Report

XV6

System Calls

1. Gotta count ‘em all
   Here the key changes are made in syscall.h, syscall.c, MAKEFILE, syscount.c, sysproc.c, user.h, usys.pl, proc.h
   To implement this system call feature in XV6, you'll first define a new system call, getSysCount, by adding a syscall number in kernel/syscall.h and implementing it in kernel/sysproc.c. A data structure is needed to maintain a count of system calls for each process, typically an array indexed by syscall number. Modify existing syscall handlers in kernel/syscall.c to increment the corresponding index in this array when a syscall is invoked. Next, create a user-space program called syscount.c that accepts a mask and a command as arguments. It should parse the mask to identify the specific syscall to count, use fork () to create a child process for executing the specified command, and wait for it to finish with wait (), after which it retrieves and prints the syscall count along with the process ID and syscall name in the specified format.

2. Wake me up when my timer ends
   For the alarm test feature, you'll define new system calls sigalarm and sigreturn, again in kernel/syscall.h and kernel/sysproc.c. Create a structure in the process control block (PCB) to store the alarm interval, handler function's address, and a state indicating if the handler is active. In the sigalarm implementation, save the interval and handler address in the PCB and set up a timer using system clock interrupts to trigger the handler function after the specified number of ticks. Modify the timer interrupt handler in kernel/trap.c to check for active alarms and switch context to execute the handler when necessary. Finally, implement sigreturn to restore the saved state of the process, allowing it to resume execution after the handler completes. User-space applications can be created to test the functionality of sigalarm and sigreturn, ensuring that processes can respond to timer signals effectively.

Scheduling

Here only changes I made is in MAKEFILE, proc.c and proc.h

In proc.c only changes are made in allocproc, scheduler and procdump functions

In allocproc just initialization is made, the main code is written in scheduler function

1. LBS

- **Ticket Calculation:** The code first calculates the total number of tickets from all runnable processes by iterating through the process list, ensuring thread safety with locks.
- **Runnable Check:** If no runnable processes exist (total tickets are zero), the scheduler skips the current iteration.
- **Random Winner Selection:** A random ticket number is generated, and the code iterates through runnable processes to accumulate ticket counts, identifying a potential winner when the accumulated count exceeds the random number.
- **Tie-Breaking Logic:** If multiple processes tie in ticket counts, the one with the earlier arrival time is prioritized, ensuring fairness.
- **Process Execution:** The selected process is marked as RUNNING, and the CPU context is switched to execute that process. Locks are released after context switching.

2. MLFQ

- **Boosting Priority:** The code begins with a loop that iterates through a defined number of queues (NQUEUE), invoking the boost_priority() function, which likely elevates the priority of processes that have been waiting longer.
- **Process Iteration:** It iterates through all processes in the system (from proc to &proc[NPROC]), checking their state and queue level.
- **Runnable Processes:** For each process that is in the RUNNABLE state and belongs to the current queue level (i), the code. Increments the ticks_used counter to track how long the process has been using the CPU.
- **Time Slice Management:** The code checks if a process has exhausted its time slice (ticks_used >= time_slice). If so, it demotes the process to the next lower queue (if it's not already at the lowest level) and resets its ticks_used counter.
- **Process Scheduling:** When a runnable process is found, it is marked as RUNNING, and a context switch is performed to start executing that process. The current CPU context is saved, and the process context is loaded.
- **Post-Switch Processing:** After switching contexts, the ticks_used for the running process is reset to zero. This prepares it for the next scheduling round.

- **Release Lock:** The lock for the process is released to allow other processes to be scheduled or modified.
- **Boosting Logic:** The code also includes a boosting mechanism that triggers after a certain number of ticks (BOOST_INTERVAL). When activated, it boosts all processes by resetting their queue levels to the highest priority (level 0) and resetting their ticks_used to ensure they get a fair chance to run.

Networking

- Part-A
  Connection Model
  TCP:
  Establishes a connection-oriented communication. This means a connection is established between the server and each client before any data is sent.
  Uses listen () and accept () to accept connections from clients, ensuring that a stable connection is maintained throughout the session.
  UDP:
  Utilizes a connectionless communication model. There is no need to establish a connection, and packets are sent without a guarantee of delivery.
  Clients send connection requests using sendto(), and the server listens using recvfrom(), without any formal connection setup.

  Socket Setup:
  The server uses socket (), bind (), and listen () to set up a TCP socket and wait for connections from two players (clients). Once both players are connected (accept ()), the game starts.
  Game Flow:
  Board Initialization: The 3x3 Tic-Tac-Toe board is stored in a 2D array and initialized with empty spaces.
  Turn-based Gameplay: The game alternates between two players:
  Player 1 uses 'X', Player 2 uses 'O'.
  On each turn, the server waits for the current player's move, validates it, and updates the board.
  Move Validation: The server ensures each move is valid (row and column within range and not already occupied).
  Game State Broadcasting:
  After each move, the server sends the current state of the board to both players so they can see the updated board.
  Checking for Winner or Draw:

The server checks the board after each move to determine if there's a winner (three matching symbols in a row, column, or diagonal) or if the game ends in a draw (all spaces are filled without a winner).
It then announces the result ("Player 1 Wins", "Player 2 Wins", or "It's a Draw").
Play Again Feature:
After the game ends, the server asks both players if they want to play again.
If both agree, the board is reset, and a new game begins. If one or both declines, the server closes the connection for both players.
Conclusion:
This code runs a server that handles all game logic and communication between the two clients, ensuring the game progresses smoothly, enforcing rules, and resetting or ending the game based on player decisions.

- Part-B
  This C program implements a basic TCP client-server communication system where messages are exchanged in chunks. The program breaks down the messages into smaller pieces (chunks), sends them individually, and reassembles them on the receiving end.
  Both the server and client use threads to handle receiving data while being able to simultaneously send data.
  This allows for asynchronous communication where both sides can send and receive messages independently without blocking each other.
  Exit Condition:

  Either the server or the client can terminate the communication by sending the message /exit, which breaks the loop, closes the connection, and shuts down the program.

  Message Chunking Mechanism:

  Both the server and client break the message into chunks of 3 bytes each.

  Each chunk is wrapped in a DataChunk struct that includes:

  seq_num: The sequence number of the chunk (which part of the message it is).
  total_chunks: The total number of chunks the message has been split into.
  data: The actual part of the message, up to 3 bytes.

  The chunks are sent and received in order, and the message is reconstructed by appending each chunk into its correct position based on seq_num.

  The commented parts in both the client and server code have lines related to skipping certain message chunks and retransmitting skipped chunks.

  The commented part that skips every 3rd chunk of a message is currently inactive. If uncommented, the code will intentionally skip sending every 3rd chunk of a message from the server (or client) to simulate packet loss or missed data.

The second commented block of code relates to retransmitting the skipped chunks after all chunks have been sent.