OSN Mini Project 2

Introduction to XV6 and Networking

XV6

System Calls

1. **syscount**

- The syscount command is used as follows: syscount <mask> command [args].
- A uint64 syscall_count[32] array has been added to the struct proc to store the system call counts for each system call.
- This array is updated in the syscall.c file within the syscall function.
- The sys_getSysCount function is implemented in sysproc.c and retrieves the count of a specific system call for a given process identified by its process ID (pid). It takes two parameters: a bitmask to determine which system call to check and the pid of the target process. The function counts the number of bits in the mask, iterates through the process table to find the specified process, and returns the corresponding system call count from that process's syscall_count array.
- Additionally, a syscount.c file has been added to process user input and provide the correct output.

2. **alarmtest**

- The `alarmtest` command runs the alarm system call. The `usertrap()` function in `trap.c` is modified to handle timer interrupts for this system. When a timer interrupt occurs (`which_dev == 2`), it checks if the process has an active alarm interval and if an alarm is not already in progress. If so, it increments the tick count. When the tick count reaches the alarm interval, it resets the count, marks the alarm as in progress, saves the current trap frame, and sets the program counter to the alarm handler's address.
- In `sysproc.c`, the `sys_sigalarm` and `sys_sigreturn` functions are implemented. The `sys_sigalarm` function sets an alarm for the current process by taking an interval for the duration between alarms and a handler pointer for the alarm handler function. It stores these values in the process structure and initialises `ticks_count` to zero, returning `0` for success.
- The `sys_sigreturn` function restores the process state after handling an alarm. It checks if an alarm is active; if not, it returns `-1`. If an alarm is in progress, it restores the trap frame from `alarm_trapframe`, resets the `alarm_in_progress` flag, and calls `usertrapret()` to return control to the user program, returning `0` for success.

Scheduling

1. **Lottery Based Scheduling: `LBS`**

A new system call, settickets, has been implemented to allow dynamic adjustment of lottery scheduling tickets for a specified process. The command is used as follows:

settickets <pid> <number_of_tickets>

Functionality of settickets

- **Purpose**: Sets the number of lottery scheduling tickets for the process identified by pid.
- **Process Lookup**: The function iterates through the process table to find the process with the specified pid. If found, it updates the tickets field; if not, it returns an error message and -1.
- **Ticket Validation**: The function ensures that the number of tickets is at least 1. If new_tickets is less than 1, it is set to 1.
- **Replacement of Ticket Values**: The existing ticket value is replaced with the new value, rather than adding to it. This prevents multiple processes from accumulating different ticket values.

Modifications in schedulertest.c

In schedulertest.c, the code assigns 10 tickets to CPU-bound processes using the settickets system call. This increases the likelihood that the CPU-bound process will be selected by the scheduler.

Scheduler Modifications in proc.c

The scheduler() function is modified to implement LBS:

- **Total Tickets Calculation**: The first pass counts the total tickets held by runnable processes (RUNNABLE) and stores them in an array of candidates.
- **Winning Ticket Selection**: A winning ticket is randomly chosen based on the total number of tickets. The second pass determines which process holds the winning ticket.
- **Tie-Breaking**: If multiple processes have the same number of tickets, the process that arrived first is selected as the winner.
- **Process Execution**: The winning process is transitioned to the RUNNING state, and the context is switched to execute it.

Random Number Generator: A function for generating random numbers is added to proc.c.

Time Update: The update_time() function is modified to update the time variables of processes in a given state.

Modifications in trap.c

The clockintr() function in trap.c is modified to force the current process to yield the CPU if it is running. This allows the scheduler to re-evaluate and potentially select a different process based on the lottery mechanism, ensuring fair CPU allocation according to the number of tickets each process holds.

Observation :

The following is recorded by running schedulertest 4 times in a sequence.

Average run time	Average Wait time
15	114
29	100
43	86
59	71

The results show an overall trend of increasing average run times with decreasing average wait times, suggesting an evolving scheduling behaviour that might be improving the efficiency of process management over successive runs.

