Lab05 Report Justin Schlag

1. Problem

This lab requires the implementation of a process scheduling simulator using a first-come, first-serve queue. The simulator must manage processes that arrive at different time steps, execute in order, and update their completion times. When a process finishes, the next process in the queue begins execution. The program must handle adding new processes, canceling the current process, and printing the queue status at each time step. The output should match the provided example to ensure correctness.

2. Solution Description:
The code solves the problem by implementing a queue-based process scheduler that follows the first-come, first-serve scheduling algorithm. A Process class represents individual processes with a name and completion time. The LLQueue class implements a linked list queue to manage waiting processes. The ProcessScheduler class manages the currently running process and the process queue. The simulator runs for a fixed number of time steps, where it may add new processes, update the current process's completion time, remove completed processes, and dequeue the next process when needed. The ProcessSchedulerSimulator driver file orchestrates the simulation and ensures output formatting matches the example.

3. Problems Encountered:

Throughout the implementation of the project, several challenges arose, ranging from syntax errors to more complex logic issues. Initially, there were difficulties ensuring that all required methods were properly implemented according to the QueueI interface, which led to compilation errors. Additionally, run-time errors occurred when attempting to dequeue from an empty queue, which was resolved by incorporating null checks and ensuring that methods handled edge cases appropriately.

One of the more significant logic errors involved the incorrect printing of the process queue. At first, the queue output displayed duplicate lines, and an additional "Queue is empty" message appeared even when processes were present. This issue stemmed from how the printProcessQueue method was handling empty queues and was resolved by adjusting its logic to only print the queue's contents when applicable. There was also an issue with process cancellation, where the scheduler was not properly dequeuing and reassigning the current process, requiring further debugging and modifications.

A frustrating issue involved handling decimal values for process completion times. Due to floating-point precision errors, the completion times were not decreasing as expected, which resulted in inaccurate process updates. This was especially problematic because it led to discrepancies between the program's output and the expected example output.

4. In this lab, the queue is implemented as a linked list queue using the LLQueue class, which follows the First-In, First-Out (FIFO) principle. This means that processes are added to the end of the queue and removed from the front, ensuring that the first process to arrive is executed first.

The ProcessScheduler class uses this queue to manage processes waiting to be executed. When a new process is added and no process is currently running, it becomes the active process. Otherwise, it is enqueued into the process queue. When a process completes, the next process in the queue is dequeued and set as the active process.

5. A queue is most appropriate in scenarios that require first-in, first-out ordering, where elements are processed in the order they arrive. In this lab, a queue is used for process scheduling, ensuring that processes are executed in the sequence they are added. This approach is necessary when managing processes dynamically over time, as it prevents starvation and maintains a fair execution order. The process scheduler dequeues processes as they complete and replaces them with the next available task, maintaining an organized and predictable system for handling execution flow.

6. Queues are commonly used in computing for task scheduling and data buffering. In task scheduling, queues manage processes by ensuring they execute in the order they arrive, preventing starvation and maintaining fairness. This is seen in operating systems where processes are queued before execution. In data buffering, queues handle data transmission between systems operating at different speeds, temporarily storing incoming data before it is processed to prevent loss or overload.

7.

After performing the given enqueue and dequeue operations, the queue will contain the following values:

1. ENQUEUE 1, 2, 3, 4, 5 → Queue: [1, 2, 3, 4, 5] (Head: 1, Tail: 5)
2. DEQUEUE 3 times → Queue: [4, 5] (Head: 4, Tail: 5)
3. ENQUEUE 25, 35, 45, 55 → Queue: [4, 5, 25, 35, 45, 55] (Head: 4, Tail: 55)
4. DEQUEUE 1 time → Queue: [5, 25, 35, 45, 55] (Head: 5, Tail: 55)
5. ENQUEUE 22 → Queue: [5, 25, 35, 45, 55, 22] (Head: 5, Tail: 22)

Final Queue State: [5, 25, 35, 45, 55, 22]
Head: 5
Tail: 22

8.

Index	0	1	2	3	4	5	6	7	8
Value	"d"	"e"	NULL	NULL	"m"	"n"	"o"	NULL	NULL

Head → Index 4 ("m")

Tail → Index 7 (next available NULL)

9.
The method fails to work correctly because the loop runs until the end of the array instead of stopping at the tail index, which causes it to access invalid elements. The condition i < queue.length should be replaced with a check that ensures i does not surpass the tail index, such as i != tail. This prevents the loop from printing unintended values. Additionally, the method does not account for the circular nature of the queue. When i reaches the end of the array, it should wrap around to the beginning instead of exceeding the bounds. To fix this, the increment statement i += 2 should be modified to (i + 2) % queue.length, ensuring that the index correctly wraps when necessary. These changes allow the method to iterate over the queue properly and print every other value as intended.

10.

The method does not work as intended because it incorrectly updates the head pointer while traversing the list, which results in losing the reference to the start of the queue. Instead of modifying head, a separate temp variable should be used to traverse the list. The assignment head = head.link should be replaced with temp = temp.link to ensure that head remains unchanged. Additionally, the while loop condition while(temp.link != null) should be replaced with while(temp != null) because the last node should also be checked. Without this fix, the method may fail to search the final node in the list, leading to incorrect results. These changes ensure that the method correctly iterates through the queue without altering essential references.