thank my parents for their sound counselling and cheerful support. They have always inspired us and kept our spirit up.

SAUBHAGYA CHAUDHARI(2100290110146)

SAURABH KASHYAP(2100290110147)

B.Tech(CSIT)

Semester: 5th

INDEX

PROBLEM STATEMENT

A task scheduler is a system that manages and executes tasks based on their priority levels. In this context, a task refers to a unit of work that needs to be executed, and each task is associated with a priority value. The priority queue and min-heap are data structures that help efficiently organize and retrieve tasks based on their priorities.

Priority Queue with Min-Heap: The priority queue is implemented using a min-heap data structure. A min-heap is a binary tree where the value of each node is less than or equal to the values of its children, ensuring that the smallest element (highest priority in this context) is at the root of the heap.

ALGORITHM WITH ANALYSIS

Heapify is an algorithm used to create or maintain the heap property of a binary heap data structure. The heap property ensures that the value of each node is less than or equal to the values of its children, creating a specific order that is useful for priority queue implementations.

Two main operations:

1. HeapifyUp

2. HeapifyDown

HeapifyUp :

HeapifyUp is used when a new element is inserted into the heap, typically at the end, and we

need to ensure that the heap property is maintained.

Algorithm for HeapifyUp:

```
procedure heapifyUp(arr, index):
    while index > 0:
        parent = (index - 1) / 2

        if arr[index] < arr[parent]:  // Min-Heap, for Max-Heap use >
            swap(arr[index], arr[parent])
            index = parent
        else:
            break
```

HeapifyDown:

HeapifyDown is used when an element is removed or replaced from the root of the heap, and we need to ensure that the heap property is maintained.

Algorithm for HeapifyDown:

```
procedure heapifyDown(arr, index, heapSize):
    leftChild = 2 * index + 1
    rightChild = 2 * index + 2
    smallest = index

    if leftChild < heapSize and arr[leftChild] < arr[smallest]:
        smallest = leftChild

    if rightChild < heapSize and arr[rightChild] < arr[smallest]:
        smallest = rightChild

    if smallest ≠ index:
        swap(arr[index], arr[smallest])
        heapifyDown(arr, smallest, heapSize)
```

1. HeapifyUp:

Time Complexity:

- The worst-case time complexity of HeapifyUp is O(log n), where n is the number of elements in the heap.
- In the worst case, the height of the heap is log n, and each swap in the while loop takes constant time.

Space Complexity:

- The space complexity is O(1) as the algorithm only requires a constant amount of extra space for temporary variables.

2. HeapifyDown:

Time Complexity:

- The worst-case time complexity of HeapifyDown is O(log n), where n is the number of elements in the heap.
- Similar to HeapifyUp, the worst-case time complexity is determined by the height of the heap.

Space Complexity:

- The space complexity is O(1) as the algorithm only requires a constant amount of extra space for temporary variables.

CODING AND IMPLEMENTATION

```cpp
#include <iostream>
#include <queue>
#include <vector>

struct Task {
    std::string name;
    int priority;
    int expectedDuration;
    int finalPriority;

    Task(const std::string& n, int p, int d)
        : name(n), priority(p), expectedDuration(d), finalPriority(p / d) {}
};

struct CompareTasks {
    bool operator()(const Task& t1, const Task& t2) {
        // Comparison for the priority queue (min-heap)
        return t1.finalPriority > t2.finalPriority;
    }
};

class TaskScheduler {
public:
    void addTask(const std::string& name, int priority, int expectedDuration) {
        Task newTask(name, priority, expectedDuration);
        tasks.push(newTask);
    }

    void deleteTask(const std::string& name) {
        std::priority_queue<Task, std::vector<Task>, CompareTasks> tempQueue;
        while (!tasks.empty()) {
            if (tasks.top().name != name) {
                tempQueue.push(tasks.top());
            }
            tasks.pop();
        }
        tasks = tempQueue;
```

```
    }

    std::string getNextTask() {
        if (tasks.empty()) {
            return "No tasks in the queue";
        }

        Task nextTask = tasks.top();
        tasks.pop();
        return nextTask.name;
    }

    void printTasks() {
        std::priority_queue<Task, std::vector<Task>, CompareTasks> tempQueue = tasks;
        while (!tempQueue.empty()) {
            Task task = tempQueue.top();
            std::cout << "Task: " << task.name << ", Priority: " << task.finalPriority << std::endl;
            tempQueue.pop();
        }
    }

private:
    std::priority_queue<Task, std::vector<Task>, CompareTasks> tasks;
};

int main() {
    TaskScheduler scheduler;

    scheduler.addTask("Task 3", 1, 2);
    scheduler.addTask("Task 1", 2, 1);
    scheduler.addTask("Task 2", 3, 3);

    std::cout << "Current tasks:" << std::endl;
    scheduler.printTasks();

    std::cout << "\nNext task: " << scheduler.getNextTask() << std::endl;

    scheduler.deleteTask("Task 1");

    std::cout << "\nRemaining tasks:" << std::endl;
    scheduler.printTasks();

    return 0;
}
```

SAMPLE OUTPUT

```
d:\DAAlab\output>cd "d:\DAAlab\output"

d:\DAAlab\output>.\"main.exe"
Current tasks:
Task: Task 3, Priority: 0
Task: Task 2, Priority: 1
Task: Task 1, Priority: 2

Next task: Task 3

Remaining tasks:
Task: Task 2, Priority: 1

d:\DAAlab\output>
```

CONCLUSION

- A task scheduler using a priority queue and a min-heap data structure is a powerful and efficient mechanism for managing and executing tasks based on their priority levels. The use of a priority queue with a min-heap ensures that the highest priority tasks are readily accessible, allowing for quick retrieval and execution.

1. Efficient Task Retrieval:

The min-heap data structure ensures that the task with the highest priority is always at the root of the heap, allowing for constant-time retrieval.

2. Dynamic Priority Management:

The scheduler can easily handle dynamic changes in task priorities. Tasks can be dequeued, their priorities updated, and then re-enqueued, ensuring flexibility in task scheduling.

3. Priority-Based Execution:

Tasks are executed based on their priority levels, allowing the scheduler to focus on high-priority tasks first. This is crucial in scenarios where tasks have varying levels of urgency or importance.