MiniProject-2 REPORT

Name : V.Akshitha Gayatri
Roll no. 2023101122

SYSTEM CALLS:

GOTTA COUNT ‘EM ALL:

I have a global array syscall_counts that counts the number of times a system call is called.
I added a file sys_names.h that defines the above array.
I defined a function int getSysCount in defs.h and user.h.
In sysproc.c I added a function sys_getSysCount :

```
uint64 sys_getSysCount(void) {
struct proc *p = myproc();
int mask2 = p->trapframe->a0;

int syscall_num = -1;
for (int i = 0; i < 64; i++) {
if (mask2 == (1 << i)) {
syscall_num = i;
break;
}
}
```

```
if (syscall_num == -1) {
printf("Invalid mask: %d\n", mask2);
return -1;
}
int op = syscall_counts[syscall_num];
for (int i = 0; i < 32; i++)
syscall_counts[i] = 0;
if (p->flagg > 0)
printf("PID %d called %s ", p->pid,
syscall_names[syscall_num]);
p->flagg++;
return op;
}
```

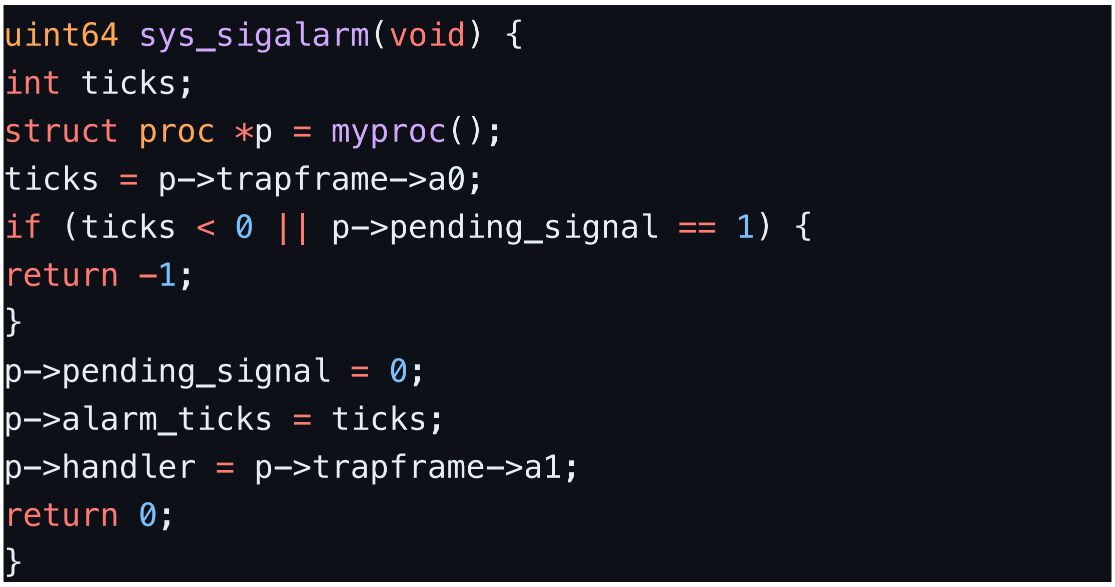
I am initalising the all the elements in the array to zero after each time getSysCount is called.If mask provided by the user is not valid then error statement is printed.

WAKE ME UP WHEN MY TIMER ENDS :

I have added the below in proc.h

```
struct trapframe *backup_trapframe; // to save the current state
int alarm_called;              // to indicate if alarm was called
int interval;                  // interval of the alarm
int pending_signal;           // Flag to indicate if an active alarm
uint64 alarm_ticks;           // Counter for ticks to track the signal alarm
uint64 handler;               // the signal handler function
```

I added two functions sys_sigalarm and sys_sigreturn in sysproc.c



```
uint64 sys_sigalarm(void) {
int ticks;
struct proc *p = myproc();
ticks = p->trapframe->a0;
if (ticks < 0 || p->pending_signal == 1) {
return -1;
}
p->pending_signal = 0;
p->alarm_ticks = ticks;
p->handler = p->trapframe->a1;
return 0;
}
```

If an alarm is pending then return -1 else initialise pending_signal to zero and alarm_ticks to the interval which is given as the 1st argument to sigalarm and handler is the 2nd argument.

```
uint64 sys_sigreturn(void) {
struct proc *p = myproc();

if (p->backup_trapframe == 0 || p->pending_signal == 0)
return -1;
p->pending_signal = 0;
p->alarm_ticks = 0;
memmove(p->trapframe, p->backup_trapframe, PGSIZE); // to
save context
p->trapframe->epc =
p->backup_trapframe
->epc; // to restore the original values of the registers
return p->trapframe->a0;
}
```
saving
I'm using backup_trapframe to save context and to retore the original values of the registers.

I modified the usertrap function in trap.c I added the below code in the else if ((which_dev = devintr()) != 0) block :

```
else if ((which_dev = devintr()) != 0)
{

if (p && which_dev == 2 && p->pending_signal == 0 && p-
>alarm_called == 1)
{
struct trapframe *new = kalloc();
memmove(new, p->trapframe, PGSIZE);
p->backup_trapframe = new;
if (p->alarm_ticks > 0)
p->alarm_ticks--;
if (p->alarm_ticks == 0)
{
p->trapframe->epc = (uint64)p->handler;
p->pending_signal = 1;
}
}
}
```

Basically Iam saving the present trapframein the backup_trapframe and changing pending_signal to 1.

SCHEDULING :

LBS:

I have added the below in proc.h

```
int tickets;            // For lottery scheduling
int arrival_time;     // To record the arrival time of the process
```

In the scheduler function in proc.c I added the code for LBS.

First, the scheduler calculates the total number of tickets by summing the tickets of all runnable processes. It then randomly picks a "winning ticket" from this total. The scheduler loops through all runnable processes to find the one whose ticket range includes the winning ticket. If multiple processes overlap, the one that arrived earlier (determined by arrival_time) is chosen. Once a process is selected, its state is set to RUNNING, and the system performs a context switch to let the process execute on the CPU. After the process finishes or yields, the scheduler resumes to choose the next process.

I added a function sys_settickets in sysproc.c that sets the tickets of a process :

```
uint64 sys_settickets(void) {
int num;
argint(0, &num);
if (num < 1)
return -1;
myproc()->tickets = num;
// printf("Setting %d tickets for process %d\n", num,
myproc()->pid);
return num;
}
```

MLFQ :

I have added the below in proc.h :

```
int ticks_used[NMLFQ];          // How many ticks the process has used at
                                each priority level
int queue;                      // Current queue level of the process
uint64 timeslice;               // Time slice for the current queue level
uint64 time_in_current_queue; // time spent by the process in the
                                current_queue
uint64 wait_time;             // to calculate the ticks a process had to
                                wait to execute
extern struct proc *mlfq[NMLFQ][NPROC]; // struct for MLFQ
```

I have added the below functions in proc.c :
In the scheduler and update_time functions in I added the code for MLFQ scheduling.

1. **enqueue**(int q, struct proc *p):
   - Adds a process p to the multi-level feedback queue (MLFQ) at level q.
   - If the queue is not full (size < `NPROC`), the process is added to the end of the queue, and the queue size is incremented.
   - The process's `queue` attribute is updated to indicate its current queue level.

2. **dequeue**(int q):
   - Removes and returns the process at the front of queue `q`.
   - The process is taken from the first position in the queue, and the remaining processes are shifted forward to fill the gap.
   - The queue size is decremented.
   - If the queue is empty, it returns `0`.

