OSN Report.

Patanjali B.

2021114014

Implementation of various scheduling algorithms:

1. Round Robin: It was already implemented when given, but the key aspect of the round robin is to keep the process running for a fixed amount of time despite the process being interactive or a long term process.

Looking at the results produced by the RR scheduler:

Average rtime : 14

Average wtime : 158

2.FCFS approach:

The First-Come, First-Served (FCFS) scheduling technique is implemented in the xv6 operating system where the processes are carried out according to the order they appear in the ready queue under FCFS, a non-preemptive scheduling mechanism. In this solution, the timeOfCreation variable is used to iteratively search through all processes in the system to identify the earliest produced runnable process (RUNNABLE state).

The firstProcess pointer, which is maintained by the algorithm, links to NULL. The firstProcess pointer is updated to point to the newly discovered process if a runnable process is identified during the loop and it is either the first encountered or has an earlier creation time than the current firstProcess.

To ensure fairness, if a firstProcess that was previously chosen exists, its lock is freed, enabling other processes to take into account. If a firstProcess that fits the bill is discovered after the iteration is finished, it is marked as RUNNING, and a context switch is made to carry out this process, enabling it to continue running until it is finished. In the absence of any runnable processes, the system remains inactive until a process becomes available for execution. This FCFS implementation guarantees that processes are carried out in accordance with the core FCFS scheduling concept, which is the order in which they are created.

Looking at the results produced by the FCFS implementation:

Average rtime: 14

Average wtime :130

3.MLFQ Approach:

An implementation of the Multi-Level Feedback Queue (MLFQ) scheduling algorithm is that we iterate through a list of processes . It uses update_queue_no() to provide each runnable process that hasn't been assigned to a queue yet a queue number and adds the process to the proper queue. It adjusts its place in the queue if the process was previously added to one. The code locates the first non-empty queue after classifying all runnable processes into queues. It advances to the following iteration if all queues are empty. In the absence of an empty queue, it picks the first process and moves it forward in the execution queue.

Later we determine whether the chosen process is still in the RUNNABLE state. If so, the code takes a lock on the process, changes its status to RUNNING, and gives it a CPU execution task. Using swtch(), the context of the CPU and the process are shifted. Following execution, the process is designated as not needing to be immediately reinserted into the queue, and the lock is released, enabling the scheduling of subsequent processes.

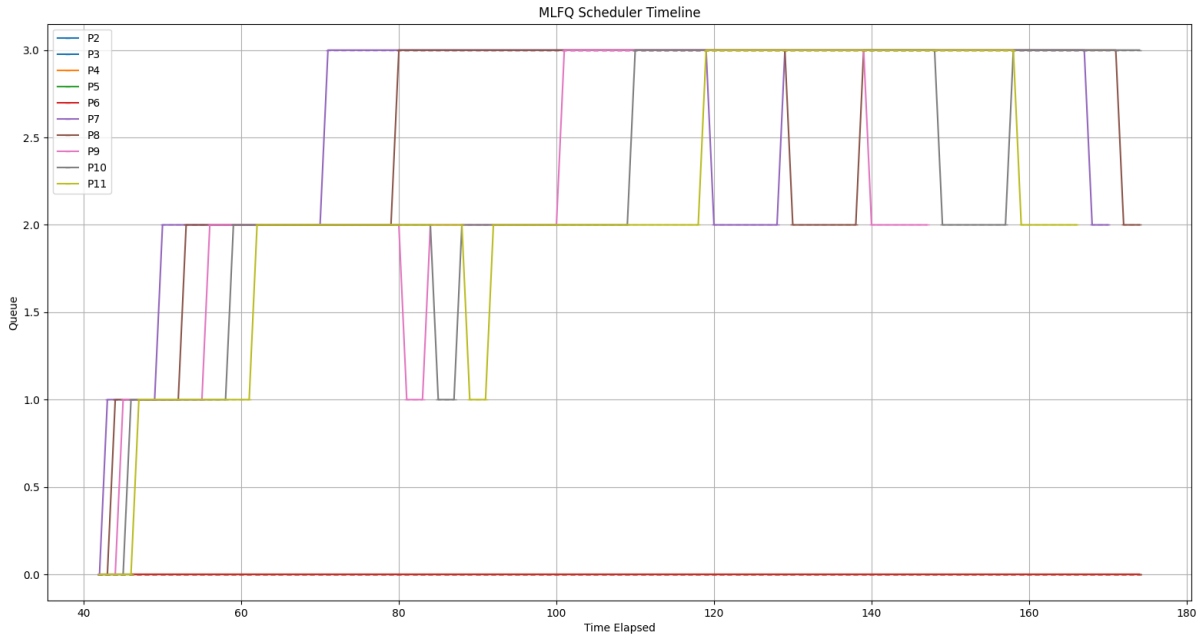
In this context, aging is implemented by the update_queue_no function, which periodically reassigns processes to different

queues based on their execution and waiting times. If a process has exceeded its allotted time in a given queue, it is demoted to a lower-priority queue, ensuring that long-waiting processes are eventually given a chance to execute. This mechanism prevents starvation by allowing lower-priority tasks to gradually move up the queue hierarchy, ensuring fairness and optimal system utilization, hence aging.

Average rtime: 15

Average wtime : 156

Here is the generated graph after piping all the points into a text file and then generating a graph on this:



Part B.

Question 1: How is your implementation of data sequencing and retransmission different from traditional TCP?

My UDP-based approach to data sequencing and retransmission shares similarities with traditional TCP, as both involve assigning sequence numbers to chunks and retransmitting data if acknowledgments (ACKs) aren't received within a timeout. However, there are key distinctions:

Reliability: TCP ensures reliable, in-order data delivery automatically, handling packet loss and out-of-order delivery. In contrast, my UDP implementation necessitates explicit retransmission and reordering, making it less robust and efficient than TCP.

Flow Control: Unlike TCP, my UDP implementation lacks flow control mechanisms. TCP adjusts the sender's rate based on receiver buffer space to prevent congestion. My approach does not include such adjustments, potentially leading to congestion and inefficiencies.

Congestion Control: TCP employs congestion control algorithms to manage network congestion. My UDP implementation lacks

these mechanisms, potentially resulting in network congestion under heavy loads.

Connection Establishment and Termination: TCP follows a connection establishment and termination handshake. In contrast, my UDP approach lacks these features, adhering to UDP's connectionless nature.

Question 2: How can you extend your implementation to account for flow control?

To enhance my implementation with flow control, I can implement a sliding window protocol inspired by TCP:

Sender-Side:

Maintain a Sender Window: Limit the number of unacknowledged chunks in-flight by employing a sender window size.

Check Receiver's Advertised Window: Before sending a new chunk, validate if unacknowledged chunks are within the sender's window and the receiver's advertised window. Wait if necessary, ensuring data transmission aligns with available receiver buffer space.

Dynamic Adjustment: Adjust the sender's window size based on received ACKs and updates to the receiver's advertised window. This ensures optimal data flow without overwhelming the receiver.

Receiver-Side:

Maintain a Receiver Window: Limit the number of out-of-order chunks accepted using a receiver window size.

Include Advertised Window Size in ACKs: Acknowledge only chunks within the receiver's window and defer ACKs for out-of-order chunks. Include the receiver's advertised window size in ACK packets sent back to the sender.

Dynamic Window Size Update: Dynamically update the receiver's window size based on available buffer space, ensuring efficient use of resources.

By implementing this sliding window mechanism, I can control data flow between sender and receiver, preventing receiver overload and facilitating efficient data transfer. It's essential to note that this extension is simplified and lacks advanced TCP features like slow start and congestion avoidance.