ID: 23301569, SECTION: 24

Fairness Issues of CPU allocation in the MLFQ scheduling algorithm

---

1. Unfair CPU Allocation Due to I/O-Bound Processes

i) Dynamic adjustment of priorities in MLFQ based on CPU-bound and I/O bursts times:

As we know till now, the Multi-Level Feedback Queue (MLFQ) scheduling algorithm is the most optimized and dynamic algorithm for both CPU efficiency and system responsiveness. It allows scheduling between a group of processes where the ready queue is divided into several queues based on their priority, and processes can move between queues. Here, each queue has its own time quantum. Higher priority queues take a lesser time quantum to run early, be interactive and fast, whereas lower priority queues take a longer time quantum for long run tasks. These priorities are adjusted by some measurements. Here's how it works: without any I/O request and execution, if a process spends its full time quantum, then the scheduler puts it in the lower priority queue, thinking it needs a long time quantum/CPU bursts(CPU-bound). On the other hand, if a process ends before its time quantum, then it voluntarily releases the CPU and the scheduler promotes it to the higher priority queue, considering it an I/O bound. That’s how this algorithm dynamically allocates priorities by learning from past behaviors.

ii) Reason why I/O-bound processes tend to monopolize higher-priority queues:

We know that in this algorithm, higher-priority queues always execute first, no matter what. If a single process is not executed in the higher priority queue, the lower priority queue in the next options won’t get the CPU. Thus, all processes of higher priority must be completely executed before going down to the lower priority queue. Now I/O-bound gets quick access to the CPU, and processes give up the CPU voluntarily every time before the time quantum expires, taking less time and therefore retaining higher priorities. Meanwhile, CPU-bound processes continuously consume the whole time quantum, resulting in being pushed down to the lower

priority over time. Thus, I/O-bound processes tend to monopolize higher-priority queues, preempting the CPU-bound ones.

iii) Starvation and unfair CPU allocation due to these imbalance:

As discussed earlier, we saw how I/O bound is constantly dominating CPU-bound, creating a space for unfairness. This disparity sometimes leads to a state where low-priority (CPU-bound) processes wait indefinitely because higher-priority processes always occupy the CPU called starvation. In the crowd of interactive systems or calls, what needs to be fast(I/O-bound tasks), long-running computational tasks might rarely get a chance to be scheduled, reducing throughput and fairness. Just think of system of where a process of small editor/terminal(I/O-bound) keeps calling for CPU, and a background compiler, which is CPU-bound in a lower queue, may never get scheduled for CPU even though the task was necessary. MLFQ lacks in this core fairness issue, starving CPU-heavy tasks.

iv) Fairness-Improving Mechanisms:

Operating systems use different strategies to deal with the aforementioned unfairness:

1. Aging: Processes with lower priority that wait too long in the lower queue get their priority increased over time. Let's say if a process is waiting in the lower queue for more than a constant time without getting CPU time, OS will increase their priority which will lead them to be in the upper queues with higher level processes.

2. Priority Boosting: Many system like windows and linux use this priority boosting to periodically run a priority boosts over total system. Here, unlike aging we don't take the processes one by one to increase their priority instead, after a fixed interval, all processes are shifted to the highest priority queue to have a chance to run at high priority occasionally and never let any process being permanently stucked at the bottom.

3. Adjusting Time Quantum: When we talked about time quantum we knew, after time quantum expires, each processes need to release the CPU and let the next lower level process hold it. Now in multi-level scheduling, among queues time quantum plays a huge role to balance between responsiveness and fairness, such that increasing the quantum time helps me reducing preemption and CPU-bound processes complete longer bursts giving me fairness whereas if the time quantum was small it would make the system responsive but at the same time it would have increased preemption and favored I/O-bound processes. A balanced one would print out the best result.

2. Quantitative Fairness Evaluation

i) Concept of fairness and its importance in CPU Scheduling:

When I have several types of competing processes to be scheduled, I must equally distribute CPU time to everyone of them. That's what fairness in CPU scheduling means, and a scheduler does the job of ensuring it that nobody gets over-served and nobody gets rotted from starving. Now in a multi-queue system, we have seen how there can be a place for infairness. The balance is hard to maintain as the algorithm prioritizes responsiveness over equality. Thus, a quantitative approach is necessary to distribute fairness.

ii+iii+iv) Fairness Index: A Quantitative Metric:

Here, if the value of FI = 1, it means a perfectly fair distribution has been done. All processes got equal CPU time. If the value is close to 0, it indicates the distribution is highly unfair. Let's see the formula:

$$Fairness\ Index\ =\ (\ (\Sigma\ T_i)^2\ )\ /\ (\ n \times \Sigma\ T_i^{\ 2}\ )$$

Here, T_i = Allocated CPU time to process i

n = number of processes

Let's measure fairness in a sample MLFQ workload:

Suppose we have four processes: P1, P2 (Short, I/O bound tasks), and P3, P4 (long, CPU-bound). We distributed 600ms among these 4 processes,

P1 = 200 ms, P2 = 200 ms, P3 = 100 ms, P4 = 100 ms.

F.I. =$(600)^2$ / (4×100000) = 0.9; indicating moderate unfair distribution, where we can see I/O-bound tasks are getting more CPU share.

With aging: P1 = 160 ms, P2 = 160 ms, P3 = 140 ms, P4 = 140 ms

F.I. = $(600)^2$ / (4×90400) = 0.995; fairness improved, almost perfect distribution.

With priority boosting: P1 = 150 ms, P2 = 150 ms, P3 = 150 ms, P4 = 150 ms

F.I. = $(600)^2$ / (4×22500) = 1.0; perfectly distributed.

With too many queues, short quantum: P1 = 220 ms, P2 = 220 ms, P3 = 80 ms, P4 = 80 ms

F.I. = $(600)^2$ / (4×109,600) = 0.82; fairness decreased, proving that if you prioritize short tasks, it wrongs CPU balance.

From this, we get that if we increase aging or priority boosting, it will increase the chance of fair distribution, but the aging interval must be small in that case. More queues reduce fairness unless balanced by boosting; a short time quantum gives faster response but reduces fairness.