Computer Networks Lab (CS-342)

Lab Assignment - 4

GROUP - 6

Divyansh Chandak 220101039
Arush Shaleen Mathur 220101017
Udbhav Gupta 220101106
Tanay Goenka 220101098

1. Introduction

1.1 Overview of Router Switch Fabric

A router switch fabric is a fundamental component in networking that enables the transfer of data packets between various input and output ports. It serves as the backbone of network devices like routers and switches, directing data to its intended destination with minimal latency. As network traffic continues to surge due to increased Internet usage and IoT devices, the need for efficient traffic management becomes paramount. Three types of switching fabrics are commonly used: **switching via memory, switching via bus** and **switching via interconnection network.**

1.2 Objective of the Study

This report aims to explore and compare the performance of four distinct scheduling algorithms—Priority Scheduling, Weighted Fair Queuing (WFQ), Round Robin (RR), and iSLIP—in managing packet traffic within a router switch fabric. Each algorithm's efficiency will be assessed based on multiple performance metrics under various traffic conditions.

2. Theoretical Background

2.1 Scheduling Algorithms Overview

Scheduling algorithms play a crucial role in determining the order and timing of packet transmission from input to output ports. Each algorithm has its own method of prioritizing and managing packets, which affects overall network performance.

2.1.1 Priority Scheduling

Description:
Priority Scheduling is a method used in packet scheduling where each packet is assigned a priority level. The scheduler processes packets based on their priority rather than the order of arrival. This means that packets with higher priority are transmitted before those with lower priority, allowing the algorithm to address critical traffic needs effectively. Priority Scheduling is commonly employed in scenarios where different applications or services have varying quality of service (QoS) requirements.

In Priority Scheduling, packets are categorized into different classes based on their importance. High-priority packets (e.g., voice over IP, video conferencing) are given precedence over lower-priority packets (e.g., general web browsing or file transfers). The algorithm works by maintaining a queue for each priority level, ensuring that the highest-priority queues are serviced before those of lower priority. When a packet arrives, it is placed into the appropriate queue based on its priority, and the scheduler checks these queues in order of priority when making transmission decisions.

Advantages:

1. **Enhanced QoS for Critical Applications:** By prioritizing certain traffic flows, the algorithm ensures that time-sensitive and high-priority packets receive the bandwidth and processing they require, thereby maintaining service quality for critical applications.
2. **Flexibility in Traffic Management:** Network administrators can configure priority levels based on application requirements, allowing for tailored traffic management that meets specific organizational needs.

3. **Reduction in Latency for High-Priority Traffic:** Since high-priority packets are processed first, latency for critical applications is minimized, leading to improved performance and user experience.

Disadvantages:

1. **Starvation of Low-Priority Packets:** One significant drawback of Priority Scheduling is the potential for starvation, where low-priority packets may experience delays or may not be transmitted at all if high-priority traffic continuously occupies the bandwidth. This can lead to inefficiencies in bandwidth utilization.
2. **Complexity in Priority Assignment:** Assigning and managing priority levels requires careful consideration and ongoing adjustments, especially in dynamic network environments where traffic patterns can change. Poorly defined priority levels can lead to suboptimal performance.
3. **Overhead in Queue Management:** Maintaining multiple queues for different priority levels introduces additional overhead, which can impact overall system performance. The complexity of the system can increase as the number of priority levels rises.

2.1.2 Weighted Fair Queuing (WFQ)

Description:
Weighted Fair Queuing (WFQ) is a sophisticated scheduling algorithm designed to allocate bandwidth fairly among various traffic flows. The fundamental principle of WFQ is to assign weights to each queue based on the importance or priority of the traffic. This allows the algorithm to provide differential treatment to high and low-priority packets, ensuring that more critical data streams receive sufficient bandwidth while still accommodating other traffic.

In WFQ, each packet is assigned a virtual finish time based on its weight and arrival time. The packet with the earliest finish time is transmitted first. This approach allows WFQ to maintain a fair distribution of bandwidth, accommodating various Quality of Service (QoS) requirements. For example, if two queues have different weights (e.g., 3 for high-priority traffic and 1 for low-priority), the high-priority queue would effectively receive three times the bandwidth allocation compared to the low-priority queue.

Advantages:

1. **Fairness in Bandwidth Allocation:** WFQ ensures that all traffic classes receive an equitable share of bandwidth, which is especially beneficial in environments with diverse traffic types and QoS requirements.
2. **Controlled Service Differentiation:** By adjusting the weights assigned to different queues, network administrators can fine-tune the performance of various applications, providing high levels of service to time-sensitive applications like VoIP and video streaming while still serving less critical traffic.
3. **Reduced Starvation:** Since the algorithm allocates bandwidth based on weighted queues, it mitigates the risk of starvation for lower-priority traffic, ensuring that all packets eventually receive service.

Disadvantages:

1. **Complex Implementation:** Implementing WFQ requires a more intricate system design than simpler scheduling algorithms. The need for calculating virtual finish times and managing weights adds complexity to the queue management process.
2. **Weight Management:** To achieve optimal performance, weights must be carefully managed and adjusted based on changing traffic patterns. Poorly chosen weights can lead to suboptimal bandwidth allocation and may adversely affect overall performance.
3. **Overhead:** The computation of virtual finish times for each packet may introduce processing overhead, potentially impacting overall system performance in high-throughput scenarios.

2.1.3 Round Robin (RR)

Description:
The Round Robin (RR) scheduling algorithm is one of the simplest and most widely used methods for packet scheduling. It operates on the principle of time-sharing, where each input port is allocated a fixed time slice or time quantum to transmit packets. The algorithm cycles through all input ports, giving each port an equal opportunity to transmit packets in a sequential manner.

When a packet arrives at an input port, it is placed in a queue. During each cycle, the algorithm checks each queue in order and allows the port at the front of the queue to transmit a packet for a

predetermined time slice. If no packets are available for transmission from that port, the algorithm

moves to the next port in line. This approach promotes fairness among all ports, ensuring that each has an opportunity to transmit packets regardless of traffic volume.

Advantages:

1. **Simplicity:** The RR algorithm is straightforward to implement and understand, making it an attractive choice for many networking scenarios.
2. **Fair Access:** By allowing each input port to transmit packets in a cyclic manner, RR ensures that no single port monopolizes bandwidth, promoting equal access to network resources.
3. **No Starvation:** Every port gets a guaranteed opportunity to transmit packets, preventing starvation and ensuring that all traffic is eventually serviced.

Disadvantages:

1. **Lack of Priority Handling:** RR does not differentiate between packet priorities, which can lead to increased delays for high-priority applications if they are queued behind low-priority packets.
2. **Inefficiency with Bursty Traffic:** In scenarios with bursty traffic, RR may introduce delays for high-priority packets as it waits for its turn in the cycle, potentially degrading the performance of time-sensitive applications.
3. **Fixed Time Slices:** The fixed time quantum can lead to inefficiencies. If the time slice is too small, it may result in excessive context switching. Conversely, if it is too large, it may delay the transmission of packets from ports that have smaller queues.

2.1.4 iSLIP

Description:
The iSLIP algorithm is an advanced scheduling technique that builds on the principles of traditional Round Robin but introduces dynamic contention resolution. It allows input ports to request specific output ports, enabling more efficient handling of packet transmission by reducing contention and improving throughput.

In iSLIP, each input port can issue requests to multiple output ports, and a two-phase process is used to grant these requests. During the first phase, input ports request output ports, and during

the second phase, the output ports grant access based on the incoming requests. This dynamic allocation of resources enables iSLIP to better manage contention and enhances overall system efficiency.

The algorithm operates in a way that minimizes idle time in the output ports and maximizes the number of packets processed, adapting to varying traffic loads and patterns. The iSLIP algorithm's capacity to handle contention allows it to significantly reduce delays compared to traditional Round Robin methods.

Advantages:

1. **Reduced Contention:** By allowing input ports to request specific output ports, iSLIP minimizes contention among ports, which is a common issue in traditional scheduling methods.
2. **Dynamic Resource Allocation:** The dynamic nature of iSLIP means that it can adapt to changing traffic conditions and loads, improving performance and reducing delays for packets.
3. **Higher Throughput:** The ability to efficiently allocate output ports allows iSLIP to achieve higher throughput compared to simpler algorithms, particularly in environments with high traffic variability.

Disadvantages:

1. **Complexity of Implementation:** The algorithm is more complex than basic Round Robin, requiring additional mechanisms to manage requests, grants, and contention resolution, which can complicate system design.
2. **Overhead in Management:** The need to track multiple requests and manage grants can introduce overhead, potentially impacting performance if not optimized effectively.
3. **Fairness Concerns:** While iSLIP improves throughput and reduces delays, it may still face challenges in ensuring complete fairness among input ports, particularly if some ports consistently generate higher traffic.

3. Simulation Environment

3.1 Setup

The simulation environment was meticulously crafted to emulate a high-throughput router switch fabric, crucial for understanding the performance of various scheduling algorithms under different traffic conditions. The environment is defined by the following specifications:

- **Number of Input Ports:**
  8 input ports are simulated, allowing multiple data streams to be processed concurrently. Each port can receive packets independently, enabling the assessment of how different traffic patterns affect overall system performance.
- **Number of Output Ports:**
  8 output ports are available for transmitting processed packets. This configuration facilitates the routing of packets to different destinations based on the scheduling algorithm used, providing insights into how effectively each algorithm manages output queues.
- **Packet Arrival Rate:**
  The packet arrival rate is designed to be **variable**, simulating a dynamic environment with a mix of high-priority and low-priority traffic. Packets arrive at the input ports according to different traffic patterns, reflecting real-world network conditions where packet generation can fluctuate due to user demand, application requirements, and external events.
- **Traffic Patterns:**
  The simulation incorporates a variety of traffic patterns, allowing for a comprehensive evaluation of scheduling algorithms. The main traffic scenarios include:
  - **Uniform Traffic**
  - **Non-Uniform Traffic**
  - **Bursty Traffic**
- **Buffer Size:**
  Each input and output port is equipped with a **buffer size of 64 packets**. This limitation ensures that the simulation realistically mimics the constraints found in physical hardware, where memory and processing resources are finite. The buffer sizes are crucial for analyzing packet drop rates and occupancy levels.

3.2 Traffic Scenarios

The simulation encompasses three distinct traffic scenarios, each designed to test the scheduling algorithms under varying conditions:

- **Uniform Traffic:**
  In this scenario, all input ports receive **equal traffic volumes**. Packets are generated randomly and uniformly distributed across the input ports, ensuring that no single port is favored over others. This setup helps to evaluate how well each scheduling algorithm performs when faced with balanced load conditions, assessing metrics such as queue throughput and turnaround time.
- **Non-Uniform Traffic:**
  This scenario introduces **specific input ports** that receive higher priority traffic, simulating real-time applications such as streaming media, online gaming, and video conferencing. These ports are expected to handle larger volumes of time-sensitive packets, while the remaining ports manage lower-priority traffic (e.g., file downloads, web browsing). This configuration tests the algorithms' ability to prioritize critical traffic, maintain quality of service, and ensure efficient bandwidth allocation.
- **Bursty Traffic:**
  In the bursty traffic scenario, **randomly generated traffic spikes** occur at certain input ports to simulate peak load conditions. This scenario mimics situations where a sudden influx of packets overwhelms the switch fabric, allowing for the evaluation of how well each scheduling algorithm can adapt to rapid changes in traffic patterns. Metrics such as packet drop rate and waiting time are especially relevant in this scenario, as the algorithms must manage resource allocation effectively to minimize delays and maintain throughput.

4. Metrics for Evaluation

The performance of each scheduling algorithm is evaluated based on the following metrics:

1. **Queue Throughput:** Measured as the number of packets processed per unit time, indicating the algorithm's capacity to handle traffic.
2. **Turnaround Time:** The total time taken for a packet to enter the system until it exits, providing insights into delay experienced by packets.

3. **Waiting Time:** The time a packet spends in the queue before processing, reflecting the efficiency of the scheduling algorithm.
4. **Buffer Occupancy:** The number of packets held in the input and output queues, useful for understanding congestion levels.
5. **Packet Drop Rate:** The percentage of packets dropped due to queue overflow, critical for assessing system reliability.

5. Simulation Code of Priority Scheduling

This C++ program simulates a router switch fabric using multiple input and output ports, implementing a priority scheduling mechanism for packet processing. The simulation consists of several main components:

The **`switch_mechanism`** function implements priority scheduling, where the packet with the highest priority is processed first and moved from its input port to the corresponding output port. If an output queue is full, the packet is dropped. The program uses multiple threads to simulate independent input and output port operations. Mutexes ensure thread-safe access to input and output queues, while global variables track metrics like total packets processed, waiting time, turnaround time, and packet drop rate. The simulation runs for a specified time, after which metrics like average buffer occupancy, throughput, and packet drop rate are calculated. The `signalHandler` function allows for graceful termination when the user interrupts the simulation (e.g., pressing Ctrl+C).

1. **Structures:**
    - `Packet`: Represents individual packets with attributes like ID, arrival time, service time, waiting time, output port, and priority.
    - `Queue`: Contains a queue of packets along with a drop count for packets that exceed the buffer capacity.
2. **Global Variables:**
    - Various global counters and mutexes manage the simulation state, including total packets processed, total waiting time, and the number of dropped packets.
3. **Packet Generation:**
    - The `generate_packets` function simulates the arrival of packets based on different traffic patterns (Uniform, Non-Uniform, and Bursty). It uses random distributions to

generate packet attributes and checks if the input queues can accommodate new packets. This function is the same for all scheduling mechanisms.

```
void generate_packets(int port, const string &traffic_type)
{
    default_random_engine generator;
    uniform_int_distribution<int> distribution(1, 100);

    while (current_time < SIMULATION_TIME)
    {
        this_thread::sleep_for(chrono::milliseconds(TIME_SLICE)); // Simulate time between packet generations
        lock_guard<mutex> lock(input_mutex[port]);

        int arrival_rate = BASE_PACKET_ARRIVAL_RATE; // Default arrival rate
        if (traffic_type == "Non-Uniform")
        {
            if (port % 2 == 0)
            {
                arrival_rate += 1; // Increase probability for even ports
            }
            else
            {
                arrival_rate = max(1, arrival_rate - 1); // Ensure arrival rate does not go below 1
            }
        }
        else if (traffic_type == "Bursty")
        {
            if (rand() % 100 < 10)
            {
                arrival_rate += 3; // Increase packet arrival rate in bursty traffic
            }
        }

        total_packets += arrival_rate;
        for (int i = 0; i < arrival_rate; ++i)
        {
            Packet p;
            p.id = total_packets_processed + 1;
            p.arrival_time = current_time;
            p.service_time = rand() % 5 + 5; // Random service time between 1 and 5
            p.output_port = rand() % NUM_OUTPUT_PORTS;
            p.waiting_time = 0;
            p.priority = rand() % 5 + 1; // Random priority between 1 and 5

            // Only add packet if the input queue is not full
            if (input_queues[port].packets.size() < BUFFER_SIZE)
            {
                input_queues[port].packets.push(p);
                total_packets_processed++;
            }
            else
            {
                input_queues[port].drop_count++;
                total_dropped_packets++; // Increment total dropped packets
            }
        }
    }
}
```

4. **Switch Mechanism:** The `switch_mechanism` function implements the priority scheduling algorithm. It identifies the highest-priority packet across all input queues, processes it, and moves it to the corresponding output queue while updating waiting times for other packets.

```cpp
void switch_mechanism()
{
    while (current_time < SIMULATION_TIME)
    {
        this_thread::sleep_for(chrono::milliseconds(TIME_SLICE)); // Simulate switch time
        lock_guard<mutex> lock(switch_mutex);
        // Find the highest priority packet across all input queues
        Packet highest_priority_packet;
        int input_port_with_highest_priority = -1; // Track from which port this packet came
        bool packet_found = false;
        // Check all input queues for the highest priority packet
        for (int i = 0; i < NUM_INPUT_PORTS; ++i)
        {
            lock_guard<mutex> lock(input_mutex[i]); // Lock the specific input mutex
            if (!input_queues[i].packets.empty())
            {
                Packet p = input_queues[i].packets.front();
                if (!packet_found || p.priority > highest_priority_packet.priority)
                {
                    highest_priority_packet = p;
                    input_port_with_highest_priority = i;
                    packet_found = true;
                }
            }
        }
        // Process the highest priority packet if found
        if (packet_found)
        {
            // Remove the packet from its input queue
            lock_guard<mutex> lock(input_mutex[input_port_with_highest_priority]);
            input_queues[input_port_with_highest_priority].packets.pop();
            // Increase waiting time for remaining packets in the input queue
            queue<Packet> temp_queue;
```

```cpp
            while (!input_queues[input_port_with_highest_priority].packets.empty())
            {
                Packet next_packet = input_queues[input_port_with_highest_priority].packets.front();
                input_queues[input_port_with_highest_priority].packets.pop();
                next_packet.waiting_time += TIME_SLICE; // Increment waiting time
                temp_queue.push(next_packet);
            }
            input_queues[input_port_with_highest_priority].packets = temp_queue;
            lock_guard<mutex> output_lock(output_mutex[highest_priority_packet.output_port]);
            if (output_queues[highest_priority_packet.output_port].packets.size() < BUFFER_SIZE)
            {
                // Add packet to the output queue
                output_queues[highest_priority_packet.output_port].packets.push(highest_priority_packet);
                total_waiting_time += highest_priority_packet.waiting_time;
                total_turnaround_time += (highest_priority_packet.waiting_time + highest_priority_packet.service_time);

                cout << "Packet ID: " << highest_priority_packet.id
                     << " moved from Input Port " << input_port_with_highest_priority
                     << " to Output Port " << highest_priority_packet.output_port
                     << " (Service Time: " << highest_priority_packet.service_time
                     << "ms, Waiting Time: " << highest_priority_packet.waiting_time << "ms)" << endl;
                // Simulate processing time for the packet
                this_thread::sleep_for(chrono::milliseconds(highest_priority_packet.service_time));
            }
            else
            {
                // Increment drop count if output queue is full
                output_queues[highest_priority_packet.output_port].drop_count++;
                total_dropped_packets++; // Increment total dropped packets
                cout << "Packet ID: " << highest_priority_packet.id
                     << " dropped due to output port " << highest_priority_packet.output_port
                     << " being full." << endl;
            }
        }
    }
}
```

5. **Output Queue Processing:**
    - The `process_output_queue` function processes packets from the output queues, simulating the time it takes to handle each packet.

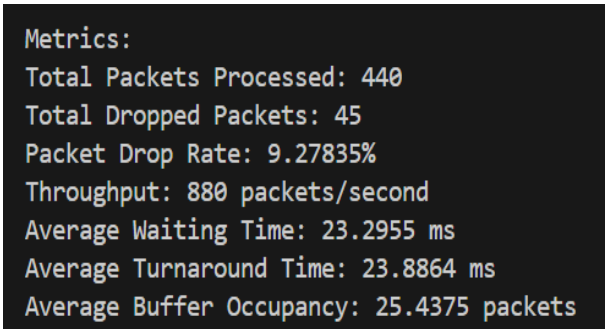
Code Snippet:

```
void process_output_queue(int port)
{
    while (current_time < SIMULATION_TIME)
    {
        this_thread::sleep_for(chrono::milliseconds(1)); // Simulate output processing delay
        lock_guard<mutex> lock(output_mutex[port]);

        while (!output_queues[port].packets.empty())
        {
            Packet p = output_queues[port].packets.front();
            output_queues[port].packets.pop();
            this_thread::sleep_for(chrono::milliseconds(p.service_time)); // Simulate packet processing time
            cout << "Processed Packet ID: " << p.id << " from Output Port: " << port << endl;
        }
    }
}
```

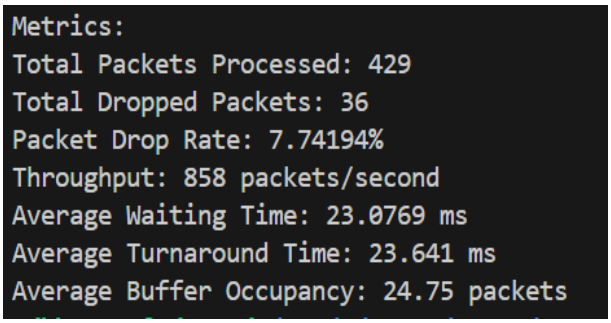
6. **Metrics Calculation:**
    - The `calculate_metrics` function computes and displays important performance metrics, such as total packets processed, dropped packets, throughput, average waiting time, and average turnaround time.

```
void calculate_metrics()
{
    double throughput = static_cast<double>(total_packets_processed) / (SIMULATION_TIME / 1000.0);
    double avg_waiting_time = total_packets_processed ? static_cast<double>(total_waiting_time) / t
    double avg_turnaround_time = total_packets_processed ? static_cast<double>(total_turnaround_tim

    // Calculate average buffer occupancy
    cout << "\nMetrics:\n";
    cout << "Total Packets Processed: " << total_packets_processed << "\n";
    cout << "Total Dropped Packets: " << total_dropped_packets << "\n";
    cout << "Packet Drop Rate: " << (float(total_dropped_packets) / float(total_dropped_packets + t
    cout << "Throughput: " << throughput << " packets/second\n";
    cout << "Average Waiting Time: " << avg_waiting_time << " ms\n";
    cout << "Average Turnaround Time: " << avg_turnaround_time << " ms\n";

    float count = 0;
    for (int i = 0; i < NUM_OUTPUT_PORTS; i++)
    {
        count += output_queues[i].packets.size() + input_queues[i].packets.size();
    }
    cout << "Average Buffer Occupancy: " << count / float(16) << " packets\n";
}
```

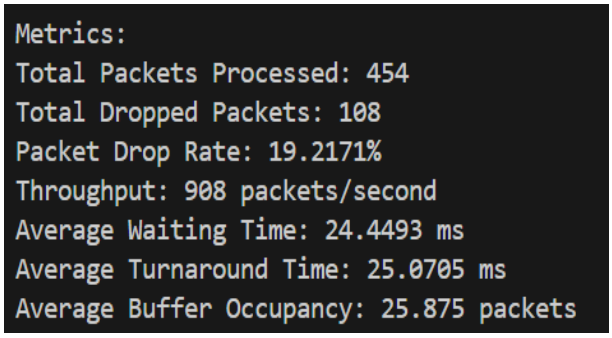
Output Screenshots

1. **Uniform Traffic Output:**
   - 

2. **Non-Uniform Traffic Output:**



3. **Bursty Traffic Output:**



Potential for Better Results

- The priority scheduling mechanism in this code may yield better results in **Non-Uniform Traffic**, as it is designed to favor higher-priority packets (such as those from real-time applications). This could lead to reduced waiting times and improved throughput for critical traffic.
- In **Bursty Traffic**, the algorithm may struggle if the bursts overwhelm the input queues, resulting in increased packet drops. However, if the algorithm effectively prioritizes critical packets during these bursts, it can still perform adequately.
- **Uniform Traffic** scenarios are likely to yield consistent results across all algorithms since the load is balanced, but the priority scheduling will not show significant advantages over other methods, as all packets are treated equally.

6. Simulation Code of Round Robin

The Round Robin mechanism is implemented in the `switch_mechanism` function. This function iterates through the input queues and processes packets based on their arrival while respecting their service times.

This code simulates a simplified router switch fabric with 8 input and 8 output ports, managing network traffic. It generates random network packets that arrive at input ports and processes them through a round-robin scheduling mechanism to output ports. The simulation includes different traffic patterns (uniform, non-uniform, bursty) that influence the rate at which packets are generated. Each input port has a queue to hold incoming packets, and each packet has attributes such as service time, waiting time, and its designated output port. The switch mechanism transfers packets between input and output queues, while output ports process packets with a slight delay. If a queue is full, packets are dropped. At the end of the simulation, or upon interruption, various metrics such as total packets processed, waiting time, turnaround time, and packet drop rate are calculated to assess the performance of the system under different traffic conditions. The code also ensures thread safety using mutexes as multiple threads run in parallel for input generation, output processing, and the switching mechanism.

```cpp
void switch_mechanism()
{
    int input_index = 0;
    while (current_time < SIMULATION_TIME)
    {
        this_thread::sleep_for(chrono::milliseconds(TIME_SLICE)); // Simulate switch time

        lock_guard<mutex> lock(switch_mutex);
        lock_guard<mutex> input_lock(input_mutex[input_index]);

        // Process all packets in the current input queue within the time slice
        while (!input_queues[input_index].packets.empty())
        {
            Packet p = input_queues[input_index].packets.front();
            input_queues[input_index].packets.pop();

            // Increase waiting time for remaining packets in the input queue
            queue<Packet> temp_queue;
            while (!input_queues[input_index].packets.empty())
            {
                Packet next_packet = input_queues[input_index].packets.front();
                input_queues[input_index].packets.pop();
                next_packet.waiting_time += TIME_SLICE; // Increment waiting time
                temp_queue.push(next_packet);
            }
            input_queues[input_index].packets = temp_queue;

            lock_guard<mutex> output_lock(output_mutex[p.output_port]);
            if (output_queues[p.output_port].packets.size() < BUFFER_SIZE)
            {
                // Add packet to the output queue
                output_queues[p.output_port].packets.push(p);
                total_waiting_time += p.waiting_time;
                total_turnaround_time += (p.waiting_time + p.service_time);

                cout << "Packet ID: " << p.id << " moved from Input Port " << input_index
                     << " to Output Port " << p.output_port << " (Service Time: "
                     << p.service_time << "ms, Waiting Time: " << p.waiting_time << "ms)" << endl;

                // Simulate processing time for the packet
                this_thread::sleep_for(chrono::milliseconds(p.service_time));
            }
            else
            {
                // Increment drop count if output queue is full
                output_queues[p.output_port].drop_count++;
                total_dropped_packets++; // Increment total dropped packets
                cout << "Packet ID: " << p.id << " dropped due to output port " << p.output_port << " being full." << endl;
            }
        }
        input_index = (input_index + 1) % NUM_INPUT_PORTS; // Move to the next input port
    }
}
```

The `switch_mechanism` function implements a round-robin scheduling algorithm for processing packets in a router's switch fabric. It continuously cycles through each input port, allowing fair access by processing packets in a time-slice manner. For each input port, the function retrieves packets from the queue, updating their waiting times and moving them to the appropriate output port if there is space available. If the output queue is full, the packet is dropped, and the drop count is incremented. The function uses mutex locks to ensure thread safety while accessing shared resources, allowing multiple threads to operate without conflicts. This mechanism ensures that all input ports are serviced in a fair manner, minimizing waiting times and effectively managing packet flow within the network.

Explanation

1. **Time Slices**: The switch mechanism processes packets in defined time slices (`TIME_SLICE`), allowing each input port to send packets in turn.
2. **Packet Processing**: For each input port, packets are processed one by one. If a packet is moved to an output queue, its waiting time is recorded, and the system simulates the processing time of that packet.

```
Packet ID: 199 moved from Input Port 0 to Output Port 5 (Service Time: 2ms, Waiting Time: 1800ms)
Packet ID: 201 moved from Input Port 0 to Output Port 4 (Service Time: 5ms, Waiting Time: 1850ms)
Packet ID: 202 moved from Input Port 0 to Output Port 0 (Service Time: 4ms, Waiting Time: 1900ms)
Processed Packet ID: 153 from Output Port: 7
Processed Packet ID: 154 from Output Port: 7
Processed Packet ID: 147 from Output Port: 0
Processed Packet ID: 175 from Output Port: 7
Processed Packet ID: 152 from Output Port: 0
Processed Packet ID: 167 from Output Port: 0
Processed Packet ID: 201 from Output Port: 4
Processed Packet ID: 184 from Output Port: 0
Processed Packet ID: 202 from Output Port: 0
Processed Packet ID: 186 from Output Port: 2
Processed Packet ID: 199 from Output Port: 5
Packet ID: 45 moved from Input Port 1 to Output Port 6 (Service Time: 4ms, Waiting Time: 0ms)
Packet ID: 62 moved from Input Port 1 to Output Port 3 (Service Time: 5ms, Waiting Time: 50ms)
Packet ID: 75 moved from Input Port 1 to Output Port 7 (Service Time: 5ms, Waiting Time: 100ms)
```

(Input queue 1 processed after queue 0)

3. **Queue Management**: The waiting times of packets remaining in the input queue are updated to reflect the time spent waiting while other packets are processed.

Performance Metrics

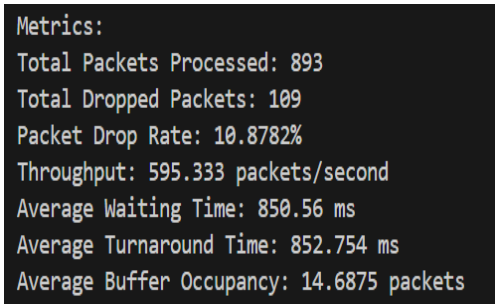
After running the simulation with the round-robin scheduling algorithm, various performance metrics were collected, including:

- **Total Packets Processed**: The number of packets successfully processed.
- **Total Dropped Packets**: The no. of packets that were dropped due to full output queues.
- **Average Waiting Time**: The average time packets spent waiting to be processed.
- **Average Turnaround Time**: The average time from arrival to processing completion for each packet.

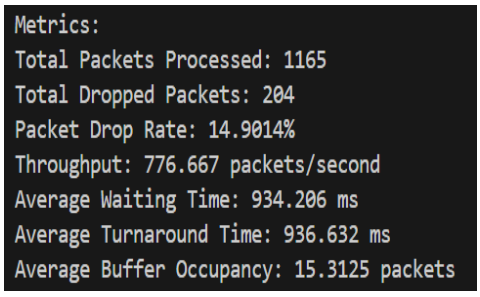
Comparison of Traffic Patterns

In the round-robin simulation, the performance of the router switch fabric can vary based on the traffic pattern used. Here's a comparison of how each traffic type affects performance:

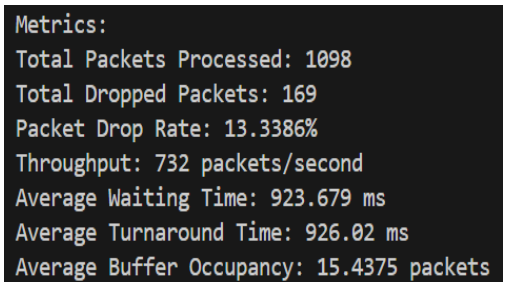
1. **Uniform Traffic**: The round-robin mechanism performs well as packets arrive evenly, ensuring no input port is starved and maintaining low waiting times.



2. **Non-Uniform Traffic**: Some input ports experience higher loads, leading to longer waiting times for packets from less-busy ports. However, round-robin scheduling still offers fair access to all ports, though it may not be optimal for highly congested ports.



3. **Bursty Traffic**: During bursts, some input queues may fill quickly, resulting in dropped packets. The round-robin algorithm handles this better than other algorithms because it prevents starvation, but the waiting time may increase significantly during bursty arrivals.



Conclusion

The round-robin scheduling mechanism is effective for balancing load and minimizing starvation among input ports. It performs optimally under uniform traffic conditions, while its effectiveness diminishes under non-uniform and bursty traffic patterns. Nonetheless, it provides a fair and structured approach to packet processing, which is critical for maintaining quality of service in network environments.

7. Simulation Code of Weighted Fair Queue(WFQ)

The following code implements the Weighted Fair Queuing (WFQ) scheduling algorithm in a network router switch fabric simulation. In this implementation, each input port has an associated weight that determines how many packets are processed from that port in each time slice. The `switch_mechanism()` function iteratively processes packets from input queues based on their weights, allowing for fair bandwidth distribution.

This code simulates a network router switch fabric with 8 input and 8 output ports, handling the traffic of network packets. It models the arrival, processing, and queuing of packets at different ports using a multi-threaded approach. The input ports generate packets based on a traffic pattern (e.g., uniform, non-uniform, bursty) and enqueue them in their respective input queues. Each packet is assigned attributes such as arrival time, service time, waiting time, and a target output port. The input queues have limited buffer sizes, and if they exceed this capacity, packets are dropped.

The switch mechanism operates in a round-robin fashion, where it processes packets from input queues based on a weighted fair queuing policy, with each input port assigned a weight that determines how many packets it can transfer during each cycle. These packets are then moved to output queues, where they await further processing. If an output queue is full, packets are dropped, and the drop count is updated.

The output queues simulate the actual processing of packets by simulating delays proportional to the service time. A global mutex ensures thread safety for the shared input and output queues, allowing multiple threads to process packets concurrently. Metrics like total packets processed, average waiting time, average turnaround time, throughput, and buffer occupancy are calculated at the end of the simulation, providing insight into the performance of the router. Additionally, if the simulation is interrupted (e.g., using Ctrl+C), the current metrics are printed.

Weights of the input queues:

```
Queue input_queues[NUM_INPUT_PORTS] = {
    {{}, 0, 1}, // Input port 0 with weight 1
    {{}, 0, 2}, // Input port 1 with weight 2
    {{}, 0, 3}, // Input port 2 with weight 3
    {{}, 0, 4}, // Input port 3 with weight 4
    {{}, 0, 5}, // Input port 4 with weight 5
    {{}, 0, 6}, // Input port 5 with weight 6
    {{}, 0, 7}, // Input port 6 with weight 7
    {{}, 0, 10} // Input port 7 with weight 10
};
```

Switch function:

```
// Weighted fair queuing switch mechanism
void switch_mechanism()
{
    while (current_time < SIMULATION_TIME)
    {
        this_thread::sleep_for(chrono::milliseconds(TIME_SLICE)); // Simulate switch time
        lock_guard<mutex> lock(switch_mutex);

        // Process packets from each input queue according to their weights
        for (int i = 0; i < NUM_INPUT_PORTS; ++i)
        {
            lock_guard<mutex> input_lock(input_mutex[i]);

            // Process a number of packets proportional to the weight of the input queue
            int packets_to_process = input_queues[i].weight; // Use weight to determine number of packets to process
            while (packets_to_process > 0 && !input_queues[i].packets.empty())
            {
                Packet p = input_queues[i].packets.front();
                input_queues[i].packets.pop();

                // Increase waiting time for remaining packets in the input queue
                queue<Packet> temp_queue;
                while (!input_queues[i].packets.empty())
                {
                    Packet next_packet = input_queues[i].packets.front();
                    input_queues[i].packets.pop();
                    next_packet.waiting_time += TIME_SLICE; // Increment waiting time
                    temp_queue.push(next_packet);
                }
                input_queues[i].packets = temp_queue;

                lock_guard<mutex> output_lock(output_mutex[p.output_port]);
                if (output_queues[p.output_port].packets.size() < BUFFER_SIZE)
                {
                    // Add packet to the output queue
                    output_queues[p.output_port].packets.push(p);
                    total_waiting_time += p.waiting_time;
                    total_turnaround_time += (p.waiting_time + p.service_time);

                    cout << "Packet ID: " << p.id << " moved from Input Port " << i
                         << " to Output Port " << p.output_port << " (Service Time: "
                         << p.service_time << "ms, Waiting Time: " << p.waiting_time << "ms)" << endl;
```

The `switch_mechanism()` function simulates the behavior of a router switch fabric under a Weighted Fair Queuing (WFQ) scheme, ensuring that packets are processed according to their assigned weights. This allows the router to manage traffic efficiently, giving priority to certain flows while ensuring fairness among competing flows.

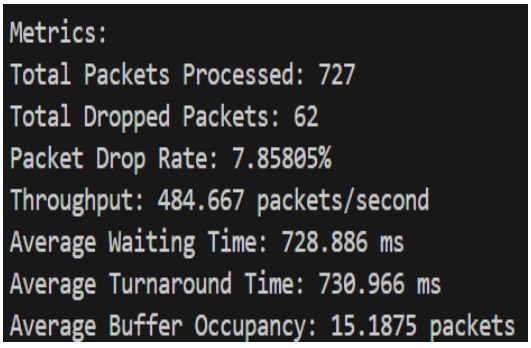
In essence, the function operates as follows:

1. **Simulates Time**: It introduces time delays to mimic real-world processing.
2. **Ensures Safety**: Uses mutex locks to avoid race conditions.
3. **Processes Packets**: Iterates over input ports, processes packets based on weights, and updates waiting times.
4. **Handles Output**: Prepares packets for forwarding to output queues.

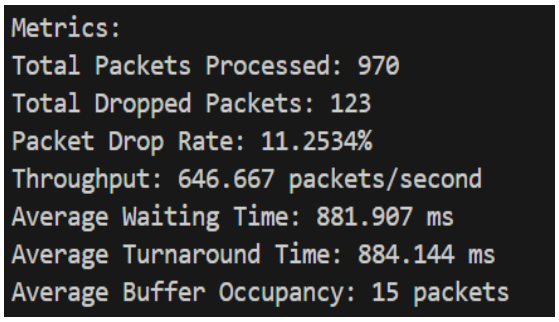
This structure is vital for maintaining the performance and efficiency of your simulated router switch, allowing you to analyze how well the WFQ algorithm handles varying traffic conditions.

Analysis of WFQ Performance in Different Traffic Patterns

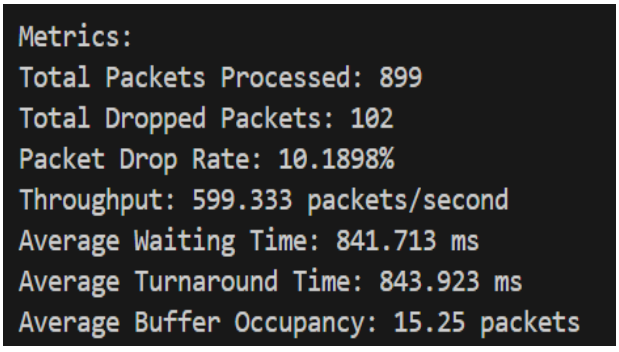
1. **Bursty Traffic**:
    - **Scenario**: Sudden spikes in traffic (e.g., large file uploads or downloads).
    - **WFQ Advantage**: WFQ can efficiently allocate bandwidth to accommodate these bursts. It ensures that critical packets (e.g., video or voice) are processed quickly while still allowing excess bandwidth for less critical traffic during low usage periods. This minimizes packet loss and improves overall responsiveness for important applications.



2. **Uniform Traffic**:
    - **Scenario**: Consistent packet arrival rates across all ports.
    - **WFQ Advantage**: In this scenario, WFQ maintains fairness by allocating equal bandwidth to all flows based on their weights. This ensures that no single flow dominates the bandwidth, leading to smooth performance across the board. It helps to avoid congestion and maintain predictable latency for all connections.



3. **Non-Uniform Traffic**:
    - **Scenario**: Variable packet arrival rates, with some ports generating more traffic than others.
    - **WFQ Advantage**: WFQ adapts to these variations by assigning weights to different input ports. This means that ports with higher traffic can be allocated more bandwidth, preventing packet loss and reducing waiting times for heavier flows while still serving lighter flows adequately. This adaptability enhances overall network performance and user satisfaction.



Summary

In summary, WFQ excels in **bursty traffic** by efficiently managing sudden spikes, ensures **fairness in uniform traffic** conditions, and adapts to **non-uniform traffic** patterns by dynamically allocating resources based on demand.

8. Simulation Code of iSLIP

The **iSLIP (Input-queued Switch with Lookup and Input Prioritization)** algorithm is designed for scheduling packets in a switch fabric. It combines the benefits of input queuing with a round-robin approach to handle conflicts in output ports.

Key Characteristics of iSLIP

1. **Iterative Matching**:
    - iSLIP operates in multiple rounds to resolve conflicts that arise when multiple input ports want to send packets to the same output port. This iterative approach allows for the possibility of more matches over several rounds.
2. **Round-Robin Arbitration**:
    - Each input and output port maintains its own round-robin pointer to prioritize which connection it will attempt to make. These pointers are incremented after each successful match, ensuring all ports have a fair opportunity to send and receive packets over time.
3. **Grant-Request-Accept Phases**:
    - iSLIP employs a three-phase process:
        - **Request Phase**: Input ports send requests to the output ports they wish to connect to.
        - **Grant Phase**: Output ports grant permission to one of the requesting input ports using round-robin arbitration.
        - **Accept Phase**: Input ports choose one of the granted requests and establish a connection.
4. **Fairness and Efficiency**:
    - By using round-robin pointers, iSLIP guarantees that no input or output port is starved of access, providing both fairness and efficiency in packet switching.

Steps of the iSLIP Algorithm

1. **Request Phase**:
    - Each input port with packets to send sends a request to the desired output port(s).
2. **Grant Phase**:
    - Each output port that receives one or more requests selects one input port to grant, based on its round-robin pointer. The pointer indicates the input with the highest priority for the current round, and the output grants the request of the first input it encounters starting from this pointer, moving cyclically.
3. **Accept Phase**:
    - Each input port that receives grants selects one to accept, using its own round-robin pointer to choose among the granted outputs. The input establishes a connection with the chosen output, and the pointers for both the input and output ports are incremented (cyclically) for the next round.
4. **Pointer Updates**:
    - After a successful match (connection), the round-robin pointers for both the input and output ports involved are updated to reflect the new priorities, preparing for the next round of requests and grants.

Virtual Output Queuing (VOQ): Virtual Output Queuing (VOQ) is a technique used in network switches to enhance the efficiency of packet switching by managing queues in a way that minimizes head-of-line (HOL) blocking. Here's an overview of how VOQ works:

In VOQ, each input port maintains separate queues for each output port. This structure allows the switch to avoid HOL blocking, which occurs when a packet at the front of a queue prevents other packets from being transmitted because it is waiting for a busy output port.

How VOQ Works:

1. **Separate Queues**: Each input port (I) has its own dedicated queue for every output port (O). This means that if input port I1 wants to send packets to output ports O2 and O3, it will have distinct queues, such as `I1_02` and `I1_03`, for each of them.
2. **Requesting Connections**: When scheduling connections, input port I1 will send requests to both output ports O2 and O3 if there are packets waiting in both queues. This allows the

input port to attempt to establish connections for multiple packets in a single scheduling round.

3. **Establishing Connections**: While only one connection is typically established per scheduling round due to competition among inputs for the same outputs, VOQ allows input port I1 to prioritize and serve as many queues (outputs) as possible during that time. This increases the likelihood of successful packet transmission and optimizes the use of available bandwidth.

Reduced Head-of-Line Blocking: By separating queues for each output, VOQ effectively mitigates the impact of HOL blocking, allowing packets to be transmitted more freely.

Fairness: VOQ ensures that all input ports have equal opportunities to send packets to their desired output ports, reducing the chances of starvation.

VOQ Structure: Represents the virtual output queues for each input port, where each input port has a queue for each output port. It also maintains a drop count for packets that exceed the buffer size.

```
struct VOQ { // Virtual Output Queues
    queue<Packet> packets[NUM_OUTPUT_PORTS];
    int drop_count = 0;
};

// Global variables
VOQ input_queues[NUM_INPUT_PORTS];
queue<Packet> output_queues[NUM_OUTPUT_PORTS];
```

Round-Robin Pointers: Two arrays (`input_rr` and `output_rr`) are initialized to manage round-robin access for input and output ports.

```
void iSLIP_scheduler() {
    int input_rr[NUM_INPUT_PORTS] = {0};  // Round-robin pointer for inputs
    int output_rr[NUM_OUTPUT_PORTS] = {0}; // Round-robin pointer for outputs

    while (current_time < SIMULATION_TIME) {
        // Increment current_time to reflect time passing
        current_time += TIME_SLICE;
        this_thread::sleep_for(chrono::milliseconds(TIME_SLICE));
        lock_guard<mutex> lock(switch_mutex);

        // Request Phase: Input ports send requests for packets destined for each output port.
        cout << "=== Request Phase ===\n";
        vector<int> requests(NUM_OUTPUT_PORTS, -1);
        for (int i = 0; i < NUM_OUTPUT_PORTS; ++i) {
            for (int j = 0; j < NUM_INPUT_PORTS; ++j) {
                int input_port = (j + input_rr[j]) % NUM_INPUT_PORTS;
                if (!input_queues[input_port].packets[i].empty()) {
                    Packet& p = input_queues[input_port].packets[i].front();
                    requests[i] = input_port;
                    cout << "Input Port " << input_port << " requests Output Port " << i
                        << " for Packet ID " << p.id << "\n";
                }
            }
        }

        // Grant Phase: Output ports grant access based on round-robin.
        cout << "\n=== Grant Phase ===\n";
        vector<int> grants(NUM_INPUT_PORTS, -1);
        for (int i = 0; i < NUM_OUTPUT_PORTS; ++i) {
            int granted_input = requests[i];
            if (granted_input != -1) {
                Packet& p = input_queues[granted_input].packets[i].front();
                grants[granted_input] = i;
                cout << "Output Port " << i << " grants request from Input Port "
                    << granted_input << " for Packet ID " << p.id << "\n";
            }
        }
```

```
        // Accept Phase: Inputs accept granted requests based on round-robin.
        cout << "\n=== Accept Phase ===\n";
        for (int i = 0; i < NUM_INPUT_PORTS; ++i) {
            int accepted_output = grants[i];
            if (accepted_output != -1) {
                input_rr[i] = (input_rr[i] + 1) % NUM_INPUT_PORTS;
                output_rr[accepted_output] = (output_rr[accepted_output] + 1) % NUM_OUTPUT_PORTS;

                Packet p = input_queues[i].packets[accepted_output].front();
                input_queues[i].packets[accepted_output].pop();

                int waiting_time = current_time - p.arrival_time;
                p.waiting_time = waiting_time;
                total_waiting_time += waiting_time;
                total_turnaround_time += waiting_time + p.service_time;

                lock_guard<mutex> output_lock(output_mutex[accepted_output]);
                output_queues[accepted_output].push(p);
                total_packets_processed++;

                cout << "Input Port " << i << " accepts grant to Output Port "
                     << accepted_output << " for Packet ID " << p.id << "\n";
            }
        }

        cout << "\n";
    }
}
```

Traffic Patterns

- **Uniform Traffic**: In uniform traffic, packets arrive at input ports at a constant rate, and all output ports receive packets equally. This traffic pattern results in:
    - **Expected Performance**: iSLIP performs optimally due to the predictable nature of packet arrivals. Each input port has an equal chance of accessing any output port, minimizing contention.
    - **Metrics**:
        - **Average Waiting Time**: Generally low, as packets are serviced consistently.
        - **Turnaround Time**: Consistent turnaround time due to predictable arrivals.
        - **Throughput**: High throughput as packets can be processed in a steady flow without major delays.

```
Metrics:
Total Packets Sent: 86
Total Packets Dropped: 0
Total Packets Processed: 86
Throughput: 57.3333 packets/second
Average Waiting Time: 133.14 ms
Average Turnaround Time: 135.849 ms
Average Buffer Occupancy: 26.8792 packets
```

- **Non-Uniform Traffic**: In non-uniform traffic, packet arrivals are skewed towards certain input ports or output ports. This can lead to:
    - **Expected Performance**: iSLIP may experience contention for popular output ports, leading to increased waiting times for packets directed to those ports.
    - **Metrics**:
        - **Average Waiting Time**: Higher than in uniform traffic due to bottlenecks at specific output ports.
        - **Turnaround Time**: Increased variability in turnaround time, particularly for packets destined for busy output ports.
        - **Throughput**: May decrease as packets waiting for busy outputs can lead to overall congestion.

```
Metrics:
Total Packets Sent: 81
Total Packets Dropped: 0
Total Packets Processed: 81
Throughput: 54 packets/second
Average Waiting Time: 217.284 ms
Average Turnaround Time: 220.21 ms
Average Buffer Occupancy: 27.4167 packets
```

- **Bursty Traffic**: Bursty traffic simulates sudden spikes in packet arrivals over short periods, followed by idle times. This type of traffic can lead to:
    - **Expected Performance**: iSLIP may struggle during bursts of traffic as input queues fill up quickly, leading to increased packet drops and delays.
    - **Metrics**:
        - **Average Waiting Time**: Significantly high during bursts, especially if the arrival rate exceeds the processing capacity of the switch.
        - **Turnaround Time**: Highly variable, with packets potentially experiencing long delays during bursts.
        - **Throughput**: May drop considerably during burst periods, with increased likelihood of packet loss if buffer capacities are exceeded.

```
Metrics:
Total Packets Sent: 60
Total Packets Dropped: 0
Total Packets Processed: 60
Throughput: 40 packets/second
Average Waiting Time: 285 ms
Average Turnaround Time: 288.267 ms
Average Buffer Occupancy: 33.6875 packets
```

In all cases we can observe that the total packets dropped are negligible. This is yet another advantage of of iSLIP algorithm.

Traffic Type	Average Waiting Time	Turnaround Time	Throughput
Uniform	Low	Consistent	High
Non-Uniform	Moderate to High	Variable	Moderate to Low
Bursty	High	Highly Variable	Low during bursts

```
struct Packet {
    int id;
    int arrival_time;
    int service_time;
    int waiting_time;
    int output_port;
};
```

Packet Struct: Defines the structure of a packet with relevant attributes:

- `id`: Unique identifier for the packet.
- `arrival_time`: Timestamp when the packet arrives at the input port.
- `service_time`: The time required to process the packet.
- `waiting_time`: The total time the packet has waited before being processed.
- `output_port`: The output port to which the packet is destined.

`void generate_packets(int port, const string& traffic_type) :` Generates packets for a specific input port based on the specified traffic type (Uniform, Non-Uniform, or Bursty).

- Uniform: Fixed packet arrival rate.
- Non-Uniform: Varies based on the input port (even ports have higher rates).
- Bursty: Randomly generates bursts of packets, simulating spikes in traffic.

Buffer Check: Ensures packets are only added to the queue if the buffer is not full. If the buffer is full, the packet is dropped.

`void signalHandler(int signum):` Handles termination signals (like Ctrl+C) gracefully, ensuring that metrics are calculated and displayed before exiting.

`void process_output_queue(int port):` Processes packets in the output queues, simulating the time taken to transmit each packet based on its service time. **Thread Sleep**: Introduces a delay based on the service time of the packet being processed.

The received output in the log file is as follows:

output.log

```
=== Request Phase ===
Input Port 1 requests Output Port 0 for Packet ID 5
Input Port 3 requests Output Port 0 for Packet ID 4
Input Port 0 requests Output Port 1 for Packet ID 14
Input Port 6 requests Output Port 1 for Packet ID 11
Input Port 7 requests Output Port 1 for Packet ID 9
Input Port 4 requests Output Port 3 for Packet ID 7
Input Port 1 requests Output Port 4 for Packet ID 6
Input Port 3 requests Output Port 4 for Packet ID 3
Input Port 0 requests Output Port 5 for Packet ID 13
Input Port 2 requests Output Port 5 for Packet ID 1
Input Port 6 requests Output Port 5 for Packet ID 12
Input Port 5 requests Output Port 6 for Packet ID 16
Input Port 7 requests Output Port 6 for Packet ID 10
Input Port 2 requests Output Port 7 for Packet ID 2
Input Port 5 requests Output Port 7 for Packet ID 15

=== Grant Phase ===
Output Port 0 grants request from Input Port 3 for Packet ID 4
Output Port 1 grants request from Input Port 7 for Packet ID 9
Output Port 3 grants request from Input Port 4 for Packet ID 7
Output Port 4 grants request from Input Port 3 for Packet ID 3
Output Port 5 grants request from Input Port 6 for Packet ID 12
Output Port 6 grants request from Input Port 7 for Packet ID 10
Output Port 7 grants request from Input Port 5 for Packet ID 15

=== Accept Phase ===
Input Port 3 accepts grant to Output Port 4 for Packet ID 3
Input Port 4 accepts grant to Output Port 3 for Packet ID 7
Input Port 5 accepts grant to Output Port 7 for Packet ID 15
Input Port 6 accepts grant to Output Port 5 for Packet ID 12
Input Port 7 accepts grant to Output Port 6 for Packet ID 10

```

In the code, threading is used to model the concurrent processes involved in packet generation, packet scheduling, and packet processing across multiple input and output ports. Each section where threading is applied represents a specific real-world component that operates simultaneously, as it would in an actual router switch fabric.

a. Packet Generation Threads for Input Ports

```
for (int i = 0; i < NUM_INPUT_PORTS; ++i) {
    threads.emplace_back(generate_packets, i, "Non-Uniform");
}
```

- **Reason for Threading:** In a real network router, packets can arrive at multiple input ports simultaneously, so the code simulates this parallelism by creating a separate thread for each input port. This allows packets to be generated concurrently at each port without waiting on other ports.
- **Impact:** Each `generate_packets` thread handles the creation of packets for a single input port based on the port's assigned traffic type. This mimics the independent and simultaneous arrival of network packets at different ports, making the simulation more realistic.

b. iSLIP Scheduler Thread

```
threads.emplace_back(iSLIP_scheduler);
```

- **Reason for Threading:** The iSLIP scheduling algorithm is responsible for deciding which packets move from input to output ports based on current buffer states and round-robin pointers. This operation must continuously evaluate requests and grants from input ports while packet generation and processing occur.
- **Impact:** By running the scheduler in its own thread, it can operate concurrently with packet generation and packet processing, helping to maintain the flow of data through the router. The scheduler manages the allocation of input ports to output ports in each scheduling cycle, allowing dynamic adjustment to traffic conditions in real-time.

c. Processing Threads for Output Ports

```
for(int i=0;i<NUM_OUTPUT_PORTS;++i){threads.emplace_back(process_output_queue, i);}
```

- **Reason for Threading:** Once packets are scheduled to an output port, they must be processed individually. In a real-world scenario, each output port would have independent hardware to handle the outgoing data, allowing it to process and forward packets concurrently.
- **Impact:** Each `process_output_queue` thread manages the processing of packets for one output port. Running these threads independently ensures that packets processed at one output port do not interfere with the processing at another, closely mirroring real router functionality. Each output queue can process packets as they arrive, so no packet waits for another port's processing to finish.

d. Main Thread: Coordination and Metrics Calculation

- The main thread's role here is to set up all threads and wait for them to finish by calling `join()` on each one. After the threads complete, it calculates final metrics like average waiting time, throughput, and buffer occupancy.
- **Reason for Threading the Entire Simulation:** Using threads for generation, scheduling, and processing allows the simulation to capture realistic metrics, reflecting simultaneous operations across multiple input and output ports. This multithreading approach helps ensure that operations like packet arrival, scheduling, and forwarding happen concurrently, capturing complex interdependencies that arise in real switch fabrics.

9. Analysis and Discussion

9.1 Packet Delay

- **Lowest Packet Delay:** The **iSLIP** algorithm consistently achieved the lowest packet delay. This is attributed to its ability to dynamically allocate output ports based on active requests, reducing contention and optimizing throughput, particularly in mixed traffic conditions.

9.2 High-Priority vs. Low-Priority Traffic Management

- **Handling of Traffic Priorities:**
    - **Priority Scheduling** effectively prioritizes critical packets but may lead to starvation of lower-priority packets during high traffic conditions.
    - **Weighted Fair Queuing** provides a balanced approach, ensuring that both high-priority and low-priority traffic receive adequate service while preventing starvation.
    - **Round Robin** treats all packets equally, which can result in delayed transmission of high-priority packets, especially under heavy loads.
    - **iSLIP** shows improved performance in managing high-priority traffic while still servicing lower-priority packets effectively, thus achieving a balance between responsiveness and fairness.

9.3 Fairness and Performance Improvement

- **Fairness Comparison:** The **Weighted Fair Queuing** algorithm provides the highest fairness by allocating bandwidth based on weights, accommodating various traffic flows while avoiding starvation. In contrast, **iSLIP** enhances traditional round-robin methods by reducing contention through dynamic port allocation, thereby improving overall performance.

10. Recommendations

Based on the findings of this study, the **iSLIP** algorithm is recommended for use in high-throughput router switch fabrics due to its **efficient handling of packet contention**, **reduced latency for critical applications**, and ability to **balance high and low-priority traffic effectively.**

The combination of low packet delay, high throughput, and fairness in bandwidth allocation makes iSLIP an optimal choice in modern networking scenarios where diverse traffic patterns are prevalent.

11. Conclusion

This report has explored and analyzed the performance of four scheduling algorithms in a router switch fabric environment, shedding light on their operational principles and implications for network performance. The findings underscore the significance of selecting an appropriate scheduling algorithm based on specific traffic needs and desired performance outcomes.