2. Multilevel Feedback Queue: MLFQ

The following have been included in the **proc.h** file

- **In struct proc:**
  1. `int queue_number:` Represents the priority queue number (0 to 3) for the process.
  2. `int ticks_in_queue:` Tracks the number of ticks the process has spent in the queue.
  3. `int wait_time:` Records the total time the process has waited in the queue.
  4. `int in_queue:` Indicates if the process is currently in a queue (1 if true).
  5. `int position_in_queue:` Shows the process's position within its queue

- **Definitions :**
  1. `#define MAX_QUEUE 64`
     Sets the maximum number of processes allowed in each queue.
  2. `#define NUM_QUEUES 4`
     Defines the total number of priority queues.

- **Global Variables:**
  - **`extern int time_slice[NUM_QUEUES]`:** Holds the time slice duration for each priority queue.
  - **`extern int boost_ticks`:** Specifies the number of ticks before a process is boosted to a higher queue. Initially set to zero.
  - **`extern struct proc* queues[NUM_QUEUES][MAX_QUEUE]`:** Arrays that hold processes for each priority queue.
  - **`extern int queue_lengths[NUM_QUEUES]`:** Tracks the number of processes in each priority queue.
- **Functions:**
  - **`extern struct proc* dequeue_from_start(int queue_no)`:** Removes and returns the process at the start of the specified queue.
  - **`extern void queue_at_end(struct proc* p, int queue_no)`:** Adds a process to the end of the specified priority queue.
  - **`extern void queue_at_start(struct proc* p, int queue_no)`:** \Adds a process to the start of the specified priority queue.
  - **`extern void dequeue_at_a_position(struct proc* p, int queue_no)`:** Removes a process from a specific position in the specified queue.

In `proc.c` file:

All the above-mentioned queue functions are implemented in `proc.c`.
Additionally, `void init_q()` is implemented to initialise the priority queues by setting the time slices for each queue and resetting their lengths and entries to zero. This ensures that all queues start empty and are configured for their respective time allocations.

Scheduler Logic:

In the `scheduler()` function, the Multilevel Feedback Queue (MLFQ) logic is implemented.

1. Boosting Mechanism

```
if (boost_ticks >= 48) {
    boost_ticks = 0;
    struct proc* b = 0;
    for (int q = 1; q < NUM_QUEUES; q++) {
        while (queue_lengths[q] > 0) {
            b = dequeue_from_start(q);
            queue_at_end(b, 0);  // Move process to topmost queue
            b->queue_number = 0;
        }
    }
}
```

Explanation:

- This snippet checks if `boost_ticks` has reached 48, indicating it's time to boost the priority of lower-priority processes.
- All processes in queues 1 to 3 are dequeued and reinserted into queue 0 (the highest priority queue). This prevents starvation of processes that may not be getting CPU time.

2. Enqueue Runnable Processes

```
for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state == RUNNABLE && p->in_queue == 0) {
        queue_at_end(p, p->queue_number);
        p->in_queue = 1;
    }
    if (p->state == SLEEPING) {
        queue_at_end(p, p->queue_number);
        p->in_queue = 1;
    }
    release(&p->lock);
}
```

Explanation:

- This loop iterates through all processes to enqueue runnable ones into their respective queues(queue 0).
- It checks if a process is `RUNNABLE` and not already in a queue; if so, it adds the process to its designated queue(queue 0).
- If a process is in the `SLEEPING` state, it indicates that it is waiting for I/O (like reading from a disk or waiting for user input).
- In this case, when the process is reinserted into the same queue it was previously in. This ensures that the process retains its priority and execution context after completing the I/O operation.

3. Finding a Runnable Process

```
struct proc *new = 0;
int runnable_found = 0;
```

```
for (int i = 0; i < NUM_QUEUES; i++) {
    while (queue_lengths[i] > 0) {
        new = dequeue_from_start(i);
        if (new->state == RUNNABLE) {
            queue_at_start(new, i);
            runnable_found = 1;
            break;
        }
    }
    if (runnable_found) break;
}
```

Explanation:

- This snippet searches for the first runnable process starting from the highest priority queue to the lowest.
- It dequeues processes until it finds one that is `RUNNABLE`, at which point it re-inserts the process at the start of its queue and sets `runnable_found` to indicate a runnable process was found.

4. Handling Time Slice and Context Switching

```
if (new) {
    acquire(&new->lock);
    if (new->state == RUNNABLE) {
        if (new->ticks_in_queue >= time_slice[new->queue_number]) {
            dequeue_from_start(new->queue_number);
            if (new->queue_number < NUM_QUEUES - 1) {
                new->queue_number++;
            }
            queue_at_end(new, new->queue_number);
            new->ticks_in_queue = 0;
        }
        if (new->queue_number == NUM_QUEUES - 1) {
            dequeue_from_start(new->queue_number);
            queue_at_end(new, new->queue_number);
        }
```

```
            new->state = RUNNING;
            c->proc = new;
            swtch(&c->context, &new->context);
            c->proc = 0;
        }
        release(&new->lock);
    }
```

Explanation: If a runnable process (`new`) is found, it acquires the lock for that process. It then checks if the process has exceeded its time slice; if so, it moves the process to a lower-priority queue and resets its `ticks_in_queue`. For processes in the lowest-priority queue, it implements round-robin scheduling by removing the process from the front and reinserting it at the back. Finally, it sets the process state to `RUNNING`, performs a context switch to start executing the process, and releases the lock.

5. Yield Behaviour

```
void yield(void) {
    struct proc *p = myproc();
    acquire(&p->lock);
    #ifdef MLFQ
    if (p->state == SLEEPING) {
    #endif
        p->state = RUNNABLE;
    #ifdef MLFQ
    }
    #endif
    sched();
    release(&p->lock);
}
```

Explanation: The `yield` function ensures that processes transition to the `RUNNABLE` state when they are in the `RUNNING` state. If they are in the `SLEEPING` state (due to preemption to reserve the parent process until the child completes), they remain in that state, preventing preemption errors.

6. Updating Process Statistics

In the `updatetime` function, process statistics such as `ticks_in_queue` and `wait_time` are updated to track the time spent in the Running and Waiting periods.

1. Promoting Processes Based on Wait Time

```
struct proc* p_new;
for (p_new = proc; p_new < &proc[NPROC]; p_new++) {
    if (p_new != 0) {
        if (p_new->state == RUNNABLE && p_new->in_queue == 1) {
            if (p_new->wait_time >= 30) {
                dequeue_at_a_position(p_new, p_new->queue_number);
                p_new->in_queue = 0;
                p_new->wait_time = 0;
                if (p_new->queue_number) {
                    p_new->queue_number--;
                    queue_at_end(p_new, p_new->queue_number);
                }
            }
        }
    }
}
```

Explanation: To prevent ageing, the code checks if the wait time of processes that are `RUNNABLE` exceeds a predefined threshold (in this case, 30 ticks). If it does, the process's priority is increased (queue level decreased) by removing it from its current queue and re-queuing it at the end of the higher priority queue. This mechanism ensures that processes waiting too long are promoted, enhancing responsiveness.

2. Checking Time Slice Exceedance

```
int time_in_queue = p->ticks_in_queue;
int assigned_ticks = time_slice[p->queue_number];
if (time_in_queue >= assigned_ticks) {
    if (queue_lengths[p->queue_number] > 0) {
        p->in_queue = 0;
        dequeue_from_start(p->queue_number);
    }
```

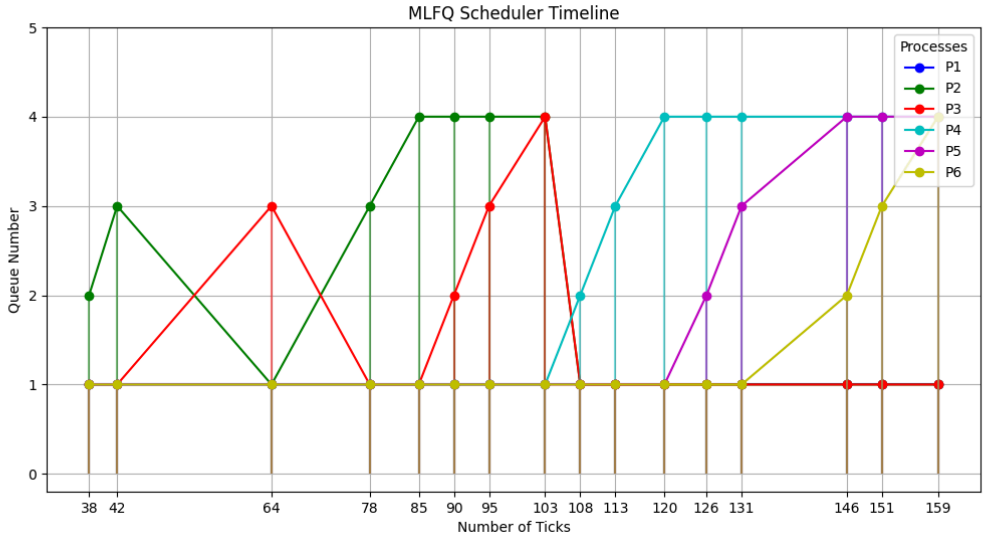
```
    if (p->queue_number < NUM_QUEUES - 1) {
        p->queue_number++;
        queue_at_end(p, p->queue_number);
    }
    p->ticks_in_queue = 0;
    p->wait_time = 0;
}
```

Explanation: This code checks whether the running time of the current process has exceeded its allocated time slice based on its queue number. If it has, the process is dequeued from its current queue. If it is not in the lowest priority queue, its queue number is incremented (lowering its priority), and it is re-enqueued at the end of the new queue. The process's runtime and wait time are then reset, allowing it to start fresh in the next scheduling cycle. This mechanism promotes fairness and responsiveness in the scheduling of processes within a multi-level feedback queue system.

3. Update Boost Ticks

In the `clockintr()` function, the boost time is updated.

MLFQ Graph Analysis



Waittime = 30

Observation :
$ schedulertest
Average rtime 9, wtime 117

The output i.e., Average running time and average wait time also depends on the waittime chosen. The above output is for the **waittime = 30** .
Output when schedulertest is run various time in a sequence

Average run time	Average Wait time
9	117
17	108
26	99
35	91

The analysis of the average run time and wait time in the scheduler test suggests that as processes are executed, the scheduler optimises the handling of processes, resulting in decreased waiting times even as total execution times rise.

Overall Comparison among the Scheduler Policies:

- Both the **MLFQ** and **LBS** scheduling algorithms exhibit improvements in average wait times over their runs, LBS shows a more substantial reduction in wait times while maintaining higher average run times. The MLFQ scheduler starts with lower run times but exhibits steady growth and improved efficiency over time. The choice between MLFQ and LBS may depend on specific workload characteristics and the importance of minimising wait times versus managing execution times effectively.

Pitfalls and Implications (LBS)

Arrival Time Impact: Adding arrival time influences the outcome of the lottery, ensuring that timely processes have a better chance of running, but it can lead to unpredictability in scheduling if not managed carefully.

Equal Ticket Scenarios: If all processes possess the same number of tickets, the scheduler relies solely on arrival times, potentially leading to inefficient scheduling and longer wait times for certain processes.

Overall, this implementation not only demonstrates the mechanics of lottery-based scheduling but also encourages thoughtful consideration of process management strategies in operating systems.

Networking

1. XOXO

- The server_tcp.c file implements a multiplayer Tic-Tac-Toe game server using TCP sockets. It initialises a game board and manages player turns, sending updates to both players. The server checks for wins or draws after each move, handles player disconnections, and offers the option to replay after a game ends. Players communicate with the server to submit their moves, and the game continues until a winner is declared or a draw occurs. Overall, it facilitates an interactive game experience between two connected clients.
- The `client_tcp.c` file implements the client-side of a multiplayer Tic-Tac-Toe game. It establishes a TCP connection to the server at a specified IP address and port. The client continuously reads messages from the server, displaying prompts for player moves or replay options. It sends player input back to the server when prompted. The loop exits when the server indicates that the game has ended with a "Goodbye" message. Overall, the client facilitates interaction with the game server, allowing players to participate in the Tic-Tac-Toe game.]
- The header file IP.h has IP defined.
- This udp_client.c implements a UDP client for a Tic-Tac-Toe game. It establishes a connection to the server using a socket and sends a "READY" message to indicate its readiness to play. The client continuously listens for messages from the server, displaying them to the user. If prompted for input, such as a game move or replay option, the client reads the user's input and sends it back to the server. The client will terminate if it receives a shutdown message from the server or encounters an error during message reception. The program ensures proper socket closure before exiting.
- The udp_server.c implements a simple two-player Tic-Tac-Toe game over a UDP server. It initialises a 3x3 game board and handles player connections using sockets. The game alternates turns between Player 1 (X) and Player 2 (O), allowing players to send their moves and broadcasting the updated board state to both players. The game checks for wins or draws after each move, providing feedback accordingly. If the game ends, players are prompted to decide whether they want to play again, and the server handles reconnections or exits as necessary.

NOTE :

- Player 1 is **X** always
- Player 2 is **O** always
- Port used : **8080**

Assumptions for Tic-Tac-Toe Game Implementation

1. **Game Restart Agreement**: The game will only be restarted if both players agree to it. If one player agrees and the other does not, the connection will be closed for both players.
2. **Server Connection Closure**: If the server closes the connection, it results in the closure of the connection for both clients.
3. **Program Termination**: Termination of the program is not explicitly handled within the code.
4. **Turn Management**: If a player inputs a move when it is not their turn, it will not affect the game state or progress.
5. **Network Compatibility**: The game can be run on different devices within the same network, provided that the IP address is correctly updated in the header file (`IP.h`).

2. Fake it till you make it

I made a simple chat application that uses UDP to implement TCP-like functionalities such as Data sequencing and retransmission. Both client and server can send messages to each other.

This Implementation of server.c and client.c handles chunked message transmission with a sliding window protocol, simulating packet loss and acknowledgment behaviour.

Key Logic:

1. **UDP Socket Creation and Binding**:
   - The server creates a UDP socket and binds it to port 8080.
2. **Packet Structure**:
   - The `Packet` struct holds the sequence number, total chunks, and data for each chunk.
3. **Acknowledgment Simulation**:
   - The `send_ack` function sends ACKs for received packets, intentionally skipping every 3rd ACK to simulate packet loss.
4. **Receiving Data**:
   - The server continuously listens for incoming packets.
   - When a packet is received, it concatenates the data to form a complete message and sends an ACK after potentially simulating a delay.
5. **Sending Response**:
   - Once the full message is received, the server prompts for a user response.
   - The response is broken into chunks, and a sliding window protocol is employed for sending these chunks, allowing multiple packets to be sent before requiring ACKs.

6. **Retransmission on Timeout**:
    - The server uses `select` for waiting on ACKs with a timeout. If ACKs are not received, it retransmits the appropriate packets while tracking the number of retransmissions to prevent indefinite attempts.
7. **Buffer Reset**:
    - After completing the sending of a message and its ACKs, the buffer for the full message is reset for the next transmission cycle.

Conclusion:

This code demonstrates the complexities of reliable data transfer over UDP by managing packet sending, acknowledgments, and retransmissions while simulating real-world network conditions such as lost ACKs and packet delays.

Assumptions:

- The first message is always initiated by the **client**.
- The **client** and **server** take **alternative turns** to send messages.
- There is **no option to skip a turn** for either party.
- If either the **client** or **server** terminates the connection, the termination of the other party is **not guaranteed**.
- An acknowledgment (ACK) is **skipped for every 3rd packet** sent.
- A packet will be **retransmitted only a maximum of 5 times** if ACKs are not received.
- The **IP assumptions** remain the same as defined in the previous part.