3.**remove_from_queue**(int q, struct proc *p):
   - Removes a specific process `p` from queue `q`.
   - The function finds the process in the queue, removes it, and shifts all processes after it forward to maintain the queue order.
   - The queue size is then decremented.

4. **promote**(struct proc *p):
   - Promotes a process `p` to a higher-priority queue.
   - If the process is not already in the highest-priority queue (queue 0), it is removed from its current queue, its queue level is decreased (moving it to a higher-priority), and it is enqueued in the new queue.

5. **demote**(struct proc *p):
   - Demotes a process `p` to a lower-priority queue.
   - If the process is not already in the lowest-priority queue, it is removed from its current queue, its queue level is increased (moving it to a lower-priority), and it is enqueued in the new queue.

So my logic is I iterate through all the queues in the MLFQ, checking each process's state. If a process is in the ZOMBIE, UNUSED, or killed state, it is removed from the queue. For runnable processes, the scheduler acquires the process's lock, marks it as RUNNING, and switches the context to allow it to execute. After the process runs, it checks if its allocated time slice has expired; if so, it resets its tick count and demotes the process to a lower-priority queue. If a process is SLEEPING, its wait time is incremented, and if it exceeds a defined threshold, the process is promoted to a higher-priority queue.
There is a boost_time of 48 ticks such that afer boost_time all processes are promoted to the highest priority queue.

PERFORMANCE COMPARISON :

MLFQ	: rtime 12,	wtime 148
DEFAULT	: rtime 13,	wtime 149
LBS	:rtime 13	wtime 149

Here as all the scheduling policies were implemented with only 1 CPU I did not find much difference in the performances of the policies.

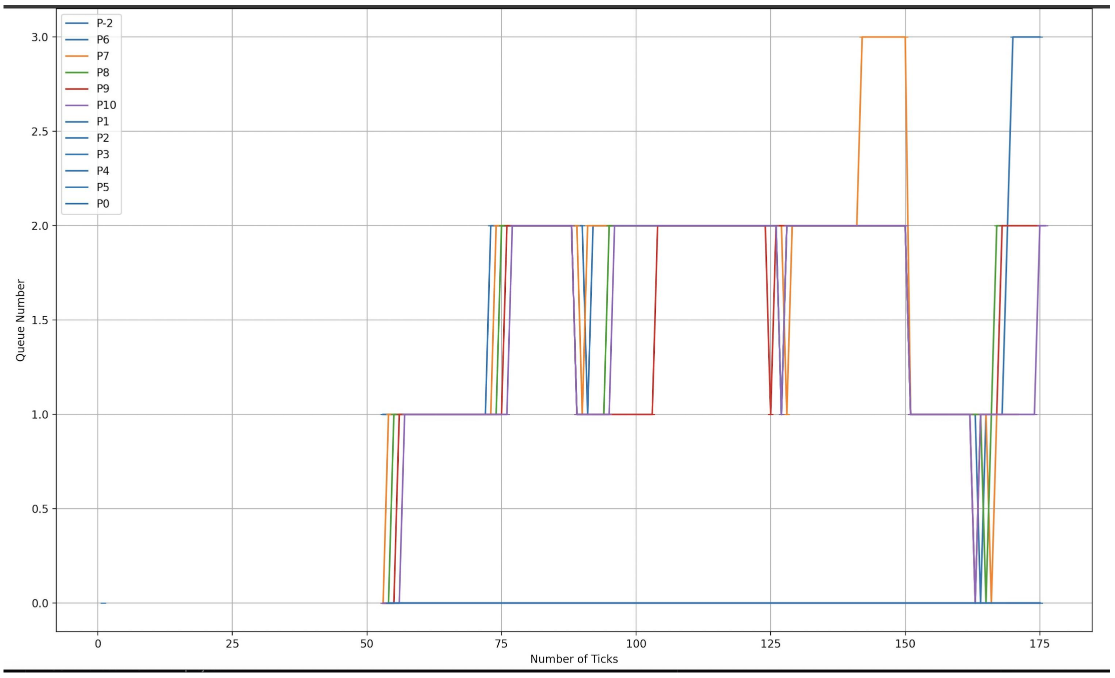
Implications :
The arrival time can be used to break ties between processes with the same number of tickets, ensuring that the process that arrived earlier gets priority.This approach can help reduce the average waiting time for processes, as the scheduler can prioritize processes that have been waiting for a longer period.

Pitfalls to watch out for:
If not implemented correctly, the arrival time can lead to starvation of processes that arrive later, as they may never get a chance to execute if earlier processes continue to hold the CPU.
There might be a chance that a high-priority process is blocked by a low-priority process that arrived earlier.

If all processes have same number of tickets:
The processes that arrived first will be executed sinceas all the processes have same tickets and as the process that arrived first is considered in LBS.

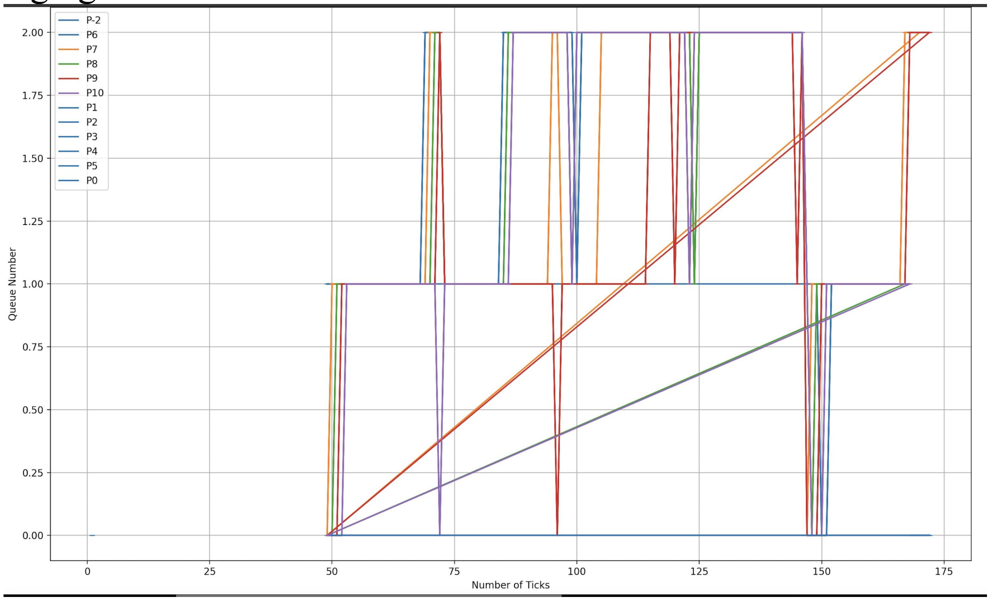
MLFQ Analysis:

