REDUCING MLFQ SCHEDULING STARVATION WITH FEEDBACK AND EXPONENTIAL AVERAGING*

Kenneth Hoganson, Ph.D.
Kennesaw State University
1000 Chastain Road, Kennesaw, Georgia
Department of Computer Science and Information Systems
770-499-3402
khoganso@kennesaw.edu

ABSTRACT

The Multi-Level Feedback Queue (MLFQ) for process scheduling is efficient and effective, but can allow low-priority and CPU-intensive processes to be starved of CPU attention and make little progress. Process scheduling algorithms in general favor the "shortest CPU-burst first" approach, which provides good performance for interactive processes. A disadvantage often cited for the MLFQ is that processes in the lowest priority queue are in danger of being starved for CPU attention. Other scheduling techniques involve periodically increasing the priority of a process to enable the process to make some forward progress.

This paper presents a technique that mitigates MLFQ starvation, which is low overhead and does not jeopardize the servicing of interactive and high-priority processes. This is accomplished by adding a second level of feedback to redirect a "safe" amount of CPU time to the lowest-priority queue to prevent starvation of processes in that queue.

1.0 INTRODUCTION

Multi-tasking operating systems strive keep multiple processes in the computer's main memory in order to keep the processor highly-utilized. This requires that an operating system include a CPU scheduling algorithm to determine how CPU time will be allocated among the available processes. Favoring shortest CPU-Burst processes

---

allows quick response time for interactive processes, while making overall efficient use of CPU time.

When the rate of arrival of new processes and high priority processes is large enough, low-priority and CPU-intensive processes can receive little or no CPU time, essentially "starving" those processes of CPU time. Any strategy that favors the allocation of CPU time to short, interactive, and high-priority processes has a risk of potential starvation of computation-intensive processes. Some operating systems that use priority queues address this problem by periodically increasing the priority of the starved and low-priority processes, so that they receive some CPU attention. These temporarily bumped-up processes will gradually have their priorities reduced back to their long-term level. Temporarily increasing the priority of a process results in the CPU being switched to service that process more often, resulting in additional context-switching overhead [1]. In addition, increasing a process's priority results in increasing memory requirements, as the process's working-set of pages must now be loaded into memory, potentially forcing pages needed for higher-priority processes to be swapped out. These pages will need to be swapped back into memory as the temporarily bumped-up processes returns to its original lower level, potentially causing a short-term memory-page-thrashing. The priority-queue solution creates the potential for lost CPU time in performing additional context switches and additional page-swapping overhead and delays.

The Multi-Layer Queue (MLQ) scheduling strategy utilizes multiple separate process queues, where each queue is scheduled individually. Processes are permanently assigned to one queue. The Multi-Level Feedback Queue (MLFQ) improves on this concept by dynamically allowing processes to move between queues. Each queue allocates a difference quantum of CPU time, graduated from the first queue to the last [5], [6]. So moving a process between queues is analogous to changing a process's priority.

Processes in a MLFQ are able to "find their own level" through a feedback mechanism based on their need for CPU time. Processes that need a large amount of CPU time migrate to the lower priority queues, while I/O bound and interactive processes remain in high priority queues [5]. Ensuring that interactive and I/O bound processes receive CPU time maximizes system performance and overall best use of system resources (CPU, memory, and I/O). An drawback to the MLFQ strategy is that processes that have "settled" into the lower-level queues may be starved of CPU time. Other research has investigated approaches to real-time processing and efficient context switching in MLFQ [1], [2]. Others have added intelligence to enhance the fault-tolerance of MLFQ implementations [3], or to mitigate starvation using neural network processing [4].

This paper presents an enhancement to the MLFQ CPU time-allocation strategy that adds a second level of feedback, one that periodically diverts time to low priority queues in order to prevent starvation, without jeopardizing the performance of the high priority queues.

2.0 THE MULTI-LEVEL FEEDBACK QUEUE

Figure 1 illustrates a Multi-Level Feedback Queue consisting of five queues (Q1-Q5) that make up the operating system's "ready queue" of processes available for CPU-execution.

Processing bursts enter in Queue 1, which is both the highest-priority queue, and also the queue that allocates the shortest burst of CPU time to processes (in this example, 16ms). A processing burst in Q1 either completes its burst, exiting the ready queue to a wait queue, or completes and exits the system entirely, or is incomplete and "falls-down" to Q2.

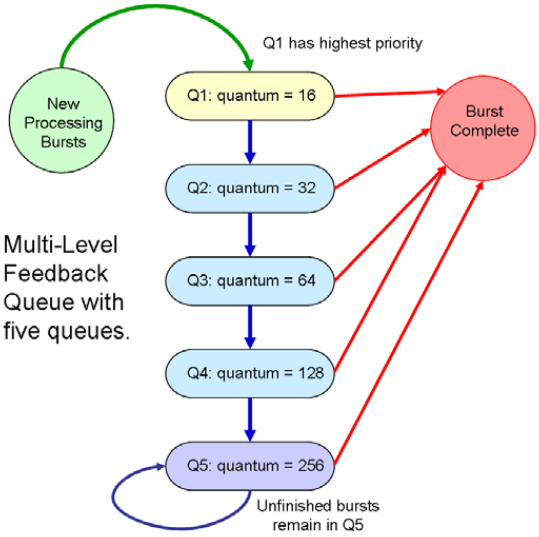


Figure 1- Multi-Level Feedback Queue

Q2 only receives CPU processing time when Q1 is empty. Q3 receives processing time only when both Q1 and Q2 are empty. Similarly, the last queue, which is also the lowest priority queue, only receives CPU time when all higher-priority queues are empty. This creates an opportunity for processes in Q5 to receive no CPU time when the workload is high (as a percentage of 100% CPU processing capability).

In the MFLQ, CPU time is allocated to queues on a non-interruptable basis for normal processing. Interrupts and critical services can interrupt current processing. For example, if queues Q1, Q2, and Q3 are currently empty, the CPU is allocated to processing work in Q4. If a new processing burst then becomes ready for execution in Q1, it will not be serviced until the current burst being processed in Q4 either completes or the entire Q4 time quantum is consumed. For either of those events, the system will check Q1 (and Q2 and Q3 as appropriate) for work in those higher-priority queues. This then results in a context-switch to service the new processing bust that arrived in Q1.

3.0 STARVATION ANALYSIS IN THE MLFQ

Starvation in a MLFQ scheduled system will only occur when the workload arrival rate exceeds the processing capacity of the system, at least temporarily. During this period, the CPU will be fully utilized servicing processes in queues with high priority. Given a system with a process arrival rate λ, and an average process CPU burst time requirement of $T_{ave\text{-}burst}$ over some period of time then,

$$\lambda * T_{ave\text{-}burst} > T_{period}$$

When the workload exceeds CPU capacity for only a short period, starvation is temporary and not a significant problem: the system will give priority to the interactive and high-priority processes in the high-priority queues as intended. Low priority processes will be delayed until the workload arrival declines, in order to favor interactive, short, and high priority processes, exactly as desired for optimal scheduling.

When the workload exceeds CPU capacity for an extended period of time, then the system is clearly overloaded, and starvation and lack of progress will occur for processes regardless of the scheduling algorithm. The only available decision is which processes to starve? Obviously the lowest-priority processes should be starved of CPU time.

The last case to consider is when the arriving workload matches or exceeds CPU capacity for a significant but limited period, and the long-term workload pattern is not overloaded. Under these conditions, it may be undesirable to allow low-priority processes to be completely starved for CPU time and unable to make any progress. In this situation, a diversion of CPU capacity to low-priority processes may be required.

An enhancement to the MLFQ scheduling algorithm that addresses starvation by periodically reallocating CPU cycles to low-priority processes will only be utilized when low-priority process starvation is occurring, while at the same time high-priority and interactive processes are being adequately serviced. In this situation, the middle priority processes are consuming all remaining CPU time not required by the high-priority queue. The proposed MLFQ enhancement safely diverts time from the middle-priority processes to the low-priority processes to prevent complete starvation there. This approach does not significantly impact that the high-priority and interactive processes.

4.0 MLFQ SOLUTION IMPLEMENTATION

The proposed solution to starvation in a MLFQ diverts CPU time from the middle-priority queues to the low priority queue. This involves analysis of how much time to divert of what period, and includes a bound that guarantees that the high priority queue will be unaffected in the case of a temporary spike in demand.

The key constraint is the requirement that high-priority and interactive processes must not be delayed beyond acceptable limits. The workload arrive may have temporary spikes, and during those spike times no CPU time should be diverted. This requires a feedback mechanism to dynamically and automatically determine when it is safe to reallocate time, and to allow sufficient reserve to accommodate unpredictable spikes in demand.

Given a MLFQ with ***n*** queues, consisting of:

Q1: the highest priority queue. All arriving processes pass through this queue. Interactive processes are serviced by this queue without falling down to the lower level queues.

Q2 ... Qn-1: these are the middle priority queues. Some processes will work through these queues and then exit the system, while others will pass through these queues on their way to the lowest-priority queue.

Qn: the lowest priority queue.

4.1 Reallocation Time Quantity

CPU Time will be reallocated from *Q2 ... Qn-1* to allocate to *Qn* to prevent starvation. *Q1* is consuming T_{Q1} time. The algorithm will track CPU usage of *Q1* over some interval of time T_{period}. When *Qn* is being starved ($T_{Qn} = 0$) then

$T_{period} - T_{Q1}$ = *time left for Q2 ... Qn-1* = T_{Avail}

T_{Q1} is a function of the arrival rate of processes into the ready queue and the average CPU time required in *Q1* (which is bounded by the time quantum the MFLQ allocates to *Q1*), determining T_{Avail}. A portion of T_{Avail} is diverted to service *Qn*.

Exponential averaging is used to track T_{Avail} as the foundation for servicing *Qn*. Exponential averaging is a very low computation-overhead algorithm, and very low memory storage algorithm, that is used to "average" or track the time used by *Q1*. This average will be tracked and computed over time interval *M*, each interval of T_{period} in duration:

$T_{Ave(m)} = T_{Ave(m-1)}$ * *weight-factor* + $T_{Q1(m)}$ * (1 - *weight-factor*)

Because exponential averaging generates a trailing average (the average trails any trend increasing or decreasing trend in usage) and because of the need to allow a buffer of time to accommodate surges and spikes, and to allow some time to service processes in *Q2 ... Qn-1*, only a fraction of T_{Avail} should be reallocated to *Qn*. That fraction ($T_{\%}$) is a configuration variable whose optimal value is dependent upon the dynamically varying workload on the system. Since changes in *Q1* workload will be tracked by T_{Ave}, $T_{\%}$ need not be adjusted often. For instance, if $T_{\%}$ = 40%, then 40% of T_{Ave} will be allocated to *Qn*:

$TQn = T_{Ave(m)} * T_{\%}$

4.2 Scheduling of Processes in Qn

The final scheduling consideration is how to allocate the time to the processes (previously starved for CPU time) in Qn. A single block allocation of time to Qn minimizes overhead due to context switching, but lengthens the time between Q1 servicing. The simulation results allocated time as a single block.

5.0 SIMULATION RESULTS

A five-queue MLFQ was simulated as both an un-enhanced MLFQ and an MLFQ enhanced with time reallocated from queues 2-4 to queue 5. Each queue allocates CPU time to a processing burst for up to its assigned queue quantum The process burst either completes within that time, or is passed down to the next queue. Processing bursts in Q5 remain in that queue until complete.

The workload was simulated by creating new processing bursts randomly:

55% of bursts were uniformly distributed between 1 and 16 milliseconds (representing short and interactive processing bursts).

44% of bursts were distributed between 16 and 256 milliseconds, with an average of 77 milliseconds. The enhanced MLFQ will divert processing time from these processing-bursts to mitigate starvation in Q5.

1% of bursts are uniformly distributed between 256 and 1256 milliseconds, with an average of 756, representing long processing bursts that "settle" down to Q5, and are in danger of starvation in the unenhanced MLFQ.

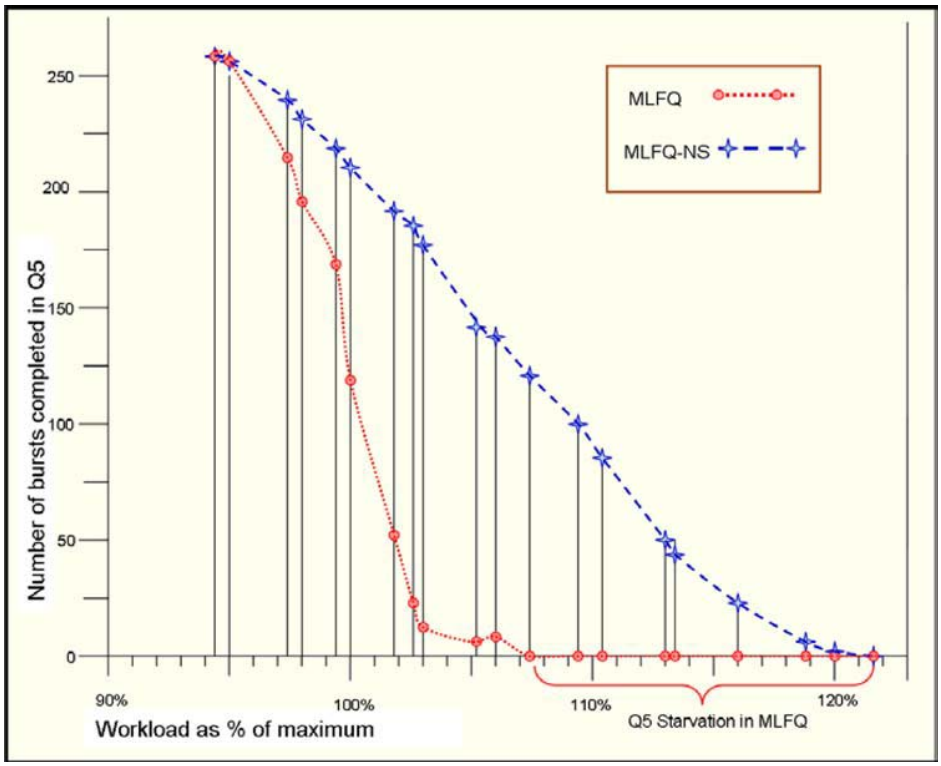
The workload magnitude was varied by adjusting the process inter-arrival rate, to acquire data on performance under workloads varying in their percentage of 100% of CPU processing capability. That percentage varied from about 94% to 121% of maximum CPU processing capability. The time consumed by executing workload, doing context-switches, and CPU-idle times was tracked. All bursts arrive in Q1 and either complete or fall-through to the next lower queue. Context-switching overhead-time was set at 1millisecond per context switch.

The enhanced MLFQ diverts work on a 1000 millisecond period. The time consumed by Q1 for processing work (excluding context-switching overhead) is counted. The remaining processing time in that period may be consumed by queues Q2-Q5. Up to 10% of this non-Q1 remaining time is diverted to service Q5 and prevent complete starvation of processing bursts in that queue.

In simulating this workload, the processing bursts entering into Q1 consumed an average of 32% of the total processing time. 10% of the remaining time resulted in an average diversion of about 68 milliseconds from queues Q2-Q4 to Q5. This time was allocated in a single burst to Q5, running bursts in that queue without interruption. Data generated by the simulation is presented graphically in Graph 1.

Graph 1 correlates the number of processing bursts completed in Q5 over workloads of varying percentages of 100% CPU processing capability. The MLFQ line is the un-enhanced MLFQ exhibiting starvation, and the MLFQ-NS is the enhanced algorithm. For workloads 95% of max CPU and less, the CPU was able to complete all processing bursts, no starvation occurred, and CPU-idle times grew.

Complete starvation of bursts in Q5 was observed in workloads over 100% of



Graph 1. Queue 5 Burst Completion with Varying Workload

maximum processing capability in the un-enhanced MLFQ, with those bursts receiving zero processing time. Partial starvation was observed for workloads representing greater than 97% of maximum processing capability.

The enhanced MLFQ with workload diversion effectively mitigated starvation in Q5, for workloads of up to 120% of CPU capability. Beyond 120% of CPU capability, starvation of long CPU bursts was so complete, that no processes made it through queues 1-4 into Q5 for processing. Effectively, the starvation at this heavy overload level occurs above Q5 in Q4.

6.0 CONCLUSIONS

This paper presents an enhancement to the Multi-Level Feedback Queue CPU time allocation strategy, by adding a second level of feedback: a mechanism that allows the allocation of time to the lowest-priority queues in order to prevent starvation. The method presented does this without jeopardizing the performance of the high priority queues.

Simulation results demonstrate and confirm that this MLFQ enhancement effectively mitigates starvation in the last queue, for large workloads (as a percent of 100% CPU capability). Simulation results also confirm that this starvation mitigation is accomplished without compromising the priority of Queue 1, which runs high-priority and interactive processing bursts.

7.0 REFERENCES

[1] L.A. Torrey, J. Coleman, B.P. Miller “A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler”, Software: Practice and Experience,Vol. 37, No. 4, Pg. 347-364, 2007, John Wiley & Sons, Ltd.

[2] P. Goyal, X. Guo, H.M. Vin, “A hierarchial CPU scheduler for multimedia operating systems”, Proceedings of the second USENIX symposium on Operating systems design and implementation, p.107-121, October 29-November 01, 1996, Seattle, Washington, United States

[3] M.R. EffaParver, K. Faez, “An Intelligent MLFQ Scheduling Algorithm (IMLFQ) with Fault Tolerant Mechanism, Sixth International Conference On Intelligent Systems Design and Applications (ISDA) 2006, Vol. 3, pp 80-85.

[4] M.R.E. Parvar, S. Safari, “A Starvation Free IMLFQ Scheduling Algorithm Based on Neural Network”, International Journal of Computational Intelligence Research.

[5] F. J. Corbato, M. M. Daggett, R. C. Daley “An Experimental Time-Sharing System” IFIPS 1962.

[6] Andrea Arpaci-Dusseau, “Multilevel Feedback Queue Scheduling in Solaris” Available: http://pages.cs.wisc.edu/ eli/537/lectures/Solaris.ps