Some discussion on implementing a Priority Queue scheduler in our simulation

Linux/Unix uses a Priority Queue scheduler, which is not exactly a priority queue (binary sorted heap) data structure. That is to say, the **ready queue** allows processes to have a priority (usually a modest number of priorities often <100), so it is a queue with priorities. Say 20 priorities. (processes also have a "nice" factor – more later). If there are 1000 processes in the ready queue, and if evenly distributed over the 20 priorities, that would be 50 processes per priority. [does not seem to match well with a binary priority queue heap implementation, though that concept could be adapted.]

With the priority ready queue(s) (see, I changed the terminology for clarity), there is also the question of ordering within each priority – are processes in FIFO order or ordered in some way? They could be ordered by the "nice" factor, if the scheduler is to key on that item in its decision algorithm. We shall simply keep the items at each priority in FIFO order, and the scheduler will do a round-robin rotation in scheduling within each priority.

Then there are implementation decisions: is this one big ready queue organized by priority; or a set of FIFO queues, one for each priority level. The **implementation mechanism** must support the **scheduling policy**. Various mechanism ideas can be examined for efficiency, and a choice made. For this simulation experiment, the mechanism work required of the computer to run the simulation is not paramount. So that allows some flexibility in implementation mechanisms, to allow us to experiment with different scheduling policy ideas. If a particular scheduling policy is then shown to have advantages, then a study of optimal implementation mechanisms could be made.

The simulation structures evolved from the MLFQ simulations, so having separate queues for each priority level was a simple adaptation, and is part of the structure given to you. When a process's "nice" value exceeds the threshold, then the process drops down to the next priority (and is removed from one queue and added to the next). This inspection and decision would occur after the completion of a CPU quantum (not CPU burst completion), when the process must be context switched from the CPU back to a ready queue anyway – so here is the point to decide whether the process drops down or not.

So what about the "nice" value? That is the number of times a process burst is allowed to cycle into the CPU before dropping to a lower priority. This is one of the simulation input values, so that we can explore the effect of alternative nice thresholds (it looks like this should be less than 10 to avoid processes "stacking up" in the same priority.)

With the MLFQ scheduling, each process returns from doing I/O back to the top level queue, and can descend from there. What does the UNIX priority scheduling queue do in this case? 1) It could do the same, or 2) it could return a process to the queue it was in when its CPU burst completed (and it left the ready queues to do I/O). (2) would require maintaining/persisting the last queue data in the process object (which is like the Process Control Block). **This scheduling decision will be explored in the simulation, and is an existing simulation input parameter to be implemented later.**

Other decisions will also come later, we (in this class with our experiments) will tend toward simple policy ideas that also have simple implementation mechanisms.

1) How much time is allocated to each priority level CPU quantum? The current mechanism uses a 2X multiplier (4,8,16,32 etc). Is that the correct policy? Unknown at this time. This could be changed easily to explore alternatives, perhaps adding the base level (4, 8, 12, 16, 20, 24…) for each queue (likely and easy).
2) What is the policy in servicing the different level queues (priorities)? MLFQ is clear and easy – all higher priority queues must be empty (starvation possibilities). The policy MUST favor real-time processes, **and** favor SJF (shortest CPU burst first). Perhaps a returning process should start again at the highest priority (like MLFQ) in order to support RT and SJF. The simulation could explore alternatives, but returning to the highest priority supports SJF and RT and also tends to avoid starvation at the lowest priorities (so we will probably just implement this idea). … Even so, that does not answer the question of when to service each priority level. We could work it like MLFQ. Or, I am pondering a scheduler that uses a simple mathematical relationship, where X processes are serviced in the queue above, before one process is serviced in the queue below. If X=2, then for instance, 2 processes might be serviced in Q1, and then one in Q2. If X is very large, the system behaves similar to the MLFQ. Perhaps there is an optimal value. With adaptation for servicing RT processes as highest priority. –TBD. **Policy questions 1) and 2) are related in determining behavior and performance in meeting the policy imperatives of RT, SJF, and avoiding starvation.**
3) Starvation mitigation? TBD. The MLFQ system I have collected data on and published, has a variable mechanism to mitigate starvation. The idea above might be sufficient to mitigate starvation – or we may need to add something more. UNIX/LINUX periodically boosts processes back up to the top level. Is that actually best? For instance perhaps processes are periodically boosted up Y levels (maybe just Y=1)? And what is the period for boosting processes up – and is this tracked per each process, or does the scheduler boost all on a fixed period. AND, do we ONLY boost up processes in the lowest priority, or ALL processes? Simplest is to just boost processes in the lowest priority, periodicity to be determined, and number of levels up TBD.

Lots of interesting ideas to explore!