Aging time : 30 ticks



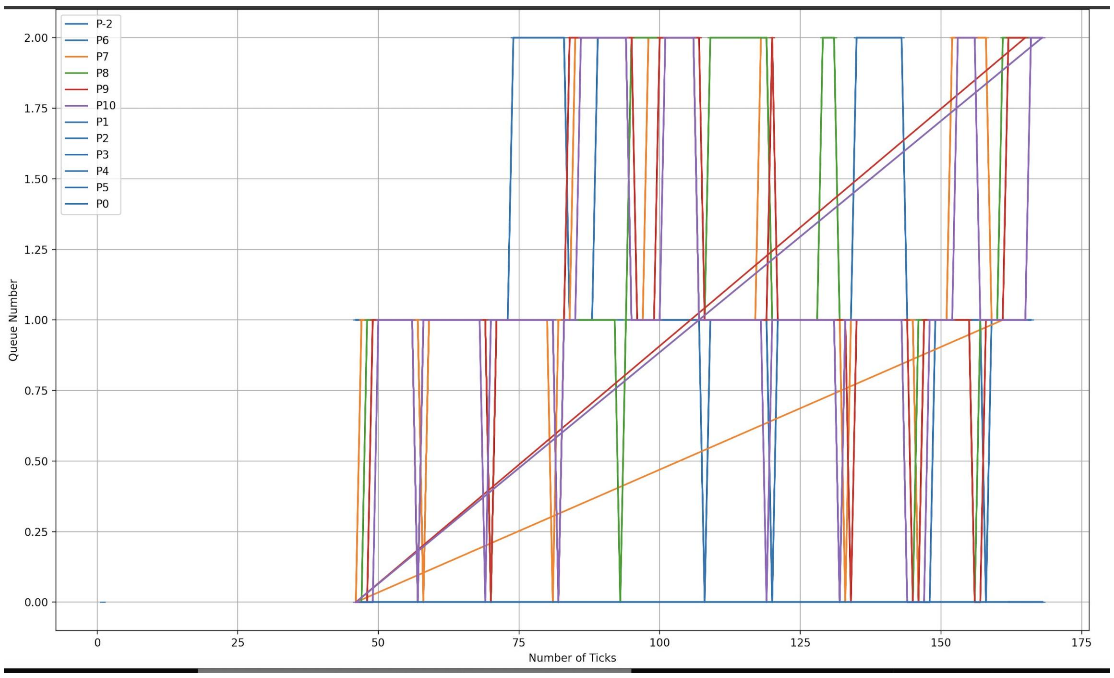
Average rtime 12, wtime 147

Aging time : 20 ticks



Average rtime 12, wtime 146

Aging time : 10 ticks



Average rtime 12, wtime 146

NETWORKING :

1. **XOXO** :

TCP :

The server creates and initializes a TCP socket on port 12345, accepts connections from two players, and manages the game logic, including player turns and board updates.The updated board is sent to both the clients by the server.It checks for valid moves and determines if there’s a winner or if the game ends in a draw, broadcasting the current board state and game results to both players after each turn. After the game concludes, the server prompts both players to decide whether to play again, resetting the board if both agree or closing the connections if they do not. The client establishes a connection to the server on 127.0.0.1 and interacts with the game by reading server messages, displaying the current game state, and sending the player's moves. It also handles prompts for replaying the game, ensuring a seamless multiplayer experience over the network.

UDP :

The server manages the game state, including the 3x3 board, current player, and game logic such as making moves, checking for a winner, and determining if the board is full. When a player joins, their address is recorded, and the server sends the current game board after each move. Clients send their moves to the server, which validates them and updates the game state accordingly. Once a game concludes, players are prompted to decide whether to play again. The server and client communicate using structured messages, and the server handles reconnections and responses based on player input.

2. FAKE IT TILL YOU MAKE IT:

I have created 2 files sender.c and reciever.c
Both the sender and reciever can send and recieve data alternatively.

sender :

The send_data function is responsible for breaking down the input data into smaller chunks and sending them over UDP. It calculates the number of chunks based on the data length and the predefined chunk size. Before sending the chunks, it first sends the total number of chunks to the receiver. Each chunk contains a sequence number and the corresponding data. After sending each chunk, the sender waits for an acknowledgment (ACK) from the receiver, employing a timeout mechanism using pselect. If the sender does not receive an ACK within the specified timeout period, it retransmits the chunk. Additionally, if an ACK is received but does not match the expected sequence number, the sender retransmits the chunk.

reciever:

It first receives the number of expected chunks and prepares to accept them. As each chunk is received, the code checks if it has already been processed using an array that tracks received chunks. If a chunk has already been received, it is ignored, and a message is printed. If it's a new chunk, the data is copied to the output buffer, and an ACK is sent back to the sender. The receiver sends an ACK for chunks , ensuring that the sender receives confirmation of the chunks arrival.