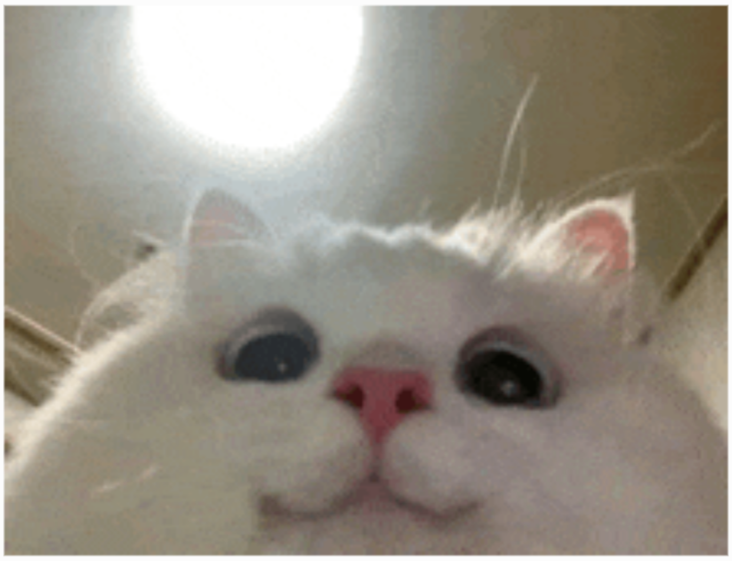
6. Multi-level Feedback



Multi-level Feedback Queue (MLFQ)

Background

1. Optimize turnaround time : running shorter jobs first
   - OS doesn't know how long a job will run for
2. Minimize response time

Opitimize turnaround time <- trade off -> response time

=> HOW to schedule without perfect knowledge?
: both minimizes response time for ineractive jobs while also minimizing turnaround time without a priori knowledge of job length?

Learn from the past to predict the future

- work when jobs have phases of behavior and are thus predictable
- cautious. it can easily be wrong => could be worser than with no knowledge

MLFQ

- Has a number of distinct queues -> each assigned a different priority level.
- uses feedback to determine the priority of a given job
- History is its guide

workload

- interactive jobs that are short-running / frequently relinquish the CPU
- longer-running "CPU-bound" jobs / need a lot of CPU time but response time isn't important

allotment : amount of time a job can spend at a given priority level before the scheduler reduces its priority

Scheduling must be secure from attack

How we guarantee that CPU-bound jobs will make some progress?

: periodically boost the priority of all the jobs in the system.

- processes are guaranteed not to starve
- if CPU-bound job -> interactive : treats it properly.

- The time should be set correctly
  - If too high : long-running jobs could starve
  - If too low : interactive jobs may not get a proper share of the CPU
  - automatic methods based on machine learning

How to prevent gaming of our scheduler?

: once a process has used its allotment, it is demoted to the next priority queue, no matter how many time it performs I/O.
=> Cannot gain an unfair share of the CPU

The high-priority queues

- usually given short time slices
- comprised of interactive jobs

The low-priority queues

- long time slices work well
- containing long-running jobs (CPU-bound)

Decay-usage algorithms

- Decreasing the priority of a job when it acquires CPU time
- Increasing its priority when it doesn't use the CPU.

ADVICE

as the OS rarely knows what is best for each and every process of the system, it is often useful to provide interfaces to allow users or administrators to provide some hints to the OS = advice

- ex) memory manager (madvise) / scheduler (nice)

MLFQ rules

1. If Priority(A) > Priority(B), A runs (B doesn't)
2. If Priority(A) = Priority(B), A&B run in round-robin fashion using the time slice (quantum length) of the given queue.
3. When a job enters the system, it is placed at the highest priority (the topmost queue).
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 (moves down one queue)
5. After some time period S, move all jobs in the system to the topmost queue

MLFQ 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.