Report :

XV6:

1) System calls (GetSyscount & Sigalarm, Sigreturn)

getSyscount:

a) A new user program entry '$U/_getSyscount' was added to the makefile & entry("getsyscount") was added to usys.pl in user folder. Function getSyscount was added to user.h in user folder.

b) A variable to store the "1<<i" called 's1' & an array to store number of times a syscall was called "uint syscall_count[32]" was added to **struct proc** in kernel/proc.h. In order to save the value of the mask across parent and child processes, the fork() function in kernel/proc.c was modified to copy the "1<<i" value from the parent to child process.

c) The function sys_getSyscount() was added to kernel/sysproc.c, which moves the mask value from user space to kernel space by saving it to a variable & then calls getSyscount helper function . The helper function int getSysCount(int mask) was added to kernel/proc.c, which prints the number of times corresponding syscall (by right shifting mask & ANDing with 1) was called by the process (parent)

d) The user program user/getSyscount.c was created, which ensures that the input is in the correct format and then **forks** . In child process it executes the command & in parent process it waits & then calls the sys_getSycount function to print the number of times the corresponding system call was called by the child process.

e) The syscall() function in kernel/syscall.c was modified to increament p->parent->sycall_count[num] ++, ensuring sycall_count array was initialized to '0's in sysproc, defined in the same file itself under syscallnames[] array created to keep track of the list of syscalls. If the syscall does not exist, an error is shown.

Sigalarm and sigreturn:

a) A variable called 'handler' and 'hlp' was added to kernel/proc.h.
'hlp' variable acts as a lock and prevents two different CPU cores
from triggering the signal alarm at once. Other variables, such as 'alarm_tf' (to keep the current state when alarm handler is called) and 'alarm_act' (to set the status of alarm if sigalarm syscall is called), and variables for tick count and tick interval were added { 's_tcks' and 'ticks' } . In **allocproc** function , p->s_tcks = 0;

p->hlp = 1;

b) The function sys_sigalarm() was added to kernel/sysproc.c, which moves the ticks, handler, from user space to kernel space by saving it in the variables and sets the alarm_act value to 1 & p->ticks = intervalj; p->handler = handlerj; .

c) The usertrap() function in kernel/trap.c was modified to trigger the alarm periodically & set hlp to 0 & p->s_tcks++.

d) The function sys_sigreturn() was added to kernel/sysproc.c, which restores the process state to before the alarm handler was called & sets back to 1.

2) Scheduling (LBS, MLFQ) .

LBS:

a) The settickets syscall changes the number of tickets (based on arguments passed) for a particular process (int tickets added to *struct proc*). p->arrival_t =ticks; in allocproc.

b) The scheduling algorithm first calculates the cumulative total of tickets held by all runnable processes.

c) A random value is selected between 0 and the cumulative ticket total (stored in sum_tkt) and stored in variable rtvll using rg function.

d) A process is selected if the randomly generated number falls within the bounds of tickets owned by a process & if any 2 are having same tickets then their arrival times are compared & final **winner** is selected. It is executed .

e) Its done in kernel/proc.c in the scheduler() function if LBS is defined.

MLFQ*:*

Implemented four priority queues with different time slices for each level: 1 tick (level 0), 4 ticks (level 1), 8 ticks (level 2), and 16 ticks (level 3).

If a process uses its entire time slice, it is moved to a lower priority queue. If a process voluntarily gives up CPU (e.g., for I/O), it re-enters the same queue. Implemented priority boosting to avoid starvation, moving all processes to the highest-priority queue after 48 ticks.

Processes in the lowest queue (queue 3) are scheduled using a round-robin approach to ensure fairness.

Modified procdump() to display the current scheduling state, such as queue level and process information, to help in debugging.

Added 'int pqtct; int queue; int wwpqtct; int qnumber;' to struct proq & used 3 new arrays in proc.c for implementing .

Sub :

Graph :

Define Log Structure: Create a structure to store log entries that contain the PID, current queue, and elapsed times

Add Logging Variables: Create an array to store the logs and a variable to keep track of the log index.

Implement Logging Function: Write a function to log the relevant information.

Call the Logging Function: Insert calls to the logging function at appropriate points in the scheduler to capture transitions.

Call this logging function in user/schedulertest.c to ***redirect ">"*** to a file & store the data. Then using python , and get the graph (neglecting first two default process).

