Journal of Engineering and Technology for Industrial Applications

ITEGAM-JETIA

Manaus, v.11 n.51, p. 236-242. January/February., 2025.

DOI: https://doi.org/10.5935/jetia.v11i51.1505

ISSN ONLINE: 2447-0228

RESEARCH ARTICLE

OPEN ACCESS

STEALING SOME NOTATION FROM BIG O NOTATION TO DEVELOP A NEW MULTITHREADING PRIORITY FORMULA

Abdulamir Abdullah Karim[1], Yaser Ali Enaya[2] and Ghassan Abdulhussein Bilal[3]

[1] Department of Computer Science, University of Technology, Baghdad, Iraq.
[2, 3] Department of ElectroMechanical Engineering, University of Technology, Baghdad, Iraq.

[1]https://orcid.org/0000-0002-8420-5681, [2]https://orcid.org/0000-0002-0669-1282, [3]https://orcid.org/0000-0002-5090-103X

Email: abdulamir.a.karim@uotechnology.edu.iq, 50111@uotechnology.edu.iq, ghassan.bilal@uotechnology.edu.iq

ARTICLE INFO

Article History
Received: December 25, 2024
Revised: January 20, 2025
Accepted: January 25, 2025
Published: February 28, 2025

Keywords:
thread,
priority,
time complexity,
big O,
inversion,
starvation.

ABSTRACT

This work aims to develop the CPU industry by distributing its time between the threads efficiently. To do so, an unprecedentedly developed equation is suggested as a new powerful software to increase the CPU performance. This proposed equation dedicates to solve the problem of children inheriting their parents priorities equivalently without a thoughtful basis in multithreading by involving big O to give threads different values, whose importance is inversely proportional to their O(n)s. The second originality is breaking complexity rule, which considers loop iterations if the threads have the same O(n), since usually threads run on the same computer. Therefore, the ratio (No. of loop's iterations to go/total iterations multiplied by O(n)) determines thread importance inversely. The third novelty is replacing Round Robin with Big O and iteration ratio. A parser is applied to seek "for" and "while" tokens for O(n) measuring purposes. Three threads, p1 $O(n^2)$, p2 O(n), and p3 $O(n^2)$, approved the equation with results of 32, 51, and 8 time slices, respectively, during the period 0-1000 ms. Meanwhile, Round Robin gives the children the same slice number.

I. INTRODUCTION

In modern computer systems, threads are preferred over processes, and multithreading over multitasking [1]. Multithreading has a problem that has not touched before which is the equivalency among the priorities of threads. This problem prevents exploiting the full multithreading utilities efficiently since its objective is to decrease CPU idle time in order to improve system performance, use less memory, and execute context switching in order to share memory and speed up thread switching (scheduling) [2]. The scheduling policy includes these rules: the threads with higher priority receive more CPU time than those with lower priority; a higher priority thread may preempt a lower priority thread; and threads with equal priority receive equal CPU time [3]. The problem is with the third rule because the scheduler gives equal priorities to the children threads without studying their background. For example, any Java program that is executed starts its code from the main function. In order to begin running the code included in the main function, the JVM generates a thread which is referred to as "main thread". The main thread is crucial to understand since it inherits the priority of all other threads, is the source from which they are formed, and must be the last thread to complete execution at all times as depicted in Figure 1 [4]. Each new process is therefore formed with a single thread that competes using priority over its parent process for the processor with the threads of other processes and shares the private segment and other resources [5].

Therefore, they are given arbitrarily the same priority causing unfair competitive between high and low priorities threads as can be seen in the priority techniques that are used by Java and IBM. So, as known, Java is fully based object-oriented which operates in a multithreading environment where a thread scheduler allocates the processor to a thread based on its priority. Java requires that every thread be given a priority when it is created. Priorities can vary from 1 to 10, with 10 being the highest priority. With IBM, for each thread, the kernel keeps track of a priority value, also known as the scheduling priority. The significance of the thread corresponds in reverse with the priority value, which is a positive integer. In other words, a thread with a lower priority value has higher priority. [6], [7].

Moreover, there are two types of thread priorities: fixed and nonfixed; the fixed-priority has an unchanged value, whereas a nonfixed-priority adjusts depending on the processor-usage penalty, the thread's nice value (20 by default), and the least priority

of user threads (40). A thread's priority value is subject to quick and frequent adjustments. The scheduler's priorities recalculation method is the consequence of the ongoing movement. However, for threads with fixed-priority, this is not the case. Meanwhile, the time slice is the maximum amount of time that a thread can be in charge before it risks being displaced by another thread [8].

There are many efforts has been done to improve the priority or utilize it in other systems. For instant, in [9], Based on the RTCOP framework and using multithreading, an architecture for preemptive layer activation called as PLAM has been presented. The non-exception handling layers can be triggered concurrently using PLAM. More work, the majority of complicated processing issues can be solved by applying the Chip Multiprocessor (CMP) technique, which is known for its good performance and high speed for personal computers and Smartphone [10], [11]. For example, in [12] and [13] Multithreading on Android Matrix multiplication program run on single and multi-core for comparing purposes in order to determine the constraints that stand up as obstacles against accomplishing the best execution of time reduction.

Additionally, multithreading middleware for sensor virtualization is built in both the sensor node and the gateway, which lessens the latency brought on by the virtualization of the sensors. Otherwise, scheduling policy, energy use, and memory resources are the three fundamental networking challenges; [14-16] offer prioritization approach to resolve these problems by spotting in the thread priorities' derivation mechanism that is based on inputs from three different sources: threads, the operating system, and external sources like timers to meet the needs of their unique nature. Else, [17] and [18] demonstrate that, in the best instances, the schedulable utilization for the hardware under consideration is roughly multiplied compared to partitioned scheduling without SMT. On the other hand, time complexity is a crucial component for efficient usage on real platforms to decrease the executing time of the algorithm and the completion time of applications, which results in lowering user waiting time [19].

The size of the input is multiplied by the time complexity of an algorithm to determine how long it will take to run [20]. Time complexity involves in many pieces of research specially these are related to the algorithms. For example, designing algorithms to reduce the schedule time for linear and binary PSO, [21] Develops an algorithm to address the issue of the subsequence matching's inherent time complexity.

Other studies, [22] and [23] identify the most effective Traveling Salesman Problem algorithm by evaluating complexity, which has been confirmed to be polynomial equation. All these works are the most related pieces of research to our paper, and it is noticeable that they are located either in multithreading priority field or time complexity field without combining between them which makes this paper the first attempt. So, in this study, a new priority equation is developed to involve time complexity for deciding the next run thread among threads that have equal priorities. Furthermore, an iteration ratio supports the time complexity taking decision among the same polynomial rank threads.

II. METHODOLOGY

This work suggests a developed Multithreading priority equation to solve the equivalent priorities problem by involving constant O(1) and polynomial O(ni) times from Big O notation as one of its terms for the first time in the priority world.

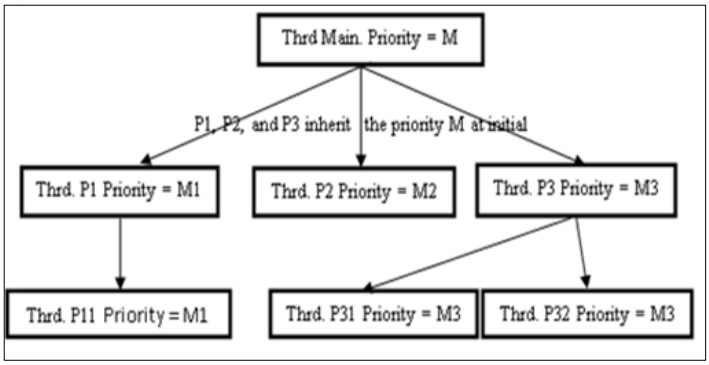

Figure 1: Priority inheriting: P1, P2, and P3 inherit the priority M of the parent Main when they are created. P11 inherits the P1 priority at its creating time t1 = M1. P31 and P32 inherit the P3 priority at their creating time t3= M3.
Source: Authors, (2025).

The big O task is the engaging in the equation calculation, Eq. (1), whenever there are equivalent priorities to give them different values which their importance is being inversely with their O(ni) levels. The second originality is breaking the rule of time complexity which is the number of loop iterations taking part in the equation calculation, Eq. (2), if the threads have the same O(ni) since usually threads run at the same computer and operating system. Therefore, Eq. (2) is multiplied by O(ni) to decide thread importance inversely as well. Third novelty is replacing the Round Robin method, which gives the same slice number to all threads with the sane priorities, by Big O and iteration ratio. The time slice is the time that a thread is allowed to consume without interrupting by the scheduler and swapping it with another same priority thread. In this algorithm, the priority value is increased by the CPU usage counter causing lowering the priority since the relation between them is reciprocal. A thread's most recent CPU usage is utilized to determine the processor penalty. At the end of each time slice (10 ms), the recent processor usage value or counter grows by 1, until reaching the value 120 when the swapper recalculates it for all threads. The swapper recalculates the recent processor usage values every second as well. The minimum priority and the nice value in Eq. (1) equal the defaults 40 and 20 respectively [6].

$$Priority = base\ process\ priority + nice\ value + ((time\ slices\ counter) \times (schedo - o\ sched\_R)) + (iteration\ ratio \times O(n)) \quad (1)$$

$$Iteration\ Ratio = \frac{Number\ of\ loop's\ iterations\ to\ go}{Total\ number\ of\ iterations} \quad (2)$$

Where nice value is the factor that controls the priority and considered a measure of how much the thread cooperates in sharing the CPU, schedo is a CPU scheduler tuning by changing its parameters that are used to calculate threads' priority [6].

For complexity and the iteration ratio part, the formula is applied as is follows:

1- The priority of thread = swapper calculation, if its iteration ratio × O (n^i) is the lowest among all threads.

2- If the thread has the lowest (iteration ratio × O (n^i)) among all threads with same O (n^i), then its priority = highest priority among all threads of O (n^i-1) + 1.

3-If the thread has higher (iteration ratio × O (n^i)) than other threads which have the same O (n^i), then the thread priority = Highest priority among these threads + 1.

So, instead of giving arbitrary equal priorities for all threads at time zero, time complexity assigns actual priorities at the same time.

II.1 MULTITHREADING

In a multithreading, the priority is assigned to thread by the scheduler of the operating system. There are multi priority levels where each thread is granted a specific priority level according to its importance [24], [25]. With large number of threads and limited resources execution environment, control the priority becomes very crucial to organize threads competing for CPU time [26]. Different operating systems implement different priority scheduling algorithms such as Earliest Deadline First (EDF), Multilevel Feedback Queue Scheduling (MLFQ), and Fixed-Priority Scheduling (FPS). Developing any priority scheduling algorithm always faces two major deficiencies that are thread inversion and starvation. Thread inversion is holding resource by low priority thread when the higher priority thread demands it. Starvation, on the other hand, is depriving a thread with lower priority of CPU time consistently because of overtaking by higher priority threads [27]. Figure 2 illustrates these problems, where threads P1 and P2 share S1 and S2 resources, according to these scenarios: if P1 priority > P2 priority and holds S1 or S2 without releasing and P2 demands that resource, then the starvation occurs. On the other hand, if the same scenario is happened, but with P1 priority < P2 priority, then inversion occurs. These two scenarios are represented by the vertical gray and white box in the figure and vice versa if P2 holds S1 or S2 which may enter P1 into starvation or inversion state represented by the horizontal light gray box in the figure. So, in this work, these two problems are solved by limiting the count of successive time slices that thread may get them. Each algorithm offers advantages and trade-offs, but no one solves the equivalent multithreading priorities, which is unique to this research, by abolition this disorder using a new priority equation, Eq. (1) that has a new pathless concept [28], [29].

II.2 TIME COMPLEXITY

Run time and scalability are the most important parameters to evaluate the algorithms' performances. Since the relation between these features is reversible, algorithms' worst-cases measuring is represented by runtime growth rate verses the increasing of the input size, and big O notation is the tool that is used as measurement for these algorithms which is called a complexity [30], [31]. The complexity of an algorithm is a scale of the data segment that is needed for processing in order to function sufficiently. The number of times the algorithm must execute, relative to the length of the input, is known as time complexity [32], [33]. Since other factors such as operating system, processor power, and programming language are considered, time complexity is not working as a measure of how long a specific algorithm taking to run. Time complexity depicts the run time needed to finish the whole algorithm, not measures exact running time in second or millisecond [34]. So, one of the tools to describe the algorithm time complexity is the Big O notation that applies mathematical equations. These equations include constant time O(1), divide and conquer O(log(n)), polynomials $O(n^i)$, exponentials $O(2^n)$, factorials O(n!) [35]; Table 1 shows some of the runtimes for various algorithms.

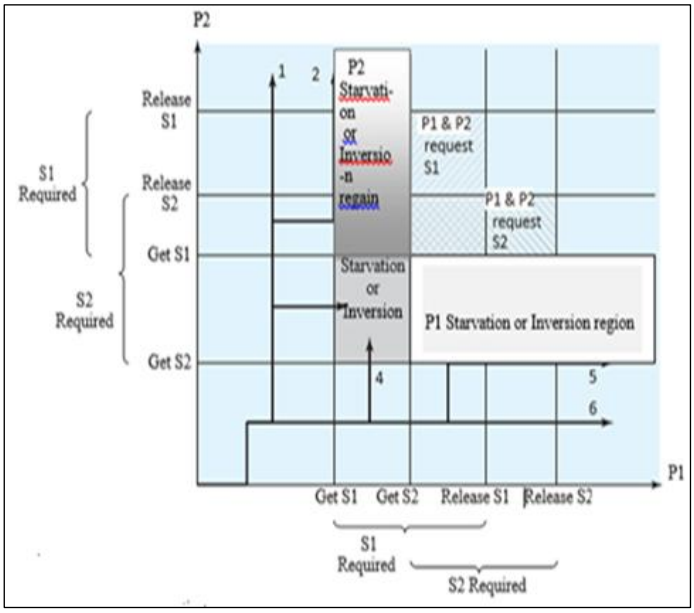

Figure 2: Threads P1 and P2 Starvation and Inversion diagram. P1 execution is represented as x-axis (arrow) and P2 is waiting. P2 execution is represented in the y-axis (arrow) and P1 is waiting.
Source: Authors, (2025).

Where:

= P1 & P2 request resource S1.

= P1 & P2 request resource S2.

= Starvation or Inversion starting region of P1 or P2.

= Starvation or Inversion region of P1.

= Starvation or Inversion region of P1.

= Starvation or Inversion region of P2.

= Possible progress path of P and Q.

Horizontal portion of path indicates P is executing and Q is waiting. Vertical portion of path indicates Q is executing and P is waiting.

Since this work is the first work that Big O notation is involving in multithreading priority, just two of its equations, constant and polynomials times, have been chosen to prove the idea since the basic concept is the same for all equations just needed to extend the parser. Figure 3 is the flowchart that illustrates the individual algorithmic loop process to measure the thread complexity that is used in this work by finding nested loop with the highest depth to use it later to specify the thread priority. By tracking the flowchart path, two things are gotten as outputs: first, the highest depth among nested loops, and second, the loop with the largest remaining iteration count. So, from the "into" and "out of" the flowchart the dominant loop is specified which is taking the largest part to the algorithm's runtime. Next, the growth rate of the dominant loop is used in the algorithm's time complexity calculation. So, let's take the bubble sort algorithm as an example to calculate the complexity where the goal of the algorithm is sorting unarranged members. To do so, the number of nested loops is calculated guiding to complexity of $O(n^2)$ because the algorithm needs two loops (nested) to reorder the members [36-38]. A parser

program in C++ has been written to seek "for" and "while" tokens for $O(n^i)$ measuring purposes. The flowchart in Figure 3, depicts the parser flow of the thread algorithm to get its complexity according to the count of nested loops in order to use it in to assign priority to the thread. The second output of the flowchart is the loop with highest remaining iteration count, which is used as priority backup plan to differentiate threads with same complexity.

Table 1:Runtime complexity for various algorithms with least numbers of consecutive operations.

Algorithm	Runtime Complexity	Consecutive Operation
Recurrent	O(n)	O(n)
Transformer	$O(n^2)$	O(1)
Sparse Transformer	$O(n\sqrt{n})$	O(1)
Reformer	O(n log(n))	O(log(n))

Source: Authors, (2025).

III. IMPLEMENTATION

For the implementation, this paper applies an experiment with three threads: p1 $O(n^2)$, p2 O(n), and p3 $O(n^2)$ as its steps are shown below where they are started at time T = 0 and ended at T = 1000 msec. By running the three threads, this information is gotten: p2 needs 70 time slices, meanwhile p1 and p3 need 2817 and 4205 time slices respectively; Figures 4-6 are the screenshots of the number of slices calculation program that are needed by each thread to finish their whole executions. Therefore, p2 should have higher priority and that would not be discovered without time complexity. Furthermore, the iteration ratio supports time complexity by differentiating threads with the same complexity. So, p2 runs first, and then gives up the processor after time slice number 8 because its priority value rises and becomes equivalent to p1 priority values because of CPU usage counter.

Next, iteration ratio gives the control to p1 since iteration numbers of p1 and p2 are 10000000, 15000000 respectively. Here, the iteration ratio is not applied at time zero because it is always equals 1 for all threads. So, the equation assigns at T = 0 the priorities 61, 60, 62 to p1, p2, and p3 respectively. Below is the actual calculations based on Eq. (1).

T = 0 p2 = 40 + 20 + (0 * 4/32) = 60
p2 takes the control
T = 0 p1 = p2 + 1 = 61
T = 0 p3 = p1 + 1 = 62
T = 10 ms p2 = 40 + 20 + (1 * 4/32) = 60
T = 20 ms p2 = 40 + 20 + (2 * 4/32) = 60
T = 30 ms p2 = 40 + 20 + (3 * 4/32) = 60
T = 40 ms p2 = 40 + 20 + (4 * 4/32) = 60
T = 50 ms p2 = 40 + 20 + (5 * 4/32) = 60
T = 60 ms p2 = 40 + 20 + (6 * 4/32) = 60
T = 70 ms p2 = 40 + 20 + (7 * 4/32) = 60
T = 80 ms p2 = 40 +20 + (8 * 4/32) = 61
p2 releases control
T = 90 ms p1 = 40 + 20 + (9 * 4/32) = 61
……
T = 160ms p1 = 40 + 20 + (16 * 4/32) = 62
p1 releases control

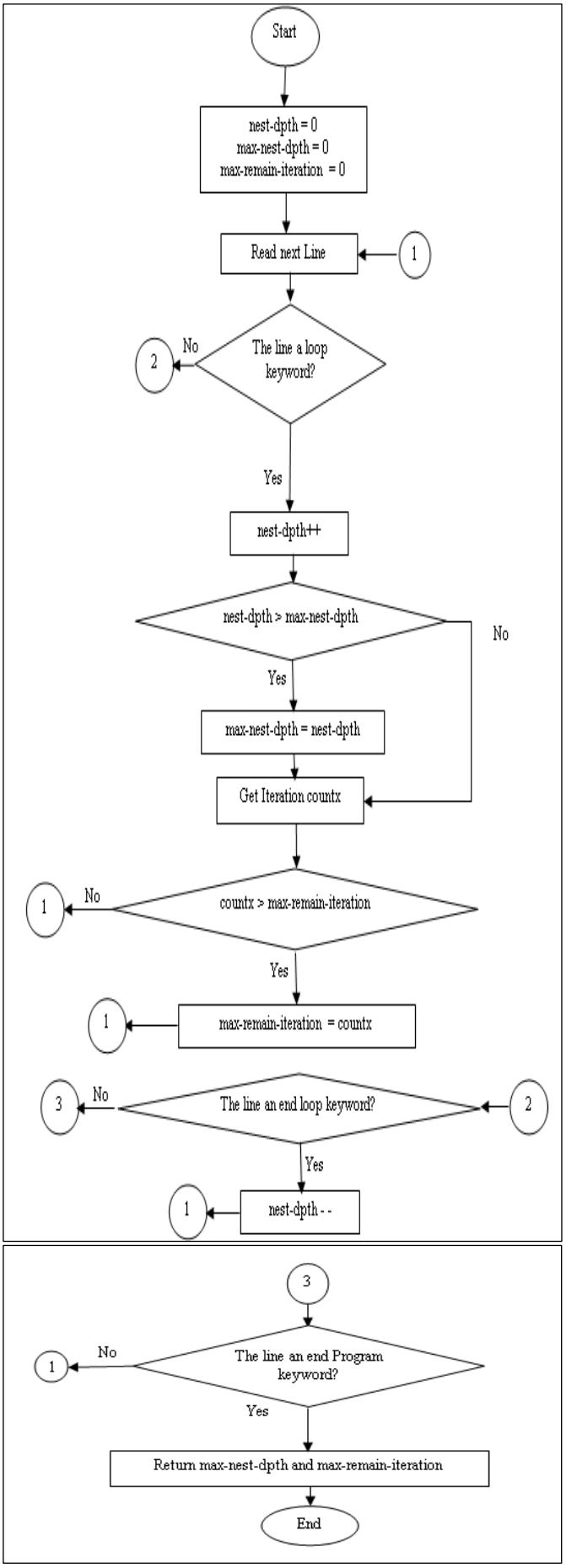

Figure 3: Parser flowchart to get thread complexity according to the count of nested loop with highest depth.
Source: Authors, (2025).

Reset counter = 9
T = 170ms p2 = 40 + 20 + (9 * 4/32) = 61
……
T = 240ms p2 = 40 + 20 + (16 * 4/32) = 62
p2 releases control
Apply Eq. (1):
P1 = 40 + 20 + (17 * 4/32) + ((10000000–26418)/10000000 × 2) = 63.9947164
P2 = 40 + 20 + (17 * 4/32) + ((40000000–49835804)/40000000 × 1) = 62.8754
P3 = 40 + 20 + (17 * 4/32) + (15000000/15000000 × 2) = 64
Therefore, p2 = 62, p1 = 63, p3 = 64, so p2 takes control
T = 250ms p2 = 40 + 20 + (17 * 4/32) = 62
……
T = 320ms p2 = 40 + 20 + (24 * 4/32) = 63
T = 330ms p1 = 40 + 20 + (25 * 4/32) = 63
……
T = 400ms p1 = 40 + 20 + (32 * 4/32) = 64
Reset counter = 25
T = 410ms p2 = 40 + 20 + (25 * 4/32) = 63
……
T = 480ms p2 = 40 + 20 + (32 * 4/32) = 64
Apply Eq. (1):
P1 = 40 + 20 + (33 * 4/32) + ((10000000–26418)/10000000 × 2) = 65.9894326
P2 = 40 + 20 + (33 * 4/32) + ((40000000–9937962)/40000000 × 1) = 64.75155095
P3 = 40 + 20 + (33 * 4/32) + (15000000/15000000 × 2) = 66
Therefore, p1 = 65, p2 = 64, p3 = 66, so p2 takes control

T = 490ms p2 = 40 + 20 + (33 * 4/32) = 64
……
T = 560ms p2 = 40 + 20 + (40 * 4/32) = 65
T = 570ms p3 = 40 + 20 + (41 * 4/32) = 65
……
T = 640ms p3 = 40 + 20 + (48 * 4/32) = 66

(Skipping forward to 1000msec or 1 second)
.
T = 1000ms p2 = 40 + 20 + (60 * 4/32) = 67
T = 1000ms swapper recalculates the accumulated CPU usage counts of all processes. For the above process:
new_CPU_usage = 67 * 31/32 = 64 (if d=31)
After decaying by the swapper: p = 40 + 20 + (64 * 4/32) = 68
Apply the equation:
P1 = 40 + 20 + (64 * 4/32) + ((10000000–26418)/10000000 × 2) = 69.9788174
P2 = 40 + 20 + (64 * 4/32) + ((40000000–15600330)/40000000 × 1) = 68.6099917
P3 = 40 + 20 + (64 * 4/32) + ((15000000-20896)/15000000 × 2) = 69.997213866
Therefore, p1 = 69, p2 = 68, and p3 = 70

Table 2 is a tracing example of p1, p2, and p3, which its explanation is as follows:

At T = 0, p1 = 61, p2 = 60, p3 = 62, therefore, p2 controls the CPU since it has the lowest time complexity.

At T = 90, p2 relinquishes the control since its priority value becomes equivalent to the p1 priority value = 61. Therefore, p1 takes the control since the iteration ratios for both of p1 and p3 =1,

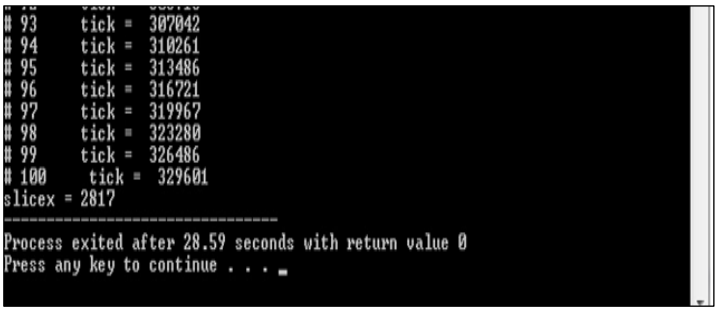

Figure 4: P1's total execution slice number calculation.
Source: Authors, (2025).

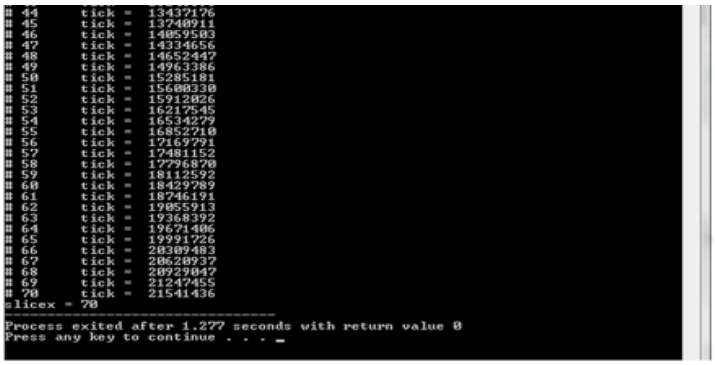

Figure 5: P2's total execution slice number calculation.
Source: Authors, (2025).

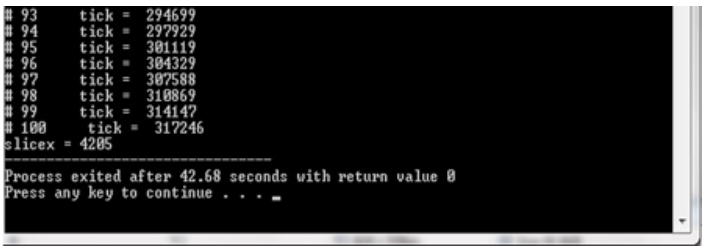

Figure 6: P3's total execution slice number calculation.
Source: Authors, (2025).

Table 2: Tracing of p1, p2, and p3.

T (ms)	p1 Priority	p2 Priority	p3 Priority	CPU control	Counter
0	61	60	62	p2	0
90	61	61	62	p1	1
160	62	61	62	p2	9 reset
240	63	62	64	p2	10
320	63	63	64	p1	11
400	64	63	64	p2	25 reset
480	65	64	66	p2	26
560	65	65	66	p3	27
1000	69	68	70	p2	31

Source: Authors, (2025).

but the number of p1 iterations = 10000000 < 15000000 the number of p3 iterations making p1 = 61 and p3 =62.

At T = 160 ms, p1 gives up the control to p2 again since its value rises to 62 while p2 value = 61. But before that, the algorithm reset the counter to 9 the start of value 61 because it reaches 62 while p2 value = 61 which means that p2 will give up the slice right away. For example, the p2 value after one round if the equation applied without resetting the counter is p2 = 40 + 20 + (17 * 4/32) = 62 making the algorithm useless.

At T = 240 ms, p2 releases the control since its value rises up to 62 and becomes equal to p1 and p3 values. Since all the

threads have equal priorities p1 = p2 = p3 = 63, the equation is applied to assign new real priorities. Therefore, the new priorities are p1 = 63, p2 = 62, and p3 = 64. Therefore, p2 takes the control.

At T = 320 ms, p2 = 63 relinquishes the control to p1 = 63 for the second time.

At T = 400 ms, p1 = 64 relinquishes the control to p2 = 63 after resetting counter to 25.

At T = 480 ms, p2 = 64 relinquishes the control. Since p1 = p2 = p3 = 64, the equation is applied and the new proprieties are: p1 = 65, p2 = 64, and p3 = 66. p2 takes the control.

At T = 560 ms, p2 = 65 releases control to p3 = 66 in spite of p1< p3 since p1 has the control two consecutive times.

At T = 1000 ms (1 sec), swapper recalculates the accumulated CPU usage counter, when thr1, thr2, and thr3 had 32, 51, and 8 time slices respectively and the number of completed iterations for each thread are 26418, 15600330, and 20896 respectively. Therefore, p1 = 69, p2 = 68, and p3 = 70, so p2 takes the control.

From the trace, it is clear that the goal is accomplished since p2 has 51 slices, p1 has 32 slices, and p3 has 8 slices during the period of time 0-1000 ms. Meanwhile, traditional method, which applies Round Robin, gives the same opportunity to the all threads with the sane priorities. The concept of this experiment is giving thread with lowest time complexity and loop iterations more time slices. Therefore, they are involving in the equation whenever the priorities of all threads become equivalent because this state turns the equation to Round Robin and gives equal time slices for every thread. So, the task is giving different priorities for each thread whenever this state occurs. Completing the concept, any state rather than the above one, the time complexity will not involve in the priority calculation, but instead Round Robin is replaced with it. So, every time there are two or more threads with the same priority but not all threads, the time complexity and iteration ratio decide the next thread to control the CPU instead of FIFO, which is used by Round Robin method. To avoid starvation among threads with the same time complexity, the algorithm takes the control from the thread with lower iteration ratio and gives it to the other threads after every two consecutive turns. For example, in this work, thr1 iteration number = 10000000, while thr3 iteration number = 15000000, so thr1 is always taking the control since its iteration number to go is always decreasing, meanwhile thr3 time to go iteration number stays still 15000000.

IV. CONCLUSION

1- This work represents a new generation where is no concept of multithreading with equivalent priorities.
2- The technique acts as Round Robin with multithreading that have constant time for all threads since there is no loops to calculate their iteration ratios
3- This equation rules out the first in first out approach including Round Robin from multithreading system.
4- This work does not work with threads having time complexity involving log, exponential, and factorial times, but extending the parser to include them solve it.
5- The starvation avoidance can be manipulated by changing the number of consecutive call times.
6- The probability that the next time slice is allocated to a thread which has allocated many time slices recently is decreasing.
7- Since time complexity considers the time of iterations' numbers trivia, the equation works more efficient with single-processor than multi-processor.

V. AUTHOR'S CONTRIBUTION

Conceptualization: Abdulamir Abdullah Karim, Yaser Ali Enaya and Ghassan Abdulhussein Bilal.
Methodology: Abdulamir Abdullah Karim, Yaser Ali Enaya.
Investigation: Abdulamir Abdullah Karim and Yaser Ali Enaya and Ghassan Abdulhussein Bilal.
Discussion of results: Abdulamir Abdullah Karim, Yaser Ali Enaya and Ghassan Abdulhussein Bilal.
Writing – Original Draft: Abdulamir Abdullah Karim and Ghassan Abdulhussein Bilal.
Writing – Review and Editing: Abdulamir Abdullah Karim and Yaser Ali Enaya.
Resources: Yaser Ali Enaya and Ghassan Abdulhussein Bilal.
Supervision: Yaser Ali Enaya and Ghassan Abdulhussein Bilal.
Approval of the final text: Abdulamir Abdullah Karim, Yaser Ali Enaya and Ghassan Abdulhussein Bilal.

VI. REFERENCES

[1] Nikolić, Goran, Bojan Dimitrijević, Tatjana Nikolić, and Mile Stojčev. "Fifty years of microprocessor evolution: from single CPU to multicore and manycore systems." Facta universitatis-series: Electronics and Energetics 35, no. 2 (2022): 155-186. https://doiserbia.nb.rs/Article.aspx?ID=0353-36702202155N

[2] He, Zichen, Lu Dong, Changyin Sun, and Jiawei Wang. "Asynchronous multithreading reinforcement-learning-based path planning and tracking for unmanned underwater vehicle." IEEE Transactions on Systems, Man, and Cybernetics: Systems 52, no. 5 (2021): 2757-2769. DOI: 10.1109/TSMC.2021.3050960

[3] Lopez, Tomas A., and Nobuyuki Yamasaki. "Prioritized Asynchronous Calls for Parallel Processing on Responsive MultiThreaded Processor." In 2022 Tenth International Symposium on Computing and Networking (CANDAR), pp. 46-55. IEEE, 2022. DOI: 10.1109/CANDAR57322.2022.00014

[4] Beronić, Dora, Paula Pufek, Branko Mihaljević, and Aleksander Radovan. "On Analyzing Virtual Threads–a Structured Concurrency Model for Scalable Applications on the JVM." In 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), pp. 1684-1689. IEEE, 2021. DOI: 10.23919/MIPRO52101.2021.9596855

[5] Tsai, Chun-Jen, and Yan-Hung Lin. "A Hardwired Priority-Queue Scheduler for a Four-Core Java SoC." In 2018 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1-4. IEEE, 2018. DOI: 10.1109/ISCAS.2018.8351129

[6] https://www.ibm.com/docs/en/aix/7.1?topic=calculation-priority.

[7] Syuhada, Rahmad. "Multi-threading on Linux Operating System Using Scheduling Algorithm." Jurnal Mantik 5, no. 2 (2021): 1334-1340. https://iocscience.org/ejournal/index.php/mantik/article/view/1506

[8] Kalla, Ron, Balaram Sinharoy, and Joel M. Tendler. "IBM Power5 chip: A dual-core multithreaded processor." IEEE micro 24, no. 2 (2004): 40-47. DOI: 10.1109/MM.2004.1289290

[9] Liu, Zihan, Ikuta Tanigawa, Harumi Watanabe, and Kenji Hisazumi. "PLAM: Preemptive Layer Activation Architecture based on Multithreading in Context-Oriented Programming." In Proceedings of the 12th ACM International Workshop on Context-Oriented Programming and Advanced Modularity, pp. 1-8. 2020. https://doi.org/10.1145/3422584.3422766

[10] Albazaz, Dhuha. "Design a mini-operating system for mobile phone." Int. Arab J. Inf. Technol. 9, no. 1 (2012): 56-65. https://www.iajit.org/PDF/vol.9,no.1/1614-7.pdf

[11] Yaser Ali Enaya. "Password-free Authentication for Smartphone Touchscreen Based on Finger Size Pattern", International Journal of Interactive Mobile Technologies, vol. 14, no. 19, 2020, pp. 163–179. DOI: 10.3991/ijim.v14i19.17239.

[12] Sallow, Amira B. "Android Multi-threading Program Execution on single and multi-core CPUs with Matrix multiplication." International Journal of Engineering & Technology 7, no. 4 (2018): 6603-6608. DOI: 10.14419/ijet.v7i4.29340

[13] Khalid, Zubair, Usman Khalid, Mohd Adib Sarijari, Hashim Safdar, Rahat Ullah, Mohsin Qureshi, and Shafiq Ur Rehman. "Sensor virtualization Middleware design for Ambient Assisted Living based on the Priority packet processing." Procedia Computer Science 151 (2019): 345-352.https://doi.org/10.1016/j.procs.2019.04.048

[14] Enaya, Yaser Ali, and Kalyanmoy Deb. "Network path optimization under dynamic conditions.", In 2014 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2014, pp. 2977-2984. DOI: 10.1109/CEC.2014.6900603

[15] Enaya, Yaser Ali, Abdulamir Abdullah Karim, Suha Mohammed Saleh, and Salam Waley Shneen. "Adapting Wired TCP for Wireless Ad-hoc Networks Using Fuzzy Logic Control." Journal Européen des Systèmes Automatisés 57, no. 5, (2024). pp. 1377-1386. https://doi.org/10.18280/jesa.570513

[16] Yaser, E., Abdulamir Abdullah Karim, Mohammed Qasim Sulttan, and Salam Waley Shneen. "Applying Proportional–Integral–Derivative Controllers on Wired Network TCP's Queue to Solve Its Incompatibility with the Wireless Ad-Hoc Network." ITEGAM-JETIA 10, no. 49 (2024): 228-232. https://doi.org/10.5935/jetia.v10i49.1346

[17] Osborne, Sims Hill, Shareef Ahmed, Saujas Nandi, and James H. Anderson. "Exploiting simultaneous multithreading in priority-driven hard real-time systems." In 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1-10. IEEE, 2020. DOI: 10.1109/RTCSA50079.2020.9203575

[18] Shomron, Gil, and Uri Weiser. "Non-blocking simultaneous multithreading: Embracing the resiliency of deep neural networks." In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 256-269. IEEE, 2020. DOI: 10.1109/MICRO50266.2020.00032

[19] Mapetu, Jean Pepe Buanga, Zhen Chen, and Lingfu Kong. "Low-time complexity and low-cost binary particle swarm optimization algorithm for task scheduling and load balancing in cloud computing." Applied Intelligence 49 (2019): 3308-3330. https://doi.org/10.1007/s10489-019-01448-x

[20] Asif, Muhammad, Muhammad Adnan Khan, Sagheer Abbas, and Muhammad Saleem. "Analysis of space & time complexity with PSO based synchronous MC-CDMA system." In 2019 2nd international conference on computing, mathematics and engineering technologies (iCoMET), pp. 1-5. IEEE, 2019. DOI: 10.1109/ICOMET.2019.8673401

[21] Mapetu, Jean Pepe Buanga, Zhen Chen, and Lingfu Kong. "Low-time complexity and low-cost binary particle swarm optimization algorithm for task scheduling and load balancing in cloud computing." Applied Intelligence 49 (2019): 3308-3330. https://doi.org/10.1007/s10489-019-01448-x

[22] Chao, Zemin, Hong Gao, Yinan An, and Jianzhong Li. "The inherent time complexity and an efficient algorithm for subsequence matching problem." Proceedings of the VLDB Endowment 15, no. 7 (2022): 1453-1465. https://doi.org/10.14778/3523210.3523222

[23] Ramirez, Anthony, and Vyron Vellis. "Time complexity of the Analyst's Traveling Salesman algorithm." arXiv preprint arXiv: 2202.10314 (2022). https://doi.org/10.48550/arXiv.2202.10314

[24] Abd Almahdi, Wijdan, Hussein Attia Lafta, and Yossra Hussain Ali. "Intelligent Task Scheduling Using Bat and Harmony Optimization." Iraqi Journal of Science (2023): 4187-4197. DOI: https://doi.org/10.24996/ijs.2023.64.8.38

[25] Suha Dh. Athab, Abdulamir A. Karim. A "Tagging Model using Segmentation Proposal Network". Fusion: Practice and Applications. 2023; 13(2): 136-144. https://doi.org/10.54216/FPA.130212.

[26] Attiya, Hagit, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. "Detectable recovery of lock-free data structures." In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 262-277. 2022. https://doi.org/10.1145/3503221.3508444

[27] Sánchez, Jesus Gerardo Ávila, Francisco Eneldