MP-2 Report

Karthik Venkat Malavathlula (2023111025)

October 2024

1 XV6

1.1 System Calls

1.1.1 Gotta Count Them All

In this specification, it is required to count the number of times a system call is counted, including if the system call is called in its child process. I implemented it by

- Creating a global integer array, that will be initialized to 0 before use.
- Calculating the number required by getting number n from mask, where $2^n = mask$.
- By noticing how the processes runs in syscall.c, I increment that element in the array if the system is not shell or kernal.
- At the end I return $array[n]$ from the global integer array.

1.1.2 Wake me up when my timer ends

To implement this specification, I have saved the context before each alarm call to use it in my `sigreturn`. I have created `time` and `interval` parameters in my `proc` struct. I increment `time`, and I send an alarm if

$$(p \rightarrow time)\%(p \rightarrow interval) = 0$$

where `interval` is the time before which I should send the next alarm.

1.2 Scheduling

I have implemented this part by first modifying the make file to define flag LBS and MLFQ. Then in the code, I check if the flags are defined, and if they are, I run that aspect of the scheduling code.

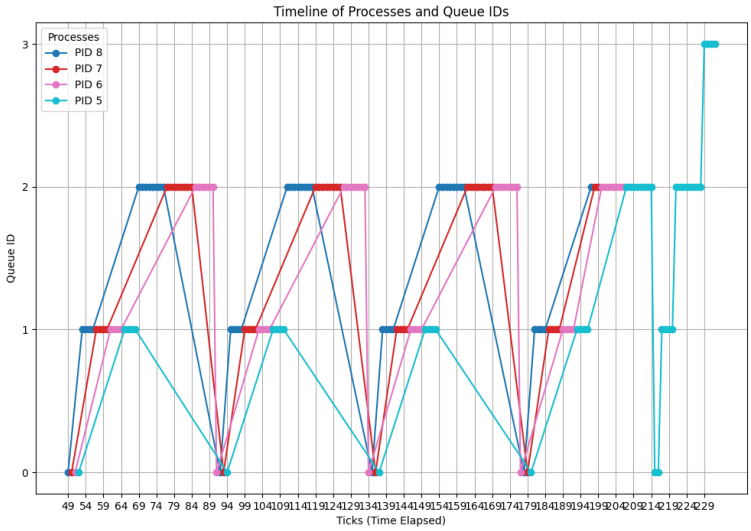
1.2.1 Lottery Based System

To implement this function, I have created the settickets() function call, along with modifying the scheduling function in proc.c. First I calculate the sum of all tickets in all processes. Then, I calculate a random number between 0 and that sum. Then, I iterate across all loops and I check if the current process has more number of tickets than the random number generated. If it has less number of tickets compared it to the random number generated, I subtract it from the random number. Otherwise, I store that process and break from the loop. Then I loop through the processes again, and I see if there is another process that has same number of tickets but created earlier. If there exits such process I replace that process with the processes that has the same number of tickets, but was created at an earlier time. Then I run that process. The settickets() function will allow you to modify the number of tickets initialized in proc, that will be set to 1 by default.

What is the implication of adding the arrival time in the lottery based scheduling policy? Are there any pitfalls to watch out for? What happens if all processes have the same number of tickets?

If we add arrival time, possible implications will be increased complexity and increased fairness. It ensures that processes that arrive earlier are not starved by later-arriving processes with the same number tickets. Common pitfalls to look out for are increase in overhead, as we are looping over the processes again to find the process with same number of tickets but creation time. If all the processes have same number of tickets, It will essentially be a first come first serve algorithm, where the processes are run in the order of arrival time.

1.2.2 MLFQ



My MLFQ works on first come first serve basis on each level. At the last level, it uses round robin to efficiently implement scheduling. I am finding the process to run each time by using a function. The function will return the process with the highest priority and first index. After getting that process, I see the queue of that process. If it is the last queue, I run round robin. Other wise, I will run that function. I have two time parameters in my proc struct, namely mlfqtime and waittime. If waittime becomes 48 ticks, I increase the priority of all processes to the highest priority. Otherwise I check mlfqtime and the time allowed for each queue, its its more that the time that is allowed for that queue, I decrease its priority.

1.2.3 Results of Each Scheduling type

- LBS: Average rtime 17, wtime 93
- MLFQ: Average rtime 18, wtime 93
- Round Robin (default): Average rtime 17, wtime 92

Overall, the 3 types seem to run for about the same amount of time.

2 Neworking

2.1 XOXO

I have implemented in two different protocols, TCP and UDP.

2.1.1 Common

- For both protocols, my implementation to create the board is similar. I have a `print_board` function that prints the board, and I am storing the tic-tac-toe board as a 3x3 integer array in my code.
- Initially, all elements are set to 0. When one player makes a move, the corresponding location is changed to 1 or 2, depending on the player. When displaying the board, these numbers are converted to characters.
- I handle invalid moves by printing the board again to the client and asking them to make a valid move.

2.1.2 TCP

- TCP is a connection-oriented protocol. This means that a connection must be established before data can be sent. In C, this involves using functions like `socket()`, `bind()`, `listen()`, and `accept()` to create and manage the connection.
- I had a form of error handling, as it is essential in TCP. It guarantees the delivery of packets.I used functions like `recv()` and `send()` for data transmission, with checks for successful transmission and handling of re transmissions as needed.
- TCP is more reliable data transfer, ensuring that packets are received in the correct order.
- I have a check winner function that I call each time that will check if player wins by checking if the symbols on the diagonals or rows are the same.
- I have a check draw function that will check if the board is a draw or not.
- I have handle rematch function that will send requests to both players if they want to play again, and if both of them say yes, then I will set the board up for the next game and restart the game.

2.1.3 UDP

- UDP is a connectionless protocol. There's no need to establish a connection before sending data. In C, you only need to call `socket()` and then use `sendto()` and `recvfrom()` for data transmission.
- Because UDP does not guarantee delivery, packets may be lost or arrive out of order.
- UDP is typically faster and has lower overhead.

2.2 Fake it till you make it

In this specification, I have created a UDP server. I have handled the protocol in the following way. For the first message, I have a "first" flag in client that initially set to true. If the first flag is true, I ask for the user input from client and send to server. Then I set that flag to false, and it is a two-way interaction from client and sever every time they read. Whenever a user enters a message from one socket, I have dividing it into chucks and sending each chunk at a time via a struct. The struct will contain a part of the message along with the total number of chucks. Then on the other end, they are stored in a struch array. At the end, I create the final message by combining all the messages into one string.