Problems w/ `MQ-PRIO`

- *Starvation*: low priority jobs may only run after the OS finished scheduling all high priority jobs. For a runtime environment with neverending stream of jobs of different priority levels, this could result in low priority jobs never getting the chance to run.
- *Deadlock*: if a low priority job holds a lock that a high priority job needs, the high priority job is always scheduled to run despite being unable to compute, while the low priority job never gets to run because high prioirty job exists.

Solution

Both problems stem from the rigid framework under which `MQ-PRIO` operates -- wherein high priority jobs MUST preceed low priority jobs with little room for alteration except via OS or user intervention through other components. The `MLFQ` algorithm alleviates this issue somewhat with the downside of making alterations to each job's priority level on-the-fly -- which could pose problems for OS telemetry.

We propose an alternative to both algorithms: the `Multiple-Queue Priority-Value` algorithm for OS scheduling. We encapsulate each job with a "priority value" that indicates its level of importance the time of this scheduling operation. At each scheduling call, one job from each priority queue is obtained and the task with the smallest priority value is selected for scheduling, forming a "scheduling pool".

If a previously scheduled job is selected again for dispatch, its priority value is incremented according to its priority level -- the higher the priority level of the job is, the less its priority value is incremented so as to retain the hierarchical scheduling process at scale. After this, the job is requeued at the back of its corresponding run-queue similar to the behavior of the Round-Robin algorithm.

After a scheduling pool (one from each priority level) is established, the job with the lowest priority value is selected for dispatch, while all other jobs have their priority value decremented. The higher the priority level of the (non-selected) job is, the more its priority value is decremented to likewise retain the hierarchical scheduling process at scale.

The `MQPV` algorithm, through its leaky priority-value implementation, prevents low-priority task starvation should the supply of higher-priority jobs be continuous and numerous -- low priority tasks are guaranteed to run after a certain scheduling sessions due to continuous priority-value decrease during the previous scheduling sessions, eventually undercutting the priority values of

the higher-priority tasks. At the same time, priority level of each tasks as well as its insinuation towards actual priority in the scheduling process is preserved, as high priority tasks are defined to have slower-rising and faster-dropping priority values than lower-priority tasks.

Analysis

Running a modified `prio-sched-test` program, with "ticker 1" clicking for 8 times and "ticker 2" clicking for 16 times in total, yields the following average run times over 4 trials:

Algorithm	Average Runtime (s; $n = 4$)
`MQPV (adv)`	16.38
`MQ`	22.56
`CFS (default)`	16.61

The `MQPV` algorithm was able to outperform both `MQ` and `CFS` (as implemented in InfOS). We analyze the reasons for this behavior as follows:

- The `MQPV` algorithm is able to outperform `MQ` by a large margin. For `MQ`, priority level differences are strictly maintained, thereby reducing the level of concurrency between long-running high-priority tasks and short-running low-priority tasks. This is especially problematic for our test program `prio-sched-test`, wherein fibonacci tasks are stuck behind ticker tasks, which have superior and insurmountable priority levels.
- The `MQPV` algorithm is able to outperform `CFS` slightly despite seemingly more heavyweight. This is due to data structure constraints -- the naive `CFS` implementation for InfOS is an $O(n)$ scheduler which iterates over all tasks schedulable. Contrastingly, the `MQPV` scheduler, despite iterating over each priority level three times, is roughly constant time in relation to the number of tasks schedulable (assuming that priority levels present in the OS is much lower than number of schedulable tasks). This gives `MQPV` a slight advantage in this situation, and we could expect `MQPV` to be able to further outperform $O(n)$ `CFS` when the number of schedulable tasks become large.

Drawbacks

Memory Use

The `MQPV` algorithm adds additional parameters to each scheduling entity at runtime by at least 3 bytes -- one for priority value, two for increment and decrement deltas -- on top of an 8B-sized pointer to the entity itself (x86-64). Moreover, during the scheduling process, the pooled wrapped entity must be copied from the runqueue due to lack of functionalities for obtaining mutable pointers / references from the API exposed via `list.h`, causing unnecessary duplicated data at runtime.

Time Complexity

The `MQPV` algorithm requires three iterations over each priority levels whenever the next task is to be scheduled. This contrasts the `MQ` algorithm requiring just one iteration over each priority level.

Non-Customizability

Priority values and its related parameters in the `MQPV` algorithm are hard-coded into the algorithm's source code and cannot be altered at runtime.