Workload, job, application, process (All terms refer to the same meaning in this document)

Motivation for MLFQ

- STCF and RR based policies optimize for either turnaround time or response time but not for both
- Assumption #5 (Knowing the process runtime beforehand) was not removed
  - This assumption helped us classify workloads as interactive or batch jobs and thus helped with scheduling

MLFQ (Multi-Level Feedback Queue)

- Optimize on both turnaround time and response time
- Without knowing beforehand whether a job is interactive or batch

Structure of MLFQ

- Multiple queues and each of them at different priorities.
- Process from highest priority is chosen (always)
- Round robin among processes within same queue
  - The above two points result in the following rules
    
    **Rule #1**: Pri(A) > Pri(B) ==> A runs
    **Rule #2**: Pri(A) == Pri(B) ==> A and B runs with RR policy

Why do we need multiple queues?

- If we could classify workloads properly then interactive process would receive higher priorities compared to batch process
  - Low usage (Interactive) - keep them at higher priority
  - High usage (Batch) - keep them at lower priority
- Assuming we could classify workloads, MLFQ could provide both response time and turnaround time

How to classify workloads?

- Use dynamic priority assignment
  
  **Rule #3**: New process entering the system gets highest priority (Assumption: all jobs are short jobs and so, receive highest priority)

  **Rule #4a**: If the process uses its entire time slice, then demote to next lower priority
  **Rule #4b**: Gives up CPU before time slice, then stay in the same queue

What are the problems that could arise with the current design of MLFQ?

- Starvation problem
  - Too many interactive jobs in the system and batch job doesn’t get to run
o Change in phase of an application is not identified
o Design flaw: We saw decrease in priority till now (but no boost)
  o Fix: Priority boost

**Rule #5: Move all process from the lower priority queue to highest priority queue after some period of time**

- Gaming problem
  o Workload could issue I/O and give up CPU just before time slice to stay in the same queue
  o Design flaw: Proper accounting is not done
    o Fix: Rewrite rule 4a and 4b

**Rule #4: After allotted time slice is used at a given level, moved to bottom level**

Parameters in MLFQ

- Number of priority queues
- Time interval to boost priority
- Time slice or schedule quantum

**Questions**

1. MLFQ puts new process in the highest priority queue. You could fork a new process at the end of time slice (assume fork is cheap) and thus continue execution since fork basically clones the creator including the instruction pointer. The new process would end up getting into the highest priority queue. How could you avoid this gaming?

2. At the end of a schedule quantum, the policy has to decide which process to run next. What is the worst case latency to pick a process? How can you optimize it?

3. All examples dealt with single CPU in the system? What are the challenges with designing a scheduler when the system could have multiple cores?