Using only **1 CPU:**
RR (default) : Average runtime: 10, Average wait time: 142
LBS: Average runtime: 10, Average wait time: 139
MLFQ: Average runtime: 10, Average wait time: 135

Questions:
What is the implication of adding the arrival time in the lottery based scheduling policy?

A: It is added as arrival_t = ticks in allocproc . It is used to compare 2 processes if they have same tickets , if arrival_t of p1 > p2 , winner is p2.

Fairness Improvement, Reduced Starvation Risk for Same-Ticket Processes, Slight Deviation from Pure Lottery.

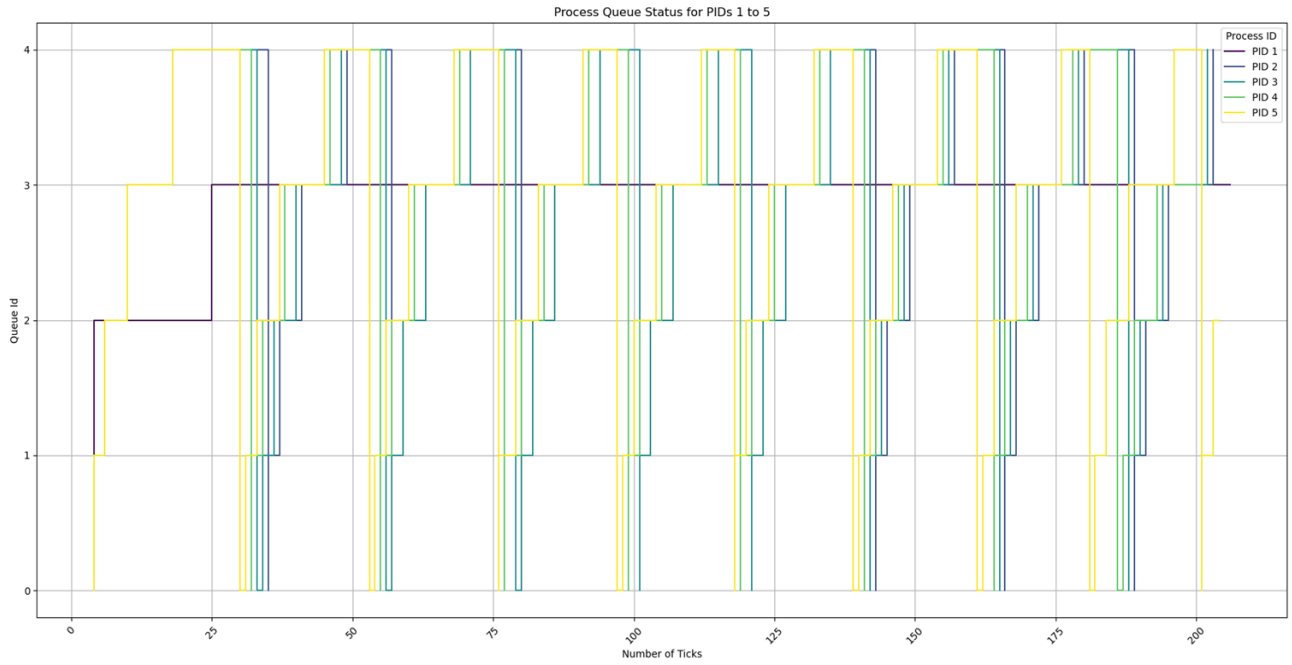
What happens if all processes have the same number of tickets?

A: Randomness increases & any process (which arrived earlier) can finish first compared to setting high tickets for a particular process. In 2nd case the process with higher tickets finishes first usually.

Are there any pitfalls to watch out for?

A: Edgecases & out of limits.

MLFQ Graph:



NETWORKING :

PART A:

Server Implementation
Main Server Logic

The server's primary responsibility is to manage the Tic-Tac-Toe board and handle player connections. Here are the main components:

Socket Creation and Binding: The server first creates a TCP socket, binds it to a local port (8080), and waits for connections from two clients (players).

Handling Player Connections: Once two players are connected, the server begins the game. The server alternates turns between the two clients, ensuring that Player 1 plays 'X' and Player 2 plays 'O'.

Board Management: The game board is a 3x3 grid stored in a 2D array. After each move from a client, the server updates the board and checks whether a player has won or if the game has ended in a draw.

Move Validation: The server ensures that players can only place their symbol in valid, unoccupied positions. If a player attempts an invalid move, they are notified, and the game does not proceed until a valid move is made.

Game Outcome: After each move, the server checks if either player has won (by forming a line of three symbols) or if the board is full, resulting in a draw. In either case, both players are informed, and the game terminates.

Key Functions

setup_board: Initializes the 3x3 Tic-Tac-Toe board to empty spaces.
display_board: Displays the board to the console (mostly for debugging).
check_for_winner: Checks if the current player has won by examining rows, columns, and diagonals.
check_draw: Determines if the game ends in a draw by checking if the board is full.

Communication Protocol

The server sends prompt messages to the players to indicate whose turn it is.
It also transmits the current state of the game board to both clients after each valid move.
If the game concludes, a final message indicating the result (Player 1 Wins, Player 2 Wins, or Draw) is sent to both players.

Client Implementation
Main Client Logic

The client code facilitates the player's interaction with the game. The client connects to the server, receives the board state, and sends back their moves (row and column) as input when prompted by the server.

Socket Creation and Connection: Each client creates a TCP socket and connects to the server IP and port (8080).

Receiving Server Messages: The client waits for the server's prompt messages, such as when it's their turn to make a move or to display the game board.

Sending Moves: When prompted by the server, the player inputs their move (row and column), which is sent to the server for validation and board updates. Used - scanf("%[^\n]s", move); scanf("%c", &p4) **logic**.

Communication Flow

Clients continually listen for messages from the server. If a message contains the phrase "Your move", the client is prompted to provide input for the next move.
Once the client sends its move, it awaits the next server message to update the board or receive the game result.

Game Flow and User Interaction

The game follows the traditional rules of Tic-Tac-Toe:

The server waits for two players to connect.
Player 1 is assigned 'X', and Player 2 is assigned 'O'.
Players take turns making moves by specifying row and column coordinates (e.g., "1 1" for the top-left corner).
After each move, the server updates the board and checks for a win or draw.
If a win or draw condition is met, the server notifies both players and terminates the game.

Part B:

UDP Port is 8081

Specifications Implemented

This project implements the following core functionalities:

Data Sequencing:
The sender (client) breaks down a data payload into multiple chunks (packets), with each chunk containing a sequence number.
The sequence number ensures that the receiver can reorder the data properly if packets arrive out of order.

Acknowledgment and Retransmission:

The receiver sends an ACK message for each packet it successfully receives.

If the sender does not receive an ACK for a packet within a set timeout (150 ms), it resends the packet.

To simulate packet loss, the server randomly skips sending ACKs for every third packet (as per the specification). The client retransmits packets for which no ACK is received, ensuring reliable delivery.

Client Implementation

The client is responsible for:

Dividing the Data into Packets:

The data is divided into chunks, each with a sequence number and content.

The total number of packets is communicated to the server at the beginning.

Sending Packets:

Each packet is sent to the server in sequence. After sending, the client waits for an ACK from the server.

Handling ACKs:

The client waits for the server's acknowledgment (ACK) for each sent packet.

If an ACK is not received within the timeout window, the client retransmits the packet.

Timeout and Retransmission:

If no ACK is received within 150 milliseconds, the packet is resent.

This is done without blocking further sending of other packets, ensuring non-blocking functionality.

A maximum retry mechanism ensures the client does not get stuck indefinitely on a single packet.

Non-Blocking Sockets:

The client uses non-blocking sockets to avoid blocking during waits for ACKs. This ensures the client can keep track of timeouts and retransmissions efficiently.

Key Client Code Features:

Non-blocking Sockets: Allows continuous processing without waiting indefinitely for an ACK.

Timeout Mechanism: If an ACK is not received within the timeout window, the packet is retransmitted.

Packet Resending: Retransmits packets for which an ACK has not been received, ensuring reliability.

Server Implementation

The server is responsible for:

Receiving Packets:
The server listens for incoming packets from the client.
Upon receiving a packet, the server logs the packet's sequence number and content.

Sending ACKs:
For each packet received, the server sends an ACK back to the client, containing the sequence number of the received packet.

Simulating Packet Loss:
As per the specification, the server randomly skips sending ACKs for every third packet to simulate packet loss.
This allows us to test the client's retransmission mechanism.

Reassembling Data:
The server reassembles the received packets in the correct order using the sequence numbers.
Once all packets are received, the server confirms the complete reassembly of the data.

Key Server Code Features:

Random ACK Skipping: Skips ACKs for every third packet to simulate network unreliability.
Handling of Out-of-Order Packets: Reorders the packets at the server side based on their sequence numbers.