- The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize turnaround time which is done by running shorter jobs first
- Second, MLFQ would like to make a system feel responsive to interactive users (i.e., users sitting and staring at the screen, waiting for a process to finish), and thus minimize response time
- Turnaround Time vs Response Time: Turnaround time is the amount of time elapsed from the time of submission to the time of completion whereas response time is the average time elapsed from submission until the first response is produced.
- How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without a priori knowledge of job length?
    - The MLFQ has a number of distinct queues, each assigned a different priority level.
    - At any given time, a job that is ready to run is on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e., a job on a higher queue) is chosen to run.
    - more than one job may be on a given queue, and thus have the same priority. In this case, we will just use round-robin scheduling among those jobs. Thus, we arrive at the first two basic rules for MLFQ:
        - Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
        - Rule 2: If Priority(A) = Priority(B), A & B run in RR.
    - Rather than giving a fixed priority to each job, MLFQ varies the priority of a job based on its observed behavior.
        - If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave.
        - f, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to learn about processes as they run, and thus use the history of the job to predict its future behavior

How To Change Priority

- We now must decide how MLFQ is going to change the priority level of a job (and thus which queue it is on) over the lifetime of a job. To do this, we must keep in mind our workload: a mix of interactive jobs that are short-running (and may frequently relinquish the CPU), and some longer-running "CPU-bound" jobs that need a lot of CPU time but where response time isn't important.
    - For this, we need a new concept, which we will call the job's allotment
    - The allotment is the amount of time a job can spend at a given priority level before the scheduler reduces its priority. For simplicity, at first, we will assume the allotment is equal to a single time slice.
        - Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
        - Rule 4a: If a job uses up its allotment while running, its priority is reduced (i.e., it moves down one queue).

        - Rule 4b: If a job gives up the CPU (for example, by performing an I/O operation) before the allotment is up, it stays at the same priority level (i.e., its allotment is reset).
- because it doesn't know whether a job will be a short job or a long-running job, it first assumes it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process. In this manner, MLFQ approximates SJF.

I/O?

- if an interactive job, for example, is doing a lot of I/O (say by waiting for user input from the keyboard or mouse), it will relinquish the CPU before its allotment is complete; in such case, we don't wish to penalize the job and thus simply keep it at the same level.

Problems With Our Current MLFQ

- With MLFQ there is the problem of starvation: if there are "too many" interactive jobs in the system, they will combine to consume all CPU time, and thus long-running jobs will never receive any CPU time (they starve).
- Second, a smart user could rewrite their program to game the scheduler. Gaming the scheduler generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource.
- The algorithm we have described is susceptible to the following attack: before the allotment is used, issue an I/O operation (e.g., to a file) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time.
- Finally, a program may change its behavior over time; what was CPUbound may transition to a phase of interactivity.

Attempt #2: The Priority Boost

- To avoid starvation, we can periodically boost the priority of all the jobs in the system.
    - Rule 5: After some time period S, move all the jobs in the system to the topmost queue.
- The new rule solves two problems at once. First, processes are guaranteed not to starve: by sitting in the top queue, a job will share the CPU with other high-priority jobs in a round-robin fashion, and thus eventually receive service

Attempt #3: Better Accounting

- We now have one more problem to solve: how to prevent gaming of our scheduler? The real culprit here, as you might have guessed, are Rules 4a and 4b, which let a job retain its priority by relinquishing the CPU before its allotment expires.
    - The solution here is to perform better accounting of CPU time at each level of the MLFQ.
    - Instead of forgetting how much of its allotment a process used at a given level when it performs I/O, the scheduler should keep track; once a process has used its allotment, it is demoted to the next priority queue. Whether it uses its allotment in one long burst or many small ones should not matter

- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Without any protection from gaming, a process can issue an I/O before its allotment ends, thus staying at the same priority level, and dominating CPU time. With better accounting in place (right), regardless of the I/O behavior of the process, it slowly moves down the queues, and thus cannot gain an unfair share of the CPU.

Tuning MLFQ

- A few other issues arise with MLFQ scheduling. One big question is how to parameterize such a scheduler. For example, how many queues should there be? How big should the time slice be per queue? The allotment? How often should priority be boosted in order to avoid starvation and account for changes in behavior?
- most MLFQ variants allow for varying time-slice length across different queues. The high-priority queues are usually given short time slices; they are comprised of interactive jobs, after all, and thus quickly alternating between them makes sense (e.g., 10 or fewer milliseconds).
    - The low-priority queues, in contrast, contain long-running jobs that are CPU-bound; hence, longer time slices work well

Summary:

- We have described a scheduling approach known as the Multi-Level Feedback Queue (MLFQ). It has multiple levels of queues, and uses feedback to determine the priority of a given job. History is its guide: pay attention to how jobs behave over time and treat them accordingly
- instead of demanding a priori knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly. In this way, it manages to achieve the best of both worlds: it can deliver excellent overall performance (similar to SJF/STCF) for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads.