Spring 4-20-2017

On Intelligent Mitigation of Process Starvation In Multilevel Feedback Queue Scheduling

Joseph E. Brown
Kennesaw State University

On Intelligent Mitigation of Process Starvation In Multilevel Feedback Queue Scheduling

Master of Science in Computer Science
Thesis

By

Joseph E Brown
MSCS Student
Department of Computer Science
College of Computing and Software Engineering
Kennesaw State University, USA

Submitted in partial fulfillment of the
Requirements for the degree of
Master of Science in Computer Science

November 2016

On Intelligent Mitigation of Process Starvation In Multilevel Feedback Queue Scheduling

This thesis approved for recommendation to the Graduate Council.

Kennesaw State University

College of Computing and Software Engineering

Thesis Title: On Intelligent Mitigation of Process Starvation In Multilevel Feedback Queue Scheduling .

Author: Joseph E Brown.

Department: Computer Science.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Ken Hoganson	Date
Thesis Reader: Dr. Jose Garrido	Date
Thesis Reader: Dr. Selena He	Date
Thesis Reader: Dr. Edward Jung	Date
Department Chair: Dr. Dan Lo	Date

Director of the Graduate School:

Dean: Dr. John Preston	Date

Dedication

To Willy Gommel.

Acknowledgments

People I intend to acknowledge: Xie, Hoganson, Bryant, Gayler,the College,...

List of Tables

List of Figures

Abstract

CPU time-share process schedulers for computer operating systems have existed since Corbato published his paper on the Compatible Time Sharing System in 1962 [8]. With this new type of scheduler came the need to effectively divide CPU time between N processes, where N could be 2 or more processes. Modern time-sharing process schedulers which have been developed in the decades since have been designed to favor shorter, interactive processes over long-running processes, especially when incoming demand for CPU time exceeds supply and process starvation is inevitable. These schedulers, including Linux CFS, FreeBSD Ule, and the Solaris Fair Share Scheduler, are all effective at favoring interactive processes under starvation conditions.

Sometimes it's not desirable that long-running processes be sacrificed altogether, but none of these schedulers have safeguards under starvation conditions. This thesis revisits and extends the research conducted in [13], in which it was demonstrated that starvation of long-running processes could be safely and effectively mitigated without adversely affecting the performance of shorter, interactive processes.

The questions this thesis will answer are:

1. Can MLFQ-NS, proposed in [13], be compared to other modern process schedulers?
2. Can MLFQ-NS be improved?

To answer the first question, a scheduler must be found which is similar enough to MFLQ for a direct comparison. This will require a survey of current schedulers. To answer the second question, the research conducted in [13] must be duplicated MLFQ-NS to ascertain the following:

1. How much diverted time is actually used?
2. Why does MLFQ-NS become ineffective past a certain system-load threshold, i.e. stop reallocating time to long-runnning processes?

In this research, the original work was duplicated in simulations to validate previous results, and determine why MLFQ-NS became ineffective after incoming CPU time demand exceeds a threshold. Research was conducted in order to determine if starvation mitigation in MLFQ-NS could be compared to other process schedulers used in production, with the conclusion that recent emphasis on priority scheduling and heurstic interactivity determination makes such a comparison impossible. Research then continued with simulations in which MLFQ-NS was given different runtime arguments than original simulations. Investigations into those results led to an algorithmic modification to MLFQ-NS called MLFQ-IM and analysis of simulations of MLFQ-IM. Conclusions about the effectiveness of MLFQ-IM will be explored. Finally, ideas for future research are offered.

Contents

List of Acronyms

CFS Complete Fair Scheduler. The default Linux kernel process scheduler, as of Linux kernel version 2.6.27.

CTSS Compatible Time Sharing System. One of the original CPU time-share schedulers.

FCFS First Come First Serve. Is a service policy which dictates that clients are served in order in which they arrived.

FIFO First In First Out. Is a data buffer insertion and removal policy that dictates data are removed in the order in which they were inserted in a data structure.

MLFQ Multilevel Feedback Queue. Process scheduler and ready queue consisting of multiple prioritized internal PCB queues. MLFQ is implemented in various computer operating systems.

MLFQ-IM MLFQ-Intelligent Mitigation. An extension of MLFQ-NS.

MLFQ-NS MLFQ-No Starvation. An extension of MLFQ which mitigates starvation of long-running, low-priority processes during periods of incoming high CPU demand.

PCB Process Control Block. Ready queues store PCBs, which represent processes to an operating system. PCB contain process state information.

ULE FreeBSD SCHED_**ULE** Scheduler. The default FreeBSD kernel process scheduler, as of FreeBSD 5.2.

Introduction

Since the introduction of CPU time-share process schedulers in 1962 for the IBM 7090 computer system [8], there has been a long evolution of CPU time-sharing schedulers for computer operating systems. This has culminated in the development of several modern schedulers, including FreeBSD ULE [16], Linux CFS [14], and the Solaris Fair Share scheduler [17] [1] which is based on the Multilevel Feedback Queue (MLFQ) scheduler [21].

With the advent of CPU time-sharing came the need to effectively divide and allocate CPU time between a potentially large number of processes. As one process receives time to execute on a CPU core, other processes wait for their turn in a ready-queue [11]. So that processes may complete in a reasonable amount of time, incoming CPU time demand must not exceed compute system capacity. If there are K units of CPU time available in some time period T, then incoming demand CPU time demand D must not exceed K during T, i.e., $D \leq K$. This assures that all processes will be allowed to execute and complete within a reasonable amount of time.

When incoming compute demand *does* exceed capacity, the compute system is considered to be overloaded. If overload occurs, starvation of some processes is possible, such that they may not complete within a reasonable time. Because of this, starvation should be addressed.

All of the schedulers mentioned have mechanisms which favor shorter, interactive processes in case overload occurs. ULE and CFS heuristically identify and schedule interactive processes ahead of lower priority processes, and dynamically adjust CPU burst times when incoming demand exceeds a threshhold. MLFQ naturally favors shorter, interactive processes by internal ordering. However, they don't address starvation of lower priority, long-running processes during overload. Whereas starvation mitigation under prolonged overload is impossible, it's been shown that it can be mitigated in MLFQ for limited duration overload [13].

Chapter 1

Previously Published MLFQ Redirection Results

This chapter briefly reviews MLFQ, including its history and evolution. It discusses previous research and simulations conducted with MLFQ. It discusses process starvation, and how it may be identified. Finally it'll review recent research on starvation mitigation.

1.1 MLFQ

Multilevel Feedback Queue scheduling is an evolution of the Compatible Time-Share System, which was first described in [8] as a multiuser CPU timeshare scheduling system, utilizing a multilevel process queue. CTSS was designed to coordinate multiple users running one process each, and was one of the first CPU timeshare scheduler implemented. It was designed to favor shorter, interactive processes over longer, lower-priority processes, and to gradually degrade latency in case of high CPU demand while still favoring interactive processes. MLFQ is an enhancement to CTSS which changed where processes are placed in the multilevel queue when they first arrive, and the number of users is largely irrelevant.

As suggested by the name, an MLFQ scheduler consists of a set of two or more FIFO process queues, and a scheduling policy. Associated with each level in the MLFQ is a maximum time quantum that each process is limited to running within before it must release the CPU. Queues store Process Control Blocks (PCB), each of which represents a process to the operating system, keeping state data about the associated process and maintaining references to memory locations. From this

point on, descriptions of processes in the context of entering and leaving queues are with respect to their associated PCB.

The set of process queues may be visualized in a vertical arrangement, as shown in Figure 1.1. The queue on top is called Q_1, and the bottom queue is called Q_N. All incoming processes are first enqueued into Q_1 while they wait for their turn to consume CPU time in FCFS order. The process is dequeued when its turn comes, and executes for a time no greater than the quantum associated with its queue. If the process has not completed, it's re-enqueued into the next lower queue to wait for the next CPU time quantum to be granted. Otherwise it may release the CPU prior to quantum expiration because it requires IO service. In this case, it enters a wait-queue for IO, and will return to Q_1 once that service is finished. Once the process reaches Q_N it's re-enqueued into Q_N as many times as necessary, with the exception that IO is requested. This sequence of enqueue, dequeue, execute, re-enqueue or request IO continues until the process has finished.

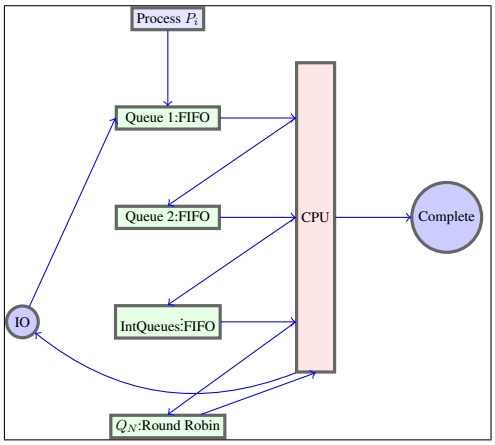

Figure 1.1: An MLFQ Implementation

The MLFQ scheduling policy is straightforward. To dequeue the next available process, the set of queues is checked in a top-down strategy, starting at Q_1. If there are no waiting processes in Q_1, then Q_2 is checked. If there are no waiting processes in Q_2, then the next queue is checked. To summarize, a process waiting in $\mathrm{Q}_{j\geq 2}$ cannot be selected unless $\mathrm{Q}_{\{1\ldots i\}}$ are all empty.

MLFQ doesn't differentiate between different types of processes, and processes aren't inherently prioritized; they naturally find their way to the queue appropriate for their behavior [23]. MLFQ is a nonclairvoyant scheduler, as scheduler decisions are made independent of the characteristics of the schedulable candidates [20]. MFLQ can be modeled as an advanced case of a Tandem Queue with Sequential Service Switching independent interarrival and service times, and a single server (CPU) [15].

1.1.1 Starvation in MLFQ

Under heavy load, while the MLFQ still prioritizes CPU bursts of short duration in Q_1, the MLFQ can disfavor processes that are enqueued in the lowest priority queue. Under extreme circumstances of very heavy processing load, CPU-intensive processes can be temporarily starved, for a short or potentially long-term period, since the scheduler always selects processes from the higher priority queues. These processes may be unable to complete within a reasonable amount of time. Ideally, while favoring shortest CPU burst processes first, an operating system process scheduler must ensure that all processes may make progress. Mathematically, this can be described as follows, where

$$\lambda * T_{\overline{burst}} > T_{Period} \tag{1.1}$$

Incoming load may exceed compute capacity for brief, significant, or extended periods of time. This research is concerned with the second case, when the system could self-correct but starvation of low-priority processes would be undesirable for the period of time required.

In a system with infinite computing capacity, i.e., possessing unlimited CPU cores, each queue would have at most one waiting process at any moment in time. Since processing capacity is never infinite, starvation can be identified by a build-up of processes in one or more queues, and a decline of CPU quanta given to processes in those queues. Usually starvation will occur in Q_N, because all other queues must be empty before processes in Q_N may receive CPU time.

1.1.2 Requirements for Starvation Mitigation in MLFQ

Starvation is most likely to occur in low-priority queues, or Q_N, which are serviced the least frequently. To mitigate starvation in Q_N, time must be redirected from other queues, without compromising performance in Q_1 and thrashing from excessive context switching.

Q_1 performance is observed via mean process wait-time, which is the intervening period of time between enqueuement in Q_1 and the first CPU burst given to the process. Mean wait-times mustn't increase significantly, or mitigation comes at the price of interactivity and high priority processes.

1.2 Literature Search

Significant research has been conducted to minimize Q_1 latency and maximize overall throughput in MLFQ. Duda experimented with allowing known-interactive threads to borrow against future

CPU allocation time to be scheduled sooner than they otherwise would be [9] [1]. Behera proposed dynamic CPU allocation per process according to system load [5], and similarly Parvar proposed IMLFQ, which changes the number of queues and associated quanta in MLFQ as system load varies [10]. Thombare proposed replacing FCFS algorithm in $Q_2, \ldots, Q_{N-1}$ with SJF to increase throughput of the processes with the shortest runtime [22] [1].

Less research has been conducted to address process starvation in Q_N under high system load. Bhunia proposed a variation to re-enqueue processes from Q_N in higher priority queues according to the amount of CPU time still required by those processes [6] [1]. Raheja, Dadhich and Rajpal claimed to have resolved the issue of starvation in Q_N altogether with VMLFQ, similar to IMLFQ, using *vague set theory* to calculate the optimum number of queues and CPU burst quanta sizes [19] [1] [2]. Hoganson proposed reallocation of time from $Q_2, \ldots, Q_{N-1}$ to address starvation in Q_N through the use of a moving average [13]. Of these two proposed variations, only the one proposed in [13] is directly implementable in real-world systems. The proposed variation in [6] requires a clairvoyant scheduler, which doesn't exist outside of simulation systems.

1.2.1 Scheduler Review

Part of the purpose of this research is to compare MLFQ-NS with other schedulers, with respect to starvation mitigation. In order to compare the performance of schedulers, they must be similar enough. In this case, we're measuring bursts given to processes in Q_5. The sections that follow describe various schedulers, and vet them for comparison.

1.2.1.1 Solaris Heuristic MLFQ Scheduler

Sun Solaris used MLFQ scheduling since version 2.5 [17] [2] [3]. MLFQ is used for its Time Share class of processes, and it uses 60 queues (0-59). It mitigates starvation in lower queues by using a timer to boost the priorities of processes in queues 1-59 approximately once per second [4]. A process in Q_{59} would be moved to Q_{58}, where theoretically it has a better chance of receiving a CPU quantum. CPU usage history is tracked per process in 1 second intervals, without past intervals contributing to present data [1].

[1] *A priori* knowledge of process requirements and characteristics is required, which isn't available in real-world systems [12] [4].

[2] Starvation is mathematically impossible to eliminate where Equation 1.1 is true.

[3] Dr. Arpaci's CV may be found here: `http://pages.cs.wisc.edu/~remzi/cv.pdf`

[4] There is no higher priority queue than 0

Process priority is implemented as a property of the process itself, and not tracked by the scheduler. Thus the priority of a process is retrieved from the process itself, and the process moves between the queues upon enqueuement, depending on its CPU usage history. A process which voluntarily relinquishes the CPU and sleeps frequently is considered interactive, and its priority is heuristically calculated. For that reason, the process may move up or down in the queue hierarchy.

Whereas the Solaris scheduler bears some striking resemblance to the MLFQ studied in this research, there are some differences which prevent a proper comparison.

1. Solaris uses heuristics to mark processes as interactive, whereas the MLFQ variant being studied doesn't differentiate interactive processes.

2. Solaris processes are explicitly assigned priorities, which can then change. Our processes aren't assigned priorities at all; priority is implicitly associated with the queue.

1.2.1.2 FreeBSD ULE Scheduler

The FreeBSD ULE scheduler is an example that uses multiple scheduling policies and process classes. Classes include real-time, system, timeshare (user), and idle[5]. Real-time, system and *interactive* timeshare processes are considered high priority, timeshare are mid-priority, and idle processes are low priority.

It uses three sets of process queues, and each set of queues has its own scheduling policy. High priority processes are stored and retrieved via MLFQ, mid-priority via CalendarQ[6], and low via MLFQ as well. When the ULE scheduler is called to retrieve the next process PCB to load, the queue sets are searched in this order [18] [16]:

1. High-priority from $\text{MLFQ}_{\text{High}}$, [7]

2. Timeshare from $\text{Calendar}_{\text{Mid}}$,

3. Idle from MLFQ_{Low} .

Similarly to the Solaris OS, priority is an inherent property of a process and therefore not implicit to the queue level in which the process is waiting. Depending on the class and behavior of

[5] "Idle" processes are those which are run when there are no high priority processes to run. Idle processes perform various OS chores, such as "zeroing-out" pages of memory which have be deallocated after some process finished with them.

[6] More informatin about calendar, also known as *circular* queueing, can be found in [7]

[7] The terms $\text{MLFQ}_{\text{High}}$, $\text{Calendar}_{\text{Mid}}$, $\text{Calendar}_{\text{Mid}}$, and MLFQ_{Low} **DO NOT** appear in ULE literature. They are used here to delineate and simplify organization.

the process, that priority can be dynamically adjusted higher or lower. In the case of a timeshare process, that priority is heuristically adjusted when as necessary, based on a ratio of voluntary sleep time to runtime.

Whereas both internal MLFQ schedulers in ULE resemble the MLFQ studied in this research, the same differences prevent a proper comparison.

1. ULE uses heuristics to mark timeshare processes as interactive, whereas the MLFQ variant being studied doesn't differentiate interactive processes.

2. FreeBSD processes are explicitly assigned priorities, which can then change. Our processes aren't assigned priorities at all; priority is implicitly associated with the queue.

3. The MLFQ in this research has *all* processes,regardless of priority, entering a single MLFQ structure at the same place: Q_1. The ULE scheduler has multiple structures, each for a different set of priorities. Processes are enqueued according to their priority, not necessarily in Q_1.

1.2.1.3 The Linux Completely Fair Scheduler

The Linux process scheduler has taken several different forms over the course of its evolution[14], a couple of which use a dual run queue variant[24]. The most recent scheduler [8], called the Completely Fair Scheduler, or "CFS", was designed with focus on CPU timesharing fairness and interactivity. This is accomplished, in part, by using nanosecond CPU time accounting, and dividing CPU time between processes as evenly as possible.

CFS is a departure from traditional schedulers in that process PCBs are not stored in FIFO queues, but a red-black sorted [9] binary search tree (BST) of "schedulables". Schedulables may be processes, groups of processes, or even "nested" run queues [25]. An interesting feature enabled by such organization is the ability to group processes and treat them similarly, thus allowing processes to spawn groups of processes with equivalent priorities. This tree of schedulables is sorted by CPU time previously granted to schedulables.

As with Solaris and FreeBSD, priority is a property of a linux process. It may be heuristically adjusted to reflect the interactivity of a process. That is where the similarity ends. Using priori-

[8] only fully SMP-compliant schedulers are considered here. There are others which shall remain unnamed, with are not scalable to an arbitrary number of CPU cores.

[9] Information about Red-Black Binary Search Trees may be found in [3]

ties in a BST that is sorted by CPU time received per-process would seem counterintuitive. The explanation cannot be more elegantly stated than this:

> CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay, where higher-priority tasks have lower factors of delay. This means that the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task. That's an elegant solution to avoid maintaining run queues per priority. [14]

Torrey, Coleman and Miller[23] compared interactivity performance between CFS and MLFQ by re-implementing MLFQ in a linux kernel. In those experiments interactive processes were heurstically identified, and their initial wait-times were recorded. Their results indicated MLFQ and CFS have comparable performance.

The binary search tree structure of the CFS scheduler is structurally different from the MLFQ scheduler, and has no equivalent to Q_1 or Q_5 . Because of this, there can be no comparison of Q_1 latencies, or bursts in Q_5 . Thus CFS isn't suitable for comparison with MLFQ-NS in this research.

1.3 MLFQ-NS

In [13] simulations were used to study process starvation in 5-level Multilevel Feedback Queue (MLFQ) schedulers, and an extension to MLFQ was developed to mitigate starvation under certain conditions. This extension to was called MLFQ-NS, where "NS" means "No Starvation". The goal of MLFQ-NS is to divert time from intermediate queues $Q_{2...N-1}$ to Q_N without compromising performance (or increasing wait-time latency) for high-priority and interactive processes. This is explained in the following sections.

1.3.1 Reallocating Time

The goal of MLFQ-NS is to mitigate starvation in Q_N without compromising performance of interactive and high-priority processes in Q_1 . Time is reallocated from Q_2...Q_{n-1} to Q_N to mitigate starvation. This is accomplished by leveraging exponential averaging with time-tracking for CPU bursts given to processes in Q_1 over some period of time T_{period}. Let

- tracked time given to processes in Q_1 be T_{Q1}
- time given to process in Q_n be T_{Qn}

- time given to processes in the intermediate $Q_{2...n-1}$ be T_{avail}
- reallocation percentage of T_{avail} be $T_{\%}$
- time reallocated to Q_n be T_{Qn}
- weight-factor be $\alpha \leq 1$

Under starvation conditions, $T_{Q_N} = 0$, which means that

$$T_{avail} = T_{period} - T_{Q1} \tag{1.2}$$

T_{avail} is used to calculate an exponential moving average for period T_M by combining the previous average with new data, as shown in the function from [13]:

$$T_{ave(m)} = T_{Ave(m-1)} * \alpha + T_{Q1(M)} * (1 - \alpha) \tag{1.3}$$

and reallocated time is then calculated as

$$T_{Qn} = T_{\%} * T_{ave(m)} \tag{1.4}$$

It should be noted that no heuristics are used to determine that starvation should be mitigated. This is because at less than full CPU utilization, there should be nothing waiting in Q_N[10] At the transition of time periods from T_m-T_N, the new T_{Q_N} is calculated, and that time is granted to processes waiting in Q_N the next time the scheduler is called to dequeue a process.

1.3.2 Original Simulations

The operational parameters and results of the original simulations from [13] are shown in Table 1.1. The CPU time requirements for jobs were generated from the statistical information in it. Processes in queues 1-5 were allotted maximum CPU time quanta shown in Table 1.2 , respectively. After each CPU burst granted to a process, a millisecond of CPU time was consumed for context switching.

IO activity was not simulated. The research goal was to focus on CPU time reallocation to Q_N and its affects on Q_1 latency in isolation, with intent to extend simulations with IO, or even experiment with MLFQ-NS in a live operating system. The simplified MLFQ without IO can be seen in Figure 1.2.

[10]There are exceptions to this which will be discussed later in this work.

$\mathbf{P_n}$	**Req.:mS**	**Dist.**
n=55	1-16	$\cup(1, 16)$
n=44	16-256	$\cup(16, 256)$
n=1	265-1256	$\cup(256, 1256)$

Table 1.1: Simulated Processes CPU burst requirement distributions.

Q	1	2	3	4	5
Burst	16	32	64	128	256

Table 1.2: CPU quanta by queue.

The above information is used to determine the mean CPU burst time requirement of all processes generated by the simulation, $\bar{P}$. For an example scenario, a set P of 100 processes will be generated for a simulation, according to the distributions specified in Table 1.2. Given that schedulers use integer precision, the calculations presented here shall also use the same precision, rounding down where necessary.

$$\text{Since } \mu_{\cup(a,b)} = \frac{a+b}{2}, A = \cup(1, 16), B = \cup(16, 256),$$
$$C = \cup(256, 1256), P = \{A, B, C\}, \lfloor \bar{A} \rfloor = 8,$$
$$\bar{B} = 136, \bar{C} = 756, \bar{P} = \frac{55 * \bar{A} + 44 * \bar{B} + 1 * \bar{C}}{100}$$
$$= \frac{55 * 8 + 44 * 136 + 756}{100} = 72 \quad (1.5)$$

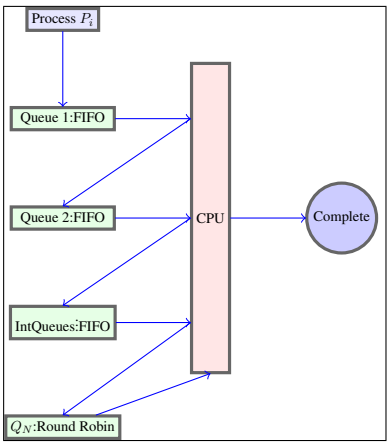

Figure 1.2: Simulated MLFQ Without IO

Thus the mean time required by incoming processes is 72ms. The mean context switching time needed for each process is computed by tallying:

μ	-16ms	-32ms	-64ms	-128ms	-256ms	Ctx.
8	0	0	0	0	0	1
136	120	88	24	0	0	4
756	740	708	644	516	260,4,0	7

The overall mean number of context switches is then calculated:

$$\mu_{CtxSw} = \left\lfloor \frac{55 * 1 + 44 * 4 + 1 * 7)}{100} \right\rfloor = 2 \quad (1.6)$$

Thus, the mean total time needed by processes is $\bar{P} + \mu_{CtxSw} = 74ms$ Because CPU time is

consumed for context switching between process bursts, the CPU will be fully utilized at less than 100% of capacity. Full utilization is forecast as $\frac{\bar{P}}{\bar{P}+\mu_{CtxSw}} = \frac{72}{74} \approx 97\%$. This is confirmed by the original results.

Finally, system load was manipulated by varying the inter-arrival rate λ_{IA}. Since the average process requires 74ms to complete, then full system load will occur at $\overline{\lambda_{IA}} = 74ms$. When $\overline{\lambda_{IA}} < 74ms$, system load>100% and when $\overline{\lambda_{IA}} > 74ms$ then system load < 100%. $\overline{\lambda_{IA}}$ was not held constant; it was allowed to vary below and above the mean, so that brief "spikes and lulls" in processing demand could be simulated.

1.3.3 Original Results

Simulations results for MLFQ and MLFQ-NS were collected and averaged to produce the graphs in Figures 1.3 and 1.4. The results were compared side-by-side. For a comparison of bursts completed in Q_5, refer to Figure 1.3. With MLFQ scheduling, bursts in Q_5 stopped at about 107% of system capacity. With MLFQ-NS, bursts in Q_5 stopped at about 120%. Bursts in Q5 ceased because there were no processes in Q5, directly resulting from starvation in Q_4. $T_{\%}$ doesn't go to 0 until $T_{avail} < 10$. This condition wasn't encountered during simulations, even when system load exceeded 300% of capacity. For a comparison of mean and maximum Q_1 wait-times, refer to Figure 1.4. This shows that MLFQ-NS didn't significantly impact mean Q_1 wait-times. Whereas there is some variation in maximum wait-times, it clearly shows that MLFQ and MLFQ-NS alternated outperforming each other at various system load levels.

1.3.4 Starvation Detection

Starvation is detected in simulations by a decrease in bursts completed in Q_n as workload increases. As shown in 1.3, MLFQ starvation starts to occur around 97%, and complete starvation at 107%. In MLFQ-NS, starvation begins around 97%, and complete starvation at 120%.

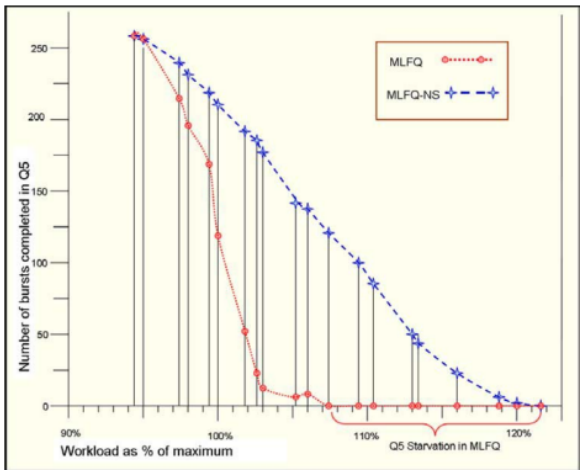

Figure 1.3: Bursts in Q5, MLFQ and MLFQ-NS side-by-side

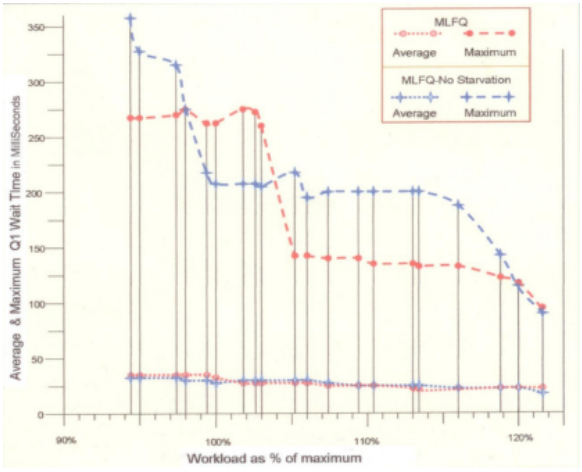

Figure 1.4: Q1 mean and max wait-times, MLFQ and MLFQ-NS side-by-side.

Chapter 2

Simulation Validation and Comparisons

In order to further study MLFQ-NS, compare it to starvation mitigation strategies in other schedulers, and extend it further, a simulation system was needed. PSimJEB [1] was developed as a fork of PSimJ2[2], and is considered a discrete simulation system. PSimJEB was used to duplicate the research conducted in [13]. The results serve to validate the results in [13], and the usage of PSimJEB for conducting further research.

2.1 Validation Methodology

Since the code used to produce original results wasn't available, the validation strategy was to duplicate the original simulation results by running new simulations on different software. They would be conducted with the same operational conditions as described in [13], or as close as possible, while maintaining optimal result collection integrity.

$\mathbf{PR_n}$	**Req.:ms**	**Dist.**
n=55	1-16	$\cup(1, 16)$
n=44	17-256	$\cup(17, 256)$
n=1	257-1256	$\cup(257, 1256)$

Table 2.1: Simulated Process CPU burst requirements.

CPU demand was simulated via incoming processes, whose CPU time requirements are shown in Table 2.1. In distinction to Table 1.1, there are no overlaps in burst requirements between the precentile ranks, for more accurate process distribution tracking. The possible effects on differing simulation results were considered negligible and acceptable.

Simulations were run in batches of 40, each for a duration of 10 kiloseconds[3], to produce data-

[1] Named for the author

[2] `http://ksuweb.kennesaw.edu/~jgarrido/psimj.html`

[3] A kilosecond is 1,000 seconds

points for system-loads between 97% and 150% of capacity. The duration was longer by an order of magnitude than in [13] because 1 kilosecond simulations produced erratic, inconsistent results in PSimJEB. The plotted data-points shown in Figures 2.1 and 2.2 were produced by averaging data-points from 50 batches.

Mean interarrival periods $\overline{\lambda_{IA}}$ were varied to simulate different system-loads. Simulation started with $\overline{\lambda_{IA}} = 80ms$, and with each successive simulation it was decremented by 1ms. The minimum interarrival period was always 1 ms, and the maximum was calculated thus:

$$\overline{\lambda_{IA}} = \frac{\min \lambda_{IA} + \max \lambda_{IA}}{2} \tag{2.1}$$
$$\implies \max \lambda_{IA} = 2 \times \overline{\lambda_{IA}} - \min \lambda_{IA}$$

2.2 Simulation Results

Figure 2.1 shows complete starvation in Q_5 around 107% of system load using MLFQ scheduling, and complete starvation in Q_N around 150% of system-load using MLFQ-NS. This result is different from what is found in [13]. Since the reallaction and diversion algorithms were copied exactly from [13], this is likely due to a subtle variation in the incoming compute demand distributions which results in complete Q_4 starvation occuring at a higher system-load than in [13].

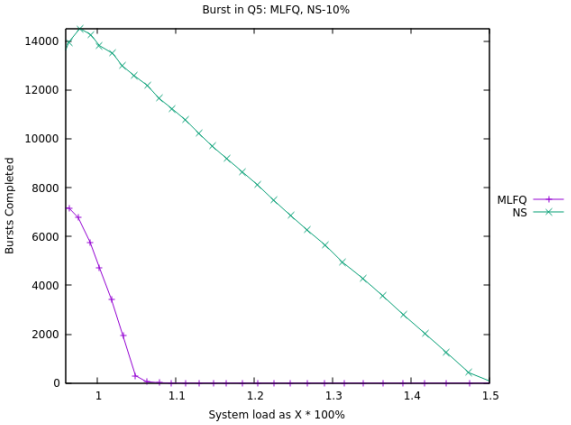

Figure 2.1: Bursts in Q_5, MLFQ and MLFQ-NS, using PSimJEB

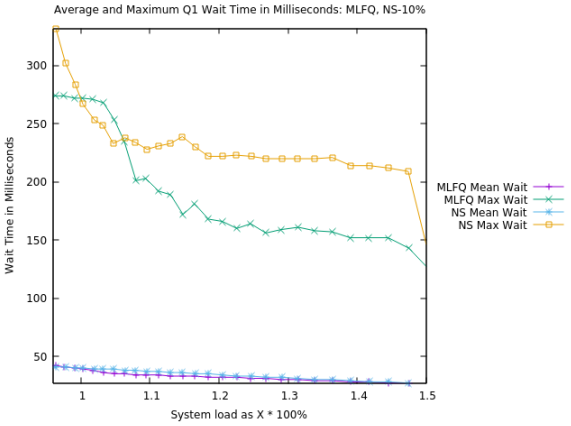

Figure 2.2: Q_1 wait-times, MLFQ and MLFQ-NS, using PSimJEB

Figure 2.2 shows Q_1 wait-times were essentially unchanged from [13]. While there was a significant impact to maximum Q_1 wait-times, impact to mean Q_1 wait-times remained negligible. Furthermore, mean Q_1 wait-times followed the gradually decreasing trend shown in [13]. These observations reflect the original simulation results.

These results confirm that it's safe and effective to divert CPU time to address starvation in Q_N. The similarity to the results in [13] validate those results, and they confirm the viability of PSimJEB to continue and extend research in mitigating starvation in MLFQ process scheduling.

Chapter 3

Simulation Results

Chapter 1 surveyed several process schedulers in search of a similar scheduler to compare Q_5 starvation mitigation effectiveness. That survey failed to identify a scheduler compatible for the comparison. Chapter 3 will compare the results of experimenting with different values for runtime arguments than were used to produce the results in [13].

Specifically this chapter compares the results of varying the weight-factor α and the reallocation percentage. The goal of these two experiments was to see if different run-time arguments yielded better results with respect to CPU bursts given to processes in Q_5 than those published in [13].

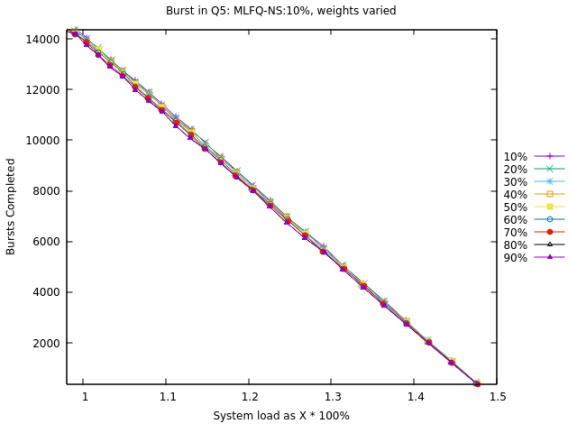

Figure 3.1: Bursts in Q_5 :reallocation constant, weights varied.

3.1 Varying Weight-factor α

Figure 3.1 shows that using different weight-factors $\alpha_{i\in[.1,...,.9]}$ has little long-term impact on bursts in Q_5. Whereas $\alpha = 10\%$ yields a marginal increase in bursts over $\alpha = 90\%$, the difference isn't sufficient to indicate a discovery. Thus varying α has no significant long-term impact on starvation mitigation in Q_5.

3.2 Varying Reallocation Percentage

In [13] it's stated that reallocation of T_{avail} was capped at $T_{\%} = 10$. This gives rise to the question, "How much of that reallocated time was diverted to processes in Q_5, and how much was returned [1] for lack of processes therein?" This question is answered in Figure 3.2. For 10% reallocation, the mean time returned per T_{period} was approximately 4ms at about 97% system load, and peaks in the graph at more than 70ms just short of 150% system load. For 1% reallocation, returned time doesn't get above 1ms per T_{period} till after system load exceeds 140%, and at 150% of system loads reaches approximately 7ms per T_{period}. There is a strong implication that that as system load increases far past 100%, reallocation becomes less effective in Q_5.

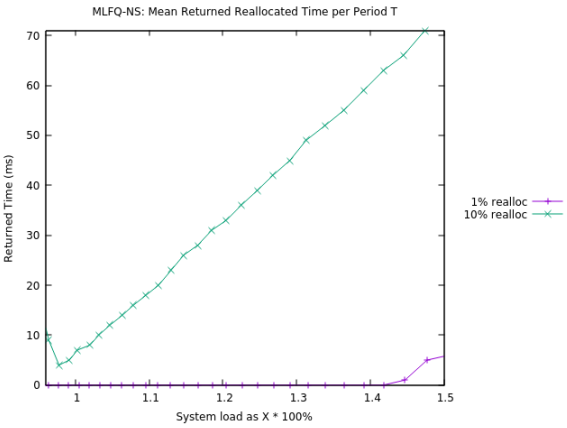

Figure 3.2: Mean Reallocated Returned Times

Figure 3.3 shows measureable differences between $T_{\%} = 1\%$ and $T_{\%} = 10\%$. With respect to the number of bursts in Q_N, $T_{\%} = 1\%$ outperforms $T_{\%} = 10\%$. This shall be referred to as the *reallocation anomaly*, and will be analyzed in the next section.

3.2.1 The Reallocation Anomaly

The following came from direct observation of a simulation in progress[2]. These are the runtime arguments for that simulation:

- $\overline{\lambda_{IA}} = 60ms \approx 120\%$ compute capacity
- simulation duration=10,000s
- $|T_{period}| = 1000ms$
- $T_{\%} = 10$

Whereas more time is reallocated for Q_N starvation by diverting 10% of T_{avail} than 1%, a lack of waiting jobs in Q_N resulted in diverted time being returned to $Q_2 \ldots Q_4$. A buildup of jobs in Q_4 was observed, such that very few jobs entered Q_5. From one period T to the next, between 0 and 5 jobs were observed in Q_5. Most frequently, there was at most 1 job in Q_5. This indicates

[1] "returned" diverted time means that reallocated time is reset to 0ms and the scheduling decision is made via standard MLFQ algorithm.

[2] Debugging running software in an IDE

starvation in Q_4 at 150% of system load, similar to the starvation described in [13].The following scenario is presented to explain the reallocation anomaly.

Suppose that have one job arrive in Q_5 needing 80ms of CPU time to complete, and the system load is 150%. Because of the severe starvation occurring in Q_4, no other jobs move down to Q_5 for a significant period of time, say ten T_{period}=1,000ms periods. Two cases are presented, one in which $T_{\%} = 1$, and one in which $T_{\%} = 10$.

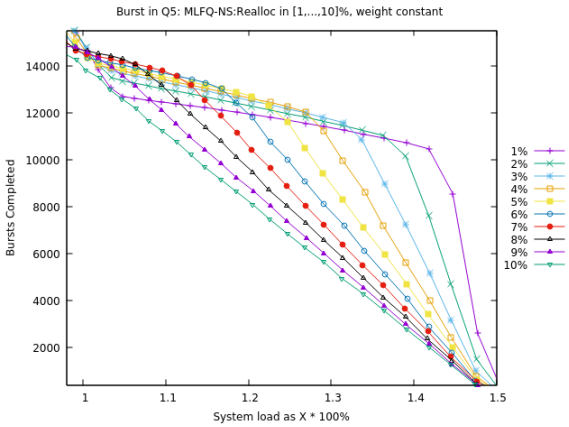

Figure 3.3: Bursts in Q_5 :reallocation varied, weights constant.

1. **10% reallocation of** T_{avail}: The single job in Q_5 will receive 72ms during period T_i, and 70ms during the next period T_j. This process therefore completes in T_j, and the simulation counter registers 2 bursts given to Q_5 between the beginning of T_i and the end of T_j. There are also 62ms of time that are returned to $Q_2 \ldots Q_4$.

2. **1% reallocation of** T_{avail}: The single job in Q_5 will receive 7ms during period T_i, and similar time for the next 9 1,000ms periods. After the ten 1,000ms periods have completed, the simulation counter registers 10 bursts for processes in Q_5. The job has not yet completed, but will likely do so after 2 more 1,000ms periods. By that time, another job may have been enqueued into Q_5.

This analysis illustrates that while there may be more bursts in Q_5 at system load $\gg 100\%$ with $T_{\%} < 10\%$, those bursts are of shorter duration and the processes in Q_5 remain in the system longer because of it. Furthermore, measuring performance in Q_5 past 120% may not be meaningful because of the starvation problem in Q4 being such that processes aren't making it to Q_5. Finally, it's evident that most of the time reallocated to Q_5 will be returned to $Q_2 \ldots Q_4$ at system loads where Q_4 is experiencing starvation.

With this explanation for the reallocation anomaly, the idea was introduced to mitigate starvation not just in Q_5, but also in Q_4. This would lead to an extension of MLFQ-NS, and increased mitigation in Q_5 and Q_4, as well as introducing new complications. This will be discussed in the next chapter.

Chapter 4

Intelligent Mitigation

Chapter 3 explored the effects of experimenting with simulation run-time arguments, and revealed a case in which MLFQ-NS becomes ineffective. This is listed below, in addition to another case in which MLFQ-NS might become ineffective:

1. Starvation in Q_4 prevents almost all long-running processes from making it to Q_N ,
2. $\overline{\lambda_{IA}}$ is small enough that $T_{\%=10} * T_{avail} = 0$ [1]

Time is still available for diversion even at high system load, since it's starvation in higher queues which prevents processes from being enqueued into Q_N .

This chapter introduces and explores MLFQ-IM, or ***Intelligent Mitigation***. MLFQ-IM is an extension to MFLQ-NS, and it's mechanics are described:

- Time is still never diverted from Q_1 ,
- IM has a set of last queues, $Q_{[M,...,N]}$, such that N doesn't necessarily equal M+1,
- IM has a set of intermediate queues numbered $Q_{[2,...,M-1]}$,
- It uses the same mathematical functions–equations 1.2,1.3,1.4–to determine the amount of time to divert to $Q_{[M,...,N]}$,
- For puposes of redirecting time, the scheduler will check *backward*, from Q_N to Q_M , till it finds a waiting process. If none are found, reallocated time is returned to $Q_2, \ldots, Q_{M-1}$.

[1] None of the simulations run in this research ever reached $\overline{\lambda_{IA}}$ small enough to induce $T_{\%=10} * T_{avail} = 0$. Mitigating starvation under such circumstances is futile, and so wasn't simulated.

This chapter will explore the results of simulating MLFQ-IM. To maintain direct comparability to [13], simulations were run with 5-level MLFQ. Burst performance in Q_5 and Q_4 will be analyzed, and then mean wait-times in Q_1. Recall that mean wait-times must not be significantly impacted, and maximum Q_1 wait-times shouldn't be impacted more than was the case with MLFQ-NS. Then the impact that MLFQ-IM has on scheduling in intermediate queues will be analyzed.

4.1 Burst Performance

Shown in Figure 4.1 are the CPU bursts granted to processes dequeued from Q_5. There is an *apparent* performance boost in Q_5. Whereas bursts in Q_5 cease altogether at approximately 150% of compute capacity with MLFQ-NS, MLFQ-IM plateaus at about 140% with 10,000 bursts per simulation[2], and extends beyond 150% compute capacity. Extended duration simulations showed that bursts in Q_5 continued till approximately 300% of compute capacity. However, at 300% of capacity the very concept of mitigation is questionable. In this scenario one is compelled to consider upgrading compute capabilities. The impact of mitigating starvation at 300% overload will be explored in a subsequent section.

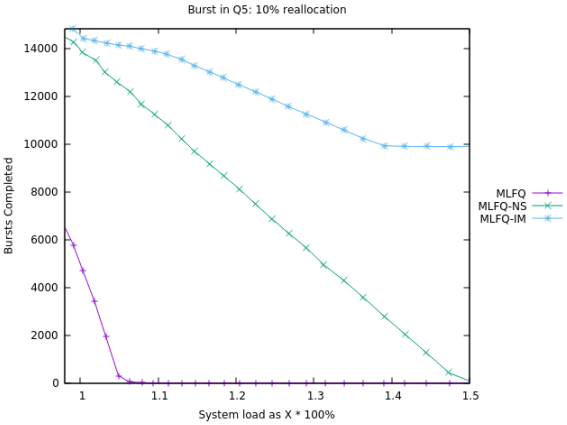

Figure 4.1: MLFQ-IM bursts in Q_5

Shown in Figure 4.2 are the CPU bursts granted to processes dequeued from Q_4. It shows that with respect to bursts in Q_4, MFLQ outperforms MLFQ-NS and MLFQ-IM till approximately 140% of compute capacity. MLFQ-NS and IM are comparable till between 95% and 140% capacity. Similarly to bursts in Q_5, Q_4 bursts plateau at 140%. There is a strong implication here that starvation in Q_4 resulting in starvation in Q_5 really begins at 140% of capacity.

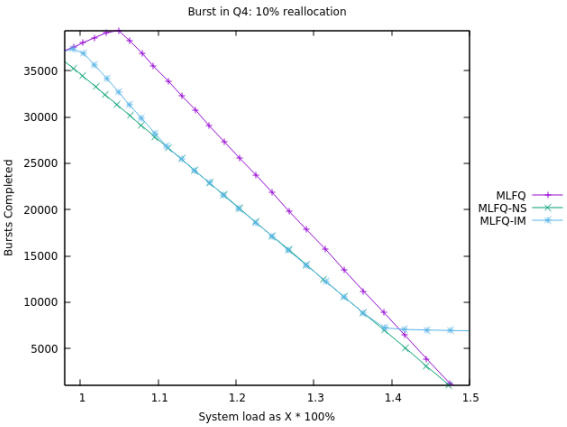

Figure 4.2: MLFQ-IM bursts in Q_4

With Q_5 and Q_4 burst plateaus comes the implication of constant performance in those two queues. This then implies that performance in other

[2]Since simulations are 10k-seconds long, 10k bursts in Q_5 per simulation implies 1 burst in Q_5 per second.

queues would likely suffer. At $System\text{-}load \gg 95\%$ there is more work in some period of time T than there is compute capacity to process demand in T. This will be addressed in a subsequent section.

4.2 Q_1 wait-times

Shown in Figure 4.3 are the mean and maximum wait-times in Q_1 for MLFQ, NS and IM. It shows a similar pattern as before, that mitigating schedulers hold a slight lead over MLFQ at about $100\% \leq System\text{-}Load \leq 107\%$ with respect to maximum Q_1 wait-times, and MLFQ outperforms NS and IM beyond 107%. Interestingly, IM outperforms NS w.r.t. maximum wait-times. Finally, mean wait-times are largely unaffected by any mitigation techniques.

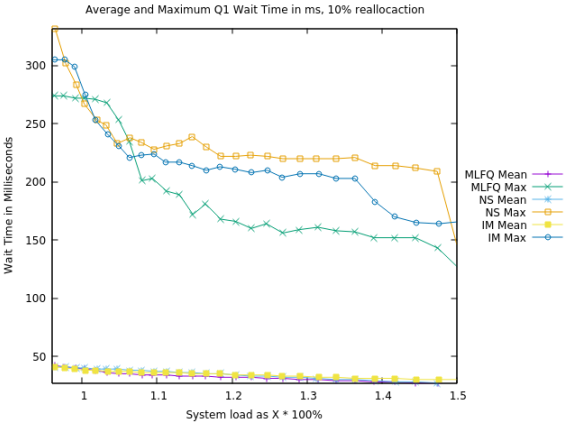

Figure 4.3: MLFQ-IM bursts in Q_4

4.3 Considering The Side Effects of MLFQ-IM

As discussed in the preceding sections, starvation mitigation in $Q_{5...4}$ with MLFQ-IM *must* have an impact w.r.t. to the intermediate queues, especially where bursts plauteau, granting constant performance through some system load percentil range. When $System\text{-}load \gg 97\%$, there simply isn't enough computing capacity to serve all processes in the ready queue. Diversion of time to $Q_{5...4}$, with little or no return of reallocated time, must have a measureable impact in other queues.

Inspection of Figure 4.3 indicates that Q_1 is not *adversely* affected by mitigation strategies, at least up to 150% capacity. A "pile-up" of processes in Q_1 would cause the mean wait to increase. For this reason, the in-

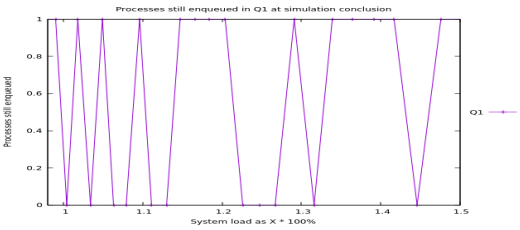

Figure 4.4: Processes remaining in Q_1

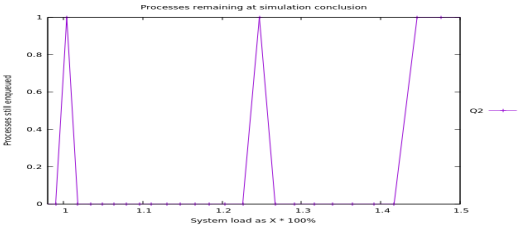

Figure 4.5: Processes remaining in Q_2

termediate queues are likely affected. We'll now inspect five figures which show the quantity of processes left in $Q_1 \ldots Q_5$.

Figure 4.4 shows that at most 1 process is remaining in Q_5 at the end of a simulation. Whereas the graph seems to vary widely between the minimum and maximum values, bear in mind that those values are 0 and 1, respectively. Figure 4.5 shows much the same, except that more frequently Q_2 has no processes remaining.

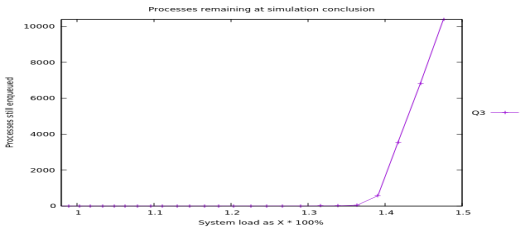

Figure 4.6: Processes remaining in Q_3

Figure 4.6 shows a different outcome, however. Just before the system load reaches 140%, the number of processes remaining in Q_3 rises dramatically, cresting past 10,000 just before system load reaches 150%. This implies the possibility that starvation has been *artificially induced* in a higher queue than otherwise might have occurred.

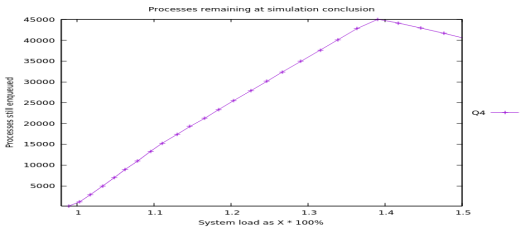

Figure 4.7: Processes remaining in Q_4

Figure 4.7 definitely shows the impact of starvation in Q_4. The number of processes remaining in Q_4 steadily rises from around 0 at approximately 95% system load to 45,000 remaining at 140% system load. It predictably starts to decrease after 140% systeload load because MLFQ-IM has at that point begun mitigating starvation in Q_4.

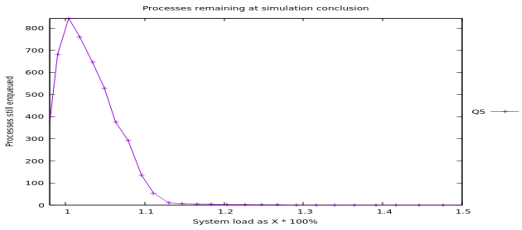

Figure 4.8: Processes remaining in Q_5

Figure 4.8 shows the affect of concentrating on Q_5 for starvation mitigation. Q_5 is always checked first for time diversion, and then Q_4. Starvation in Q_5 peaks at about 800 processes at just past 100% system load, then quickly descends to almost 0 just past 110%.

Chapter 5

Conclusion

In this thesis we've reviewed the origins[1] and evolution of MLFQ. After a survey of modern schedulers it was concluded that a direct comparison with another scheduler was *not* possible. We've explored recent extensions to MLFQ with respect to starvation mitigation[13], Q_1 latency, and overall throughput. We've reviewed MLFQ-NS, duplicated the research in [13], and discovered that MLFQ-NS *can* be improved. We've explored a possible extension of MLFQ-NS, which is MLFQ-IM, and concluded that there is a range of system load percentiles in which uage of MLFQ-IM is appropriate.

While research on MLFQ-IM was ongoing, it appeared to produce remarkable results. Bursts in Q_4 and Q_5 increased substantially, with no apparent effects in Q_1. This exploration leads us to conclude that MLFQ-IM is effective *to a point*.

It counteracts the very nature of MLFQ in general to mitigate starvation in $Q_{5...4}$ only to induce starvation in Q_3 and perhaps even higher than that. Past 140% system load it essentially reprioritizes low priority processes over higher priority processes, when the original goal stated in [13] was to prevent starvation of some low priority processes in cases where it was safe to divert time from higher priority processes, under certain conditions.

Given the tendency of MLFQ-IM to induce starvation in higher priority queues when $system\text{-}load \geq 140\%$ it's our conclusion that MFLQ-IM should not be used past 140%. The range of system loads in which it's appropriate to use MLFQ-IM should be stated as $[97\% \ldots 140\%]$. This has the minimum impact on higher priority processes, and maximizes the usage of time diversion to starving processes. Diverted time which may be been "returned" in MLFQ-NS may be diverted by MLFQ-IM to mitigate starvation in Q_M.

[1] Compatible Time Sharing System [8]

Chapter 6

Future Work

This section offers two possible heuristics to more finely control when time is diverted from $Q_2 \ldots Q_{N-1}$ to Q_N .

6.1 Additional Comments Regarding MLFQ-NS

There is no mechanism by which MLFQ-NS is activated or deactived; it is an enhanced MLFQ and is always in operation. Ideally unless starvation is occuring in Q_N there won't be any processes in Q_N to divert time to. However, a scenario exists in which it's possible that diverted time may be granted to a process in Q_N inappropriately. In this scenario, one or more processes are enqueued into Q_N a short time before time period T_i progresses to T_j. These processes then don't have a long wait before receiving diverted time via MLFQ-NS. The following two heuristic methods proposed to account for this scenario.

6.1.1 Q_1 Wait-Time Heuristic

It is proposed that the next scheduled process waiting in Q_N shall be required to have waited for some period of time $T_{Q_N Wait}$ before receiving diverted time. This introduces a new but minor datum which must be tracked–the last time that a process was enqueued. This datum must be tracked per process; this could be tracked as an attribute of the process itself, or as something the sheduling mechanism tracks[1]. This could be a subject for further research.

[1] scheduler based tracking makes little sense, as the data structure used by the scheduler must be able to scale to very large numbers.

6.1.2 Q_1 CPU Usage Tracking

It is proposed that CPU usage by processes in Q_1 must reach a certain point before time diversion to $Q_{M...N}$ is used. This requires tracking of CPU time in Q_1 , T_{Q_1}. However, this is already done in MLFQ-NS and MLFQ-IM, and so only requires additional evaluation of T_{Q_1} with respect to some other value. T_{Q_1} would have to reach some percentage of T_{period} :

$$P_{activation} = \frac{|T_{Q_1}|}{|T_{period}|} \tag{6.1}$$

Currently $P_{activation}$ is unknown. It's existence is certain because T_{Q_1} increases as system load increases. This heuristic requires very little additional effort, considering it's a comparison of data already known and a division operation. This could be the subject of further research.

References

[1] Andrea. Arpaci-Dusseau. *Multilevel Feedback Queue Schedulers*. 2000 (accessed October 6,2016. URL: http://pages.cs.wisc.edu/~eli/537/lectures/Solaris.pdf.

[2] R.H. Arpaci-Dusseau and A.C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2012. ISBN: 9781105979125. URL: https://books.google.com/books?id=orxwMwEACAAJ.

[3] Rudolf Bayer. "Symmetric binary B-Trees: Data structure and maintenance algorithms". In: *Acta Informatica* 1.4 (1972), pp. 290–306. ISSN: 1432-0525. DOI: 10.1007/BF00289509. URL: http://dx.doi.org/10.1007/BF00289509.

[4] Luca Becchetti and Stefano Leonardi. "Nonclairvoyant Scheduling to Minimize the Total Flow Time on Single and Parallel Machines". In: *J. ACM* 51.4 (July 2004), pp. 517–539. ISSN: 0004-5411. DOI: 10.1145/1008731.1008732. URL: http://doi.acm.org.proxy.kennesaw.edu/10.1145/1008731.1008732.

[5] HS Behera, Reena Kumari Naik, and Suchilagna Parida. "Improved multilevel feedback queue scheduling using dynamic time quantum and its performance analysis". In: *International Journal of Computer Science and Information Technologies* 3 (2012), pp. 3801–3807.

[6] Ayan Bhunia. "Enhancing the performance of feedback scheduling". In: *Int. J. Comput. Appl., vol* 18 (2011).

[7] R. Brown. "Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem". In: *Commun. ACM* 31.10 (Oct. 1988), pp. 1220–1227. ISSN: 0001-0782. DOI: 10.1145/63039.63045. URL: http://doi.acm.org/10.1145/63039.63045.

[8] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. "An Experimental Time-sharing System". In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE '62 (Spring). San Francisco, California: ACM, 1962, pp. 335–344. DOI: 10.1145/1460833.1460871. URL: http://doi.acm.org.proxy.kennesaw.edu/10.1145/1460833.1460871.

[9] Kenneth J. Duda and David R. Cheriton. "Borrowed-virtual-time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler". In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP '99. Charleston, South Carolina, USA: ACM, 1999, pp. 261–276. ISBN: 1-58113-140-2. DOI: 10.1145/319151.319169. URL: http://doi.acm.org.proxy.kennesaw.edu/10.1145/319151.319169.

[10] M. EffatParvar et al. "An Intelligent MLFQ Scheduling Algorithm (IMLFQ) with Fault Tolerant Mechanism". In: *Sixth International Conference on Intelligent Systems Design and Applications*. Vol. 3. 2006, pp. 80–85. DOI: 10.1109/ISDA.2006.10.

[11] Jose M Garrido, Richard Schlesinger, and Kenneth Hoganson. *Principles Of Modern Operating Systems*. Jones & Bartlett Learning, 2011. ISBN: 1449626343.

[12] Liang Guo and Ibrahim Matta. "Scheduling Flows with Unknown Sizes: Approximate Analysis". In: *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '02. Marina Del Rey, California: ACM, 2002, pp. 276–277. ISBN: 1-58113-531-9. DOI: 10.1145/511334.511378. URL: http://doi.acm.org.proxy.kennesaw.edu/10.1145/511334.511378.

[13] Kenneth Hoganson. "Reducing MLFQ Scheduling Starvation with Feedback and Exponential Averaging". In: *J. Comput. Sci. Coll.* 25.2 (Dec. 2009), pp. 196–202. ISSN: 1937-4771. URL: http://dl.acm.org.proxy.kennesaw.edu/citation.cfm?id=1629036.1629067.

[14] Tim Jones. *Inside the Linux 2.6 Completely Fair Scheduler*. 2009. URL: http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

[15] Tsuyoshi Katayama. "Mean sojourn times in a multi-stage tandem queue served by a single server." In: *J. OPER. RES. SOC. JAPAN.* 31.2 (1988), pp. 233–247.

[16] Jeff Roberson Kirk McKusick. "The Freebsd ULE Scheduler". In: *Freebsd Journal* (2014).

[17] Jim Mauro. *Solaris internals : core kernel components*. Palo Alto, CA: Sun Microsystems, Inc, 2001. ISBN: 0-13-022496-0.

[18] Marshall Kirk McKusick and George V. Neville-Neil. "Thread Scheduling in FreeBSD 5.2". In: *Queue* 2.7 (Oct. 2004), pp. 58–64. ISSN: 1542-7730. DOI: 10.1145/1035594.1035622. URL: http://doi.acm.org/10.1145/1035594.1035622.

[19] Supriya Raheja, Reena Dadhich, and Smita Rajpal. "Designing of vague logic based multilevel feedback queue scheduler". In: *Egyptian Informatics Journal* 17.1 (2016), pp. 125–137. ISSN: 1110-8665. DOI: http://dx.doi.org/10.1016/j.eij.2015.09.003. URL: http://www.sciencedirect.com/science/article/pii/S1110866515000481.

[20] Guido Schafer et al. "Average case and smoothed competitive analysis of the multi-level feedback algorithm". In: *IEEE FOCS03* (2003), p. 462.

[21] L. E. Schrage. "The Queue M/G/1 with Feedback to Lower Priority Queues". In: *Management Science* 13.7 (1967), pp. 466–474. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/2627689.

[22] M. Thombare et al. "Efficient implementation of Multilevel Feedback Queue Scheduling". In: *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. 2016, pp. 1950–1954. DOI: 10.1109/WiSPNET.2016.7566483.

[23] Lisa A Torrey, Joyce Coleman, and Barton P Miller. "A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler". In: *Software: Practice and Experience* 37.4 (2007), pp. 347–364.

[24] S. Wang et al. "Fairness and Interactivity of Three CPU Schedulers in Linux". In: *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 2009, pp. 172–177. DOI: 10.1109/RTCSA.2009.26.

[25] Chee Siang Wong et al. "Towards Achieving Fairness in the Linux Scheduler". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 34–43. ISSN: 0163-5980. DOI: 10.1145/1400097.1400102. URL: http://doi.acm.org.proxy.kennesaw.edu/10.1145/1400097.1400102.