Optimizing the Task Scheduling Algorithm with LLM Assistance

Step 1: Understanding the Problem

SwiftCollab's current API request handling system uses a **priority queue** to ensure high-priority requests (e.g., authentication, health checks) are processed first. However, the implementation faces **performance bottlenecks** when handling large workloads due to inefficient sorting operations.

Key Issues:

1. **Inefficient Sorting:** Using List.Sort() to order requests results in **O(n log n)** complexity for each enqueue operation.
2. **Slow Dequeue:** Removing the highest-priority request takes **O(n)** time.
3. **Lack of Bulk Processing:** The system does not support **batch enqueuing**.
4. **No Thread Safety:** The queue lacks mechanisms to handle concurrent access safely.

Goal: Optimize the priority queue using **a binary heap (min-heap/max-heap)** to improve **insertion and removal efficiency**, support bulk processing, and ensure **thread safety**.

Step 2: Reviewing the Initial Implementation

Given C# Code (Partial Implementation with Issues)

```
using System;

using System.Collections.Generic;


public class ApiRequest

{

   public string Endpoint { get; set; }

   public int Priority { get; set; }


   public ApiRequest(string endpoint, int priority)

   {

      Endpoint = endpoint;

      Priority = priority;

   }

}
```

```
public class ApiRequestQueue
{
   private List<ApiRequest> queue = new List<ApiRequest>();

   public void Enqueue(ApiRequest request)
   {
      queue.Add(request);
      queue.Sort((a, b) => a.Priority.CompareTo(b.Priority)); // O(n log n)
   }

   public ApiRequest? Dequeue()
   {
      if (queue.Count == 0) return null;
      ApiRequest request = queue[0];
      queue.RemoveAt(0); // O(n)
      return request;
   }
}
```

Problems in this Code:

- **Enqueue Complexity:** Sort() runs in **O(n log n)** each time a request is added.
- **Dequeue Complexity:** Removing the first element shifts the rest, making it **O(n)**.
- **Scalability Issues:** Large workloads make sorting increasingly expensive.
- **Concurrency Issues:** This queue is **not thread-safe** in multi-threaded environments.

Step 3: LLM-Powered Optimizations

After prompting an LLM (e.g., Microsoft Copilot), the following suggestions were provided:

1. **Use a Binary Heap (Min-Heap) Instead of Sorting a List**
   - Enqueue operation improves from **O(n log n) → O(log n)**.
   - Dequeue operation improves from **O(n) → O(log n)**.
2. **Implement Bulk Enqueue Support**
   - Allows efficient batch insertion instead of inserting one by one.

3. **Use a Thread-Safe Priority Queue**
   - Utilize ConcurrentDictionary or lock mechanisms for safe multi-threading.
4. **Optimize Memory Allocation**
   - Use SortedDictionary<int, Queue<ApiRequest>> to group same-priority requests.

Step 4: Optimized Implementation

Improved C# Implementation with Min-Heap & Thread Safety

```
using System;
using System.Collections.Generic;
using System.Threading;

public class ApiRequest
{
    public string Endpoint { get; set; }
    public int Priority { get; set; }

    public ApiRequest(string endpoint, int priority)
    {
        Endpoint = endpoint;
        Priority = priority;
    }
}

// Min-Heap based Priority Queue
public class ApiRequestQueue
{
    private readonly List<ApiRequest> heap = new List<ApiRequest>();
    private readonly object lockObj = new object();

    // Enqueue in O(log n)
    public void Enqueue(ApiRequest request)
```

```
{
    lock (lockObj)
    {
        heap.Add(request);
        HeapifyUp(heap.Count - 1);
    }
}

// Bulk Enqueue for efficiency
public void EnqueueBatch(IEnumerable<ApiRequest> requests)
{
    lock (lockObj)
    {
        foreach (var request in requests)
        {
            heap.Add(request);
            HeapifyUp(heap.Count - 1);
        }
    }
}

// Dequeue in O(log n)
public ApiRequest? Dequeue()
{
    lock (lockObj)
    {
        if (heap.Count == 0) return null;

        ApiRequest topRequest = heap[0];
        heap[0] = heap[heap.Count - 1];
        heap.RemoveAt(heap.Count - 1);
```

```
        HeapifyDown(0);

        return topRequest;
    }
}

// Heapify Up to maintain Min-Heap property
private void HeapifyUp(int index)
{
    while (index > 0)
    {
        int parent = (index - 1) / 2;
        if (heap[index].Priority >= heap[parent].Priority) break;
        Swap(index, parent);
        index = parent;
    }
}

// Heapify Down to maintain Min-Heap property
private void HeapifyDown(int index)
{
    int leftChild, rightChild, smallest;
    while (true)
    {
        leftChild = 2 * index + 1;
        rightChild = 2 * index + 2;
        smallest = index;

        if (leftChild < heap.Count && heap[leftChild].Priority < heap[smallest].Priority)
            smallest = leftChild;
```

```
            if (rightChild < heap.Count && heap[rightChild].Priority < heap[smallest].Priority)
                smallest = rightChild;

            if (smallest == index) break;

            Swap(index, smallest);
            index = smallest;
        }
    }

    // Swap helper function
    private void Swap(int i, int j)
    {
        (heap[i], heap[j]) = (heap[j], heap[i]);
    }
}

// Main Program to Test the Optimized Queue
public class Program
{
    public static void Main()
    {
        ApiRequestQueue queue = new ApiRequestQueue();

        queue.Enqueue(new ApiRequest("/auth", 1));
        queue.Enqueue(new ApiRequest("/data", 3));
        queue.Enqueue(new ApiRequest("/healthcheck", 2));

        Console.WriteLine($"Processing: {queue.Dequeue()?.Endpoint}"); // Expected: /auth

        queue.EnqueueBatch(new List<ApiRequest>
```

```
        {
            new ApiRequest("/payment", 2),
            new ApiRequest("/user", 1),
            new ApiRequest("/logs", 4)
        });

        while (true)
        {
            var request = queue.Dequeue();
            if (request == null) break;
            Console.WriteLine($"Processing: {request.Endpoint}");
        }
    }
}
```

Step 5: Analysis & Reflection

Improvements from LLM Suggestions

Issue	Old Implementation	Optimized Implementation
Enqueue Complexity	O(n log n) (due to sorting)	**O(log n)** (heap insertion)
Dequeue Complexity	O(n) (list shifting)	**O(log n)** (heap removal)
Bulk Processing	Not supported	**Batch insert in O(k log n)**
Thread Safety	Not thread-safe	**Uses lock for safe multi-threading**

Lessons Learned from LLM Assistance

1. **Efficient Data Structures Matter:** Using **binary heaps** significantly improves the efficiency of priority queues.
2. **Bulk Processing Optimization:** Batch operations can reduce computational overhead.
3. **LLMs Provide High-Level Guidance:** While LLMs provide useful suggestions, **code validation and refinement** are crucial.
4. **Thread Safety Needs Manual Handling:** LLMs suggested ConcurrentDictionary, but **using locks** was more appropriate for a heap.

Conclusion

By leveraging **LLM-assisted insights**, we successfully optimized **SwiftCollab's API request scheduler**:

- **Replaced List.Sort() with a Min-Heap** for **O(log n)** efficiency.
- **Added batch processing** to improve system throughput.
- **Ensured thread safety** to handle concurrent API requests.
- **Improved scalability** to handle large request volumes efficiently.

This **LLM-driven approach** showcases how **AI can enhance algorithmic optimization** while still requiring **human expertise to refine and validate suggestions**. 🚀