SCHEDULER DESIGN

Multi-Level Feedback Queue Scheduler (MLFQ)

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

Our schedueIr is inspired by the MLFQ scheduler but only has two levels.

https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf

SCHEDULER PSEUDOCODE

- Initialize a scheduler backed by a two-level queue
- Compute the Round Robin quantum as the median burst time of all customers
- On each timer tick, do:
  - Feed new customers into the first-level queue
  - Feed the currently playing customer back into the second-level queue if he/she used up one quantum
  - Elevate all customers in the second-level queue to the first-level queue if the elevation timer timeouts
  - Start Serving the next customer if the machine is idle

CUSTOMER CLASS

```
struct Customer {
  const int id;
  int priority;
  const int arrival_time;
  const int burst_time;
  int ticks;
};
```

Worth mentioning:

Priority: high priority 0 or low priority 1

Ticks: total number of time unit this customer has been scheduled

TWO-LEVEL QUEUE

```
std::queue<Customer *> que0;
std::queue<Customer *> que1;
```

que0: waiting queue for high-priority customers
que1: waiting queue for low-priority customers

```
void feed(Customer *c, const bool first) {
  if (first) {
    que0.push(c);
  } else {
    que1.push(c);
  }
}
```

Why queue?

A queue is FIFO (first-in, first out) which exactly reveals the order customers being feeded into the scheduler

When a new customer arrives, it is feeded into the first-level queue no matter of its priority.

Less response time for low-priority customers

When a playing customer is suspended, it's feeded back into the corresponding queue according to its priority.l

Less wait time for high-priority customers

ROUND ROBIN

The scheduler works in a Round Robin way

- Schedule each customer at most one quantum

Why median?

- The mean burst time may be affected by the convoy effect
- Test prooved

```
const int median_index = int(burst_times.size()) >> 1;
auto n = std::next(burst_times.begin(), median_index);
std::nth_element(burst_times.begin(), n, burst_times.end());
quantum = std::accumulate(burst_times.cbegin(), burst_times.cend(), 0) /
          burst_times.size();
quantum /= quantum_factor;
```

SCHEDULE NEXT

```
if (running_customer == nullptr) {
  running_customer = next();
}
```

On each tick, try to schedule the next waiting customer

Inspect the first-level queue first

- Less response time and wait time for high-priority customers

```
Customer *next() {
  Customer *c{nullptr};
  if (!que0.empty()) {
    c = que0.front();
    que0.pop();
  } else if (!que1.empty()) {
    c = que1.front();
    que1.pop();
  }
  return c;
}
```

ELEVATION

```
void elevate() {
  while (!que1.empty()) {
    Customer *c = que1.front();
    que1.pop();
    que0.push(c);
  }
}
```

```
if (do_elevate && elevate_elapsed >= elevate_timeout) {
  elevate_elapsed = 0;
  elevate();
}
```

Elevation: move waiting customers in the second-level queue to the first-level queue

- Avoid starvation

Elevation is performed each time the elevation timer timeout

- Not too frequent, not too lazy

TUNABLE PARAMETERS

```
// higher this, total_wait_0 < total_wait_1.
const double elapsed_factor = 1.55;
// lower this, larger quantum, n_switches less.
const double quantum_factor = 3;
```

```
// true to turn on elevation.
const bool do_elevate = false;
// unclear yet, but useful.
const double elevate_factor = 1;
```

quantum_factor: the Round Robin quantum is calculated from `median burst time * quantum_factor`

elapsed_factor: the two queues are assigned different Round Robin quantums to provide different services to jobs with different burst times.

do_elevate: trun on/off elevation

elevate_factor: the elevation timeout is calculated from `quantum * elevate_factor`

BENCHMARK RESULT

```
  data_1111
my
total_wait_0 total_wait_1 total_wait longest_response n_switches
11768 15015 26783 221 179

ref
total_wait_0 total_wait_1 total_wait longest_response n_switches
12455 13178 25633 296 186
```

The benchmark results for the other four tests have the same pattern of statistics

Out design beats the baseline scheduler on `longest_response` and `n_switches`

But loses on `total_wait`

We have not yet agreed on what attributes to this phenomenon…

In future, we might come up with a systematic way to tune the paremeters and improve our scheduler