xv6 Scheduling Report

1. *FCFS Scheduling:*
    - We store a pointer called call_time in struct proc which stores sys_uptime() or ticks i.e. number of CPU clock ticks so far.
    - We iterate over all processes stored in the proc table, and select the one with the least call_time.
    - We acquire its locks before changing state and release them after, to prevent another CPU from using the same resources and corrupting the program.
    - FCFS is a non-preemptive scheduling algorithm, hence we will never give up the CPU for one scheduling round. So we disable the yield() function in kernel/trap.c
    - All usertests except preempt work

2. *Lottery Based Scheduling:*
    - We store a pointer called tickets in struct proc which stores the value of the process's tickets (Default value = 1)
    - Also implemented a system call settickets which takes one parameter (new tickets) and sets it to the given proc.
    - The basic logic of this LBS is that it will first sum up the values of tickets of all runnable processes, and then generate a random number from 0 to that sum. Now we take this random number and map it to a particular process.
    - More the number of tickets for the process, higher the probability of it being scheduled by LBS.
    - Eg: If there are 4 processes with 3,7,9,1 tickets each, then the total number of tickets is 20.
      So if the randomly generated number lies between 0-2, we schedule process-1, if it lies between 3-9 we schedule process-2, if it lies between 10-18 we schedule process-3 & if it lies between 19-20 we schedule process-4.
      Thus,
      P(process-1) = 3/20
      P(process-2) = 7/20
      P(process-3) = 9/20
      P(process-4) = 1/20
    - This is a preemptive scheduler, hence we do not disable yielding of the CPU after one round i.e. yield().

- I have used the rand() function in user/grind.c for generating a random integer using a changing seed variable, and modulus with sum of tickets.
- schedulertest.c has a settickets() function call which can be modified to see results.
- All usertests work

3. *Priority Based Scheduler (PBS):*
    - We store pointers called static_priority, niceness, dynamic_priority & sched_count in struct proc.
    - I calculate the highest dynamic_priority (min) among all runnable processes, and store all those processes with such dynamic_priority in an array ‘pties’.
    - If the number of elements in this array are > 1, then we do a tiebreak check. We iterate over ‘pties’ and find the min value of ‘sched_count’ in each (explained later).
    - Now, i store all those processes (in pties with such ‘sched_count’) in pties2, and lastly if elements > 1, then we use FCFS scheduling algorithm to get the process required to schedule.
    - After this process is done with one round of CPU scheduling, we calculate niceness using p->runticks and p->sleepticks, which are 2 variables in proc. (Let p be the struct proc * for the process)
    - When a process begins, runticks is initialised to 0. When this process is scheduled, we initialise 2 local variables beg & end, beg = ticks before process and end = ticks after process. So now we set p->runticks = end - beg, thus for the next time this process is scheduled this is the number of ticks when process was running since it was previously scheduled.
    - In kernel/trap.c, we notice that a process calls the sleep() function when it goes to sleep and wakeup() when it becomes runnable again. So we initialise p->sleepticks = ticks in sleep() & also compute p->sleepticks = ticks - p->sleepticks in wakeup(). Thus, when this process is next scheduled, p->sleepticks is the ticks during which process was sleeping since it was last called.
    - After a process is finished running a CPU round, it takes these new values of p->runticks and p->sleepticks and computes niceness. Using this niceness, we compute the new dynamic_priority for the process.
    - Next time the process is scheduled, PBS will use the new value of DP to check whether to schedule it or not.

- We also have a syscall set_priority(num, pid) which checks the proc table for the given pid and sets the static priority of process with given pid to num. Checked it's working with schedulertest.c
- All usertests work.

4. *Multi - Level Feedback Scheduler:*
    - I have created a queue struct ADT which stores the head pointer, tail pointer, max_ticks and struct proc *array which is the queue intuitively.
    - We store a pointer p->timeslice in proc which is initialised to 0 whenever it enters a queue of any priority.
    - Whenever the the timeslice is >= max_ticks of a particular queue (except last), we shift the process to the next queue in priority.
    - I have also implemented last queue as a normal RR scheduler.

Performance Comparision: (check user/scheduler.c)

Scheduler	**CPUS=1**	**CPUS=3 (Default)**
Round Robin (Default)	rtime = 16, wtime = 165	rtime = 16, wtime = 114
FCFS (w/o IO processes)	rtime = 33, wtime = 156	rtime = 38, wtime = 44
FCFS (with IO processes)	rtime = 17, wtime = 140	rtime = 20, wtime = 108
Lottery Based Scheduler (LBS)	rtime = 17, wtime = 152	rtime = 12, wtime = 117
Priority Based Scheduler (PBS)	rtime = 17, wtime = 224 (for desc priorities) rtime = 17, wtime = 138 (for asc priorities)	rtime = 20, wtime = 107 (for desc priorities) rtime = 20, wtime = 107 (for asc priorities)
MLFQ Scheduler	rtime = 17, wtime = 138	NA

From the above observations,
We can conclude that over one CPU, MLFQ scheduler has the least wait time and most sched policies are better than Round Robin. Run time is similar to each other. (except FCFS w/o IO processes)

Over 3 CPUS, FCFS has the least wtime but high runtime as it is non-preemptive. LBS is preemptive and thus has lower runtime.

We also notice that rtime is generally lower for more CPUS than just one. Also, IO processes take up much more runtime than CPU-bound ones, hence avg runtime is also greater as seen in the 2 FCFS cases.

Majorly, MLFQ Scheduler has least wtime because it is a preemptive policy which has multiple queues, and with aging this could be even better.

Answer to question given in PDF on MLFQ IO processes:
A process could be in some queue and be CPU-bound for as long as the CPU ticks stay lower than the queue's given max_ticks. Now, to prevent it from going to a lower priority queue, this process can call I/O operation(interrupt) right then, so it finishes the op and gets scheduled in the same queue again. Hence, the process exploits this one constraint which is not applicable to CPU interrupts in order to not lose priority in the scheduler.