Scheduling Policies Implementation:

FCFS :

- Run —> make qemu SCHEDULER=FCFS
- Changes :
  - Modified scheduler code in proc.c for implementation of FCFS scheduling.
  - Modified Trap.c to disable the preemption of the process.
- Implementation :
  - Gone through all processes and selected one with minimum creation time and then allocated cpu to it.
- Average rtime 13, wtime 126

MLFQ :

- Run —> make qemu SCHEDULER=FCFS CPUS=1.
- Added some new variables in struct proc and defined new struct Queue in proc.h .
- Modified the allocproc() function in proc.c to initialise new variables.
- Added new functions enqueue ,dequeue and update for changes in queue.
- Modified scheduler code in proc.c for implementation of MLFQ scheduling.
- Modified trap.c to enable preemption after the time slice.
- Implementation :
  - Aging Processes: The first for loop iterates through all processes (proc) and checks if a runnable process has been waiting for at least 30 ticks (time units). If it has, and it's not in the highest-priority queue (p->level), it gets moved to a lower-priority queue. This aging mechanism prevents starvation of processes in lower-priority queues.

    - Enqueuing Processes: The second for loop iterates through all processes again and enqueues them into their respective queues if they are runnable and haven't already been enqueued (p->inside == 0). This step ensures that processes are placed in the appropriate priority queue based on their current priority level (p->level).
    - Selecting the Next Process: The code then searches for the next process to run. It iterates through the four priority queues and dequeues a runnable process from the highest-priority non-empty queue. The process selected for execution is stored in the curr variable.
    - Running the Selected Process: If a process is selected (curr != 0), it acquires the process's lock, marks it as RUNNING, sets its time, and performs a context switch to run the selected process. After running, it releases the lock.
- Average rtime 13, wtime 151

RR :

- Average rtime 13, wtime 153

Based On rtime :

- Value of time is almost same for all 3 scheduling policies.

Based on wtime :

- Wtime of RR and MLFQ came out to be almost same which are far greater than FCFS.
- RR = MLFQ > FCFS.

Networks :

Difference between my implementation and Traditional TCP :

- In my implementation I used UDP as the main underlying transport protocol, UDP is connectionless but TCP is connection oriented and also lacks many of the features and guarantees that TCP provides.
- TCP establishes a connection between the client and server through a three-way handshake, ensuring that both parties are aware of the communication. In my implementation, there's no connection establishment phase; the client and server can start communicating immediately.

Data sequencing :

- Both my implementation and TCP use sequence numbers to order data packets. However, in TCP, sequence numbers are crucial for ensuring in-order and reliable delivery of data, while in my implementation, they are primarily for tracking and sorting chunks.

Retransmissions :

- **Error Handling :** TCP includes mechanisms for handling various errors, including checksums, windowing, and selective acknowledgment (SACK) options. My implementation has limited implementation of retransmission and no error handling in comparison.
- **Reliability:** TCP is designed to provide reliable data transmission. If a data packet is not acknowledged within a certain timeout or is acknowledged with an error, TCP will automatically retransmit that packet. In my implementation, there is limited retransmission, but it's not as comprehensive or robust as TCP.
- In TCP, the retransmission strategy is complex, with multiple mechanisms to detect lost packets and retransmit them selectively. My implementation lacks these advanced mechanisms and instead focuses on sending chunks sequentially without extensive error recovery.

Flow Control :

Adding flow control to your implementation based on UDP involves ensuring that data is sent at a rate that the receiver can handle without overloading it. Here are some steps

- **Define a Buffer:** Create a buffer on both the client and server sides to hold data that is ready to be sent but hasn't been transmitted yet. This buffer will allow you to control the rate of data transmission.
- **Sender's Buffer Management:**
  a. When the client wants to send data, it first checks if the buffer has space available. If the buffer is full, it waits until space becomes available.
  b. Use a sliding window mechanism to manage the buffer. The sender can only send data within the window size, which prevents overloading the network.
  c. As acknowledgments (ACKs) are received from the server, shift the window forward and make space in the buffer for new data.
  d. Implement a mechanism to handle congestion. If the buffer becomes full, you may implement backpressure by not reading more data from the user until space becomes available.
- **Receiver's Buffer Management:**
  a. On the server side, create a buffer to hold incoming data packets until they can be processed in order.
  b. Implement a mechanism to handle out-of-order packets. When an out-of-order packet arrives, store it in the buffer until the missing packets arrive to maintain proper ordering.
  c. Only process packets from the buffer when they are in sequential order. This ensures that you don't process packets faster than the server can handle.
- **Adjust Chunk Size:** You may need to adjust your chunk size based on the available buffer space and the network conditions. Smaller chunk sizes can allow for finer-grained control.
- **Flow Control Signals:** Optionally, you can implement flow control signals in your protocol. For example, you can include a "window size" field in your packet structure to indicate how many more packets the sender can send.
- **Error Handling:** Implement proper error handling and timeouts to account for scenarios where packets are lost or delayed due to congestion.