REPORT

XV6:

1.) Gotta Count Em All:

i.) First we will add a new system (getsyscount) call to ‘syscall.h’ giving a number to it and then we will also map it in ‘syscall.c’.

ii.) Also we will extern the function getsyscount in syscall.c and we will write the function in sysproc.c

iii.) In proc.h, for struct proc we will add a 32 size integer array, to store the number of times each syscall is called in that process. In allocproc()
function in proc.c we will initialise all 32 values to zero’s.

iv.) In syscall.c ,in function syscall we will increment the array[syscall_num] by 1, for current process.

v.) Since we also have to add child processes count, in wait function(proc.c) in we will increment the parent process values with values of the child process for all 32 syscalls , so after every time when wait is completed the parent will get accumulated with total count of all system calls along with its child processes.

vi.) In sysproc.c, getsyscount will just take the arguments and return the count for that system call.

vii.) In user we will create newfile ‘syscount.c’, which will call getsyscount in it after , executing the command.Also,in ‘user.h ‘we will add/declare getsyscount function and in ‘usys.pl’ also we will add it.And Finally we will modify make file by adding ‘syscount.c’ file to it.

2.) Wake Me Up When My Time Ends:

i.) First we will add a new two system calls (sigalarm,sigreturn) to ‘syscall.h’ giving a number to it and then we will also map them in ‘syscall.c’.

ii.) Also we will extern the functions sigalarm,sigreturn in syscall.c and we will write the functions for them in sysproc.c

iii.) In proc.h, for struct proc we will add

```
void (*alarmhandler)();
int alarmticks;
int tickcount;
int in_alarm_handler;
struct trapframe tf_backup;
```

- alarmhandler: Pointer to the function that handles the alarm signal.
- alarmticks: Number of ticks after which the alarm is triggered.
- tickcount: To track the elapsed ticks since the last alarm.
- in_alarm_handler: Flag indicating whether the process is currently inside the alarm handler.
- tf_backup: Backup of the process's trapframe, used to restore the original state after handling the alarm.
-

iv.) In sysproc.c , for sigalarm we will just take two arguments from function , function handler and alarmticks value and set them to the respective fields of proc. Also we will initialise tickcount and in_alarm_handler to 0.

v.) In trap.c, in usertrap when we get a time interrupt , we will increment the tickcount. We keep doing this until,tickcount equals alarmticks. Whemn they are equal, we will store the current trapframe in backup, and we will set in_alarm_handler flag to 1 and we will make sure that next instruction will call handler function.

vi.) In sigreturn, We will get current running process and use the saved trapframe to come back to the current trapframe which contains the CPU registers and process state at the time when the signal was triggered. Also we will set the in_alarm_handler flag to 0, indicating that the process has finished executing the alarm handler.

SCHEDULING:

1.) LBS:

My code implements a lottery scheduler to select a RUNNABLE process based on ticket allocation. It first iterates over the process list to calculate the total number of tickets held by all runnable processes. A winning ticket is randomly chosen, and the scheduler again iterates over the processes to find the one whose ticket range includes the winning ticket. If two processes have the same number of tickets, it selects the one that arrived earlier. Once a process is selected, its state is changed to Running, and a context switch (swtch()) is performed to execute it.

2.) MLFQ:

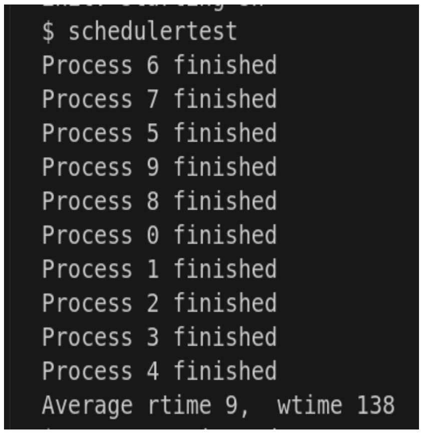
My code implements a Multi-Level Feedback Queue (MLFQ) scheduling algorithm for managing process states in an operating system.

1. **Initialization**: Two arrays, queue_head and queue_tail, are initialized to represent the heads and tails of four priority queues.
2. **Boosting Processes**: Every 48 ticks, the function boost_all_processes_to_top_queue() is called to reset all processes to the highest priority queue (queue 0) with a time slice of 1.
3. **Enqueueing Runnable Processes**: The code iterates through all processes and enqueues any Runnable processes into their respective queues using enqueue_end().
4. **Selecting a Process to Run**: For each queue, it checks the *RUNNABLE* processes and selects the one with the earliest arrival time to run. If a process is selected, its state is set to *RUNNING*, and a context switch occurs to run that process.
5. **Time Slice Management**: After the process runs, its remaining time slice is decremented. If the time slice expires, the process is moved to the next lower priority queue, its time slice is reset based on its new queue level, and its arrival time is updated.
6. **Handling Timer Interrupts**: In trap.c, if a timer interrupt occurs (indicated by which_dev == 2), the currently running process's time slice is decremented. If the time slice reaches zero, it yields control back to the scheduler for context switching.
7. **Updating Arrival Time**: If a process is running and an I/O device interrupt occurs (indicated by which_dev == 1), the process's arrival time is updated to the current tick count, indicating its recent activity.

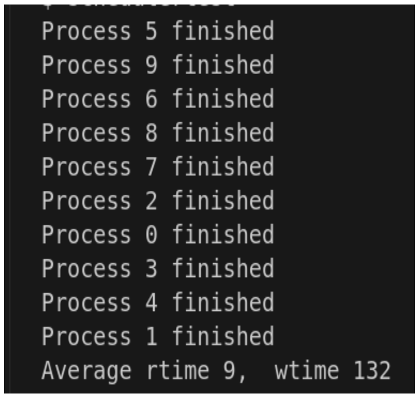
Overall, this implementation allows dynamic adjustment of process priorities and ensures efficient CPU utilization through preemptive scheduling based on process behavior and time slice management.

PERFORMANCE COMPARISION:

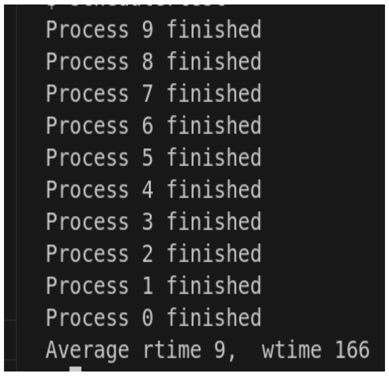
For round-robin


```
$ schedulertest
Process 6 finished
Process 7 finished
Process 5 finished
Process 9 finished
Process 8 finished
Process 0 finished
Process 1 finished
Process 2 finished
Process 3 finished
Process 4 finished
Average rtime 9,  wtime 138
```

For LBS


```
Process 5 finished
Process 9 finished
Process 6 finished
Process 8 finished
Process 7 finished
Process 2 finished
Process 0 finished
Process 3 finished
Process 4 finished
Process 1 finished
Average rtime 9,  wtime 132
```

For MLFQ


```
Process 9 finished
Process 8 finished
Process 7 finished
Process 6 finished
Process 5 finished
Process 4 finished
Process 3 finished
Process 2 finished
Process 1 finished
Process 0 finished
Average rtime 9,  wtime 166
```

LBS (lottery-based scheduler) performs better in the overall comparision

wtime(wait time): 132 (almost near to RR(138), better than MLFQ(166))
rtime(run time): 9 (almost same for all)

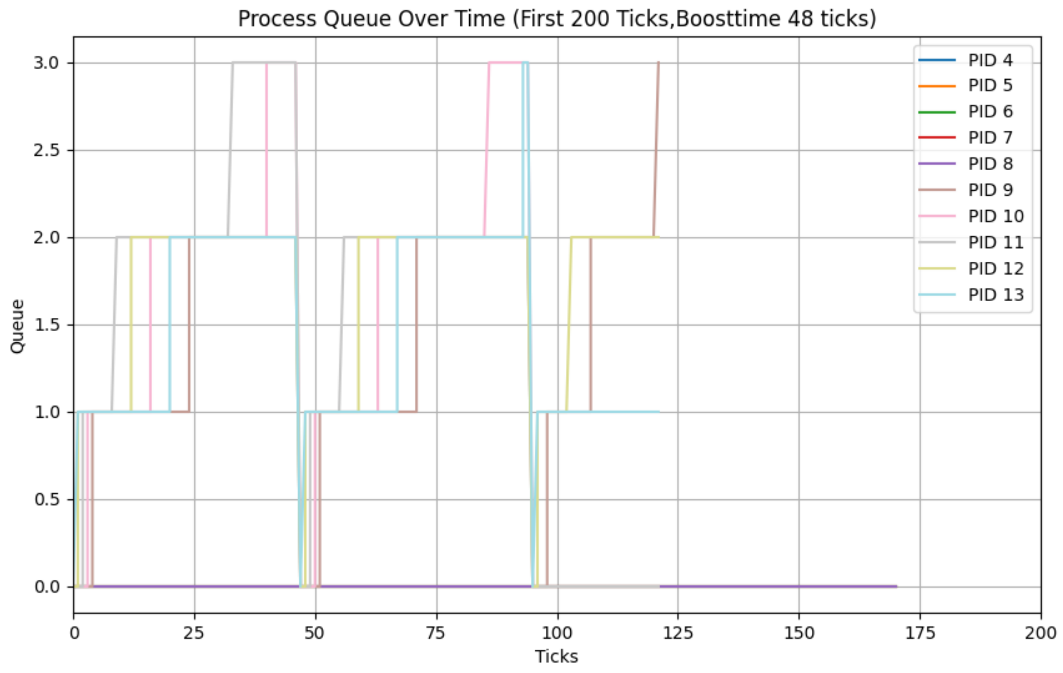
ANSWERS:

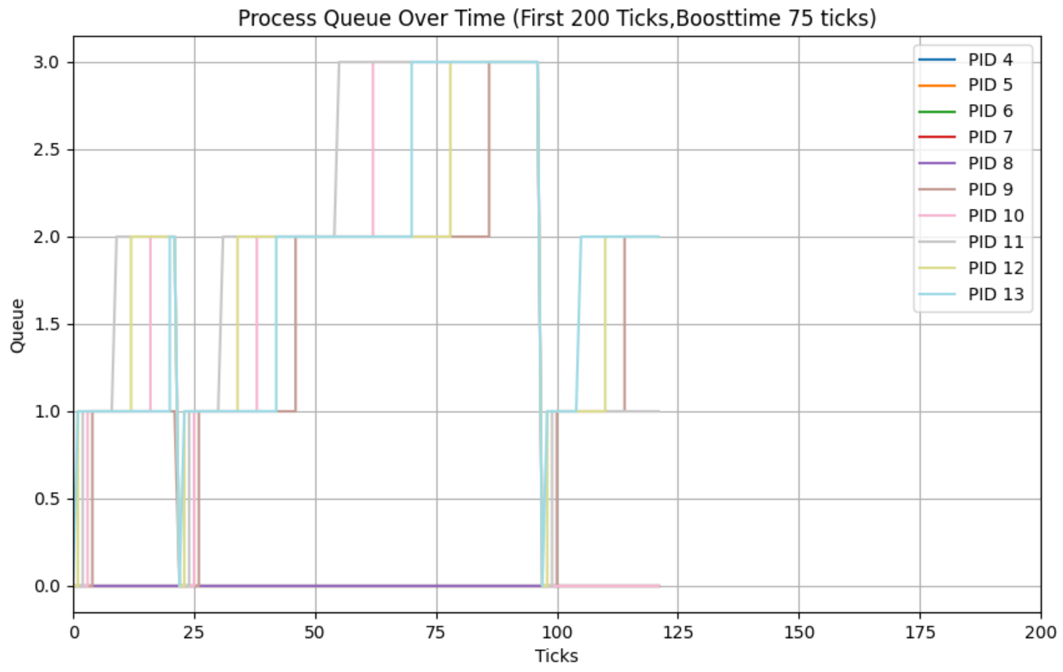
1.) Adding **arrival time** to the lottery scheduler will increase fairness by prioritizing long-waiting processes with equal tickets, reducing starvation risk. It also reduces the random ness of pure lottery scheduling, therefore process selection becomes more predictable.

2.) Pitfalls: In **Lottery-Based Scheduling (LBS)**, problems can include low-ticket processes getting little or no CPU time, leading to **starvation**. If some processes get too many tickets (dynamic allocation), they get more priority reducing fairness as earlier processes have to wait . LBS also doesn't support strict priorities, so important tasks may not run fastly. There's also some extra work involved in picking random winners and checking tickets, which can slow things down in systems with many processes.

3.) If every process have same number of tickets then Our LBS will become FCFS (First Come First Serve) as we are considering arrival_time for selecting, in lottery process .we always check for other process that have same tickets and early arrival_time. As every process have same tickets it will use arrival time as basic factor and process with early arrival time will be selected to run which will be similar to First Come First Serve Scheduling.

MLFQ ANALYSIS:


Process Queue Over Time (First 200 Ticks,Boosttime 48 ticks)
Queue
Ticks
PID 4
PID 5
PID 6
PID 7
PID 8
PID 9
PID 10
PID 11
PID 12
PID 13


Process Queue Over Time (First 200 Ticks,Boosttime 75 ticks)
Queue
Ticks
PID 4
PID 5
PID 6
PID 7
PID 8
PID 9
PID 10
PID 11
PID 12
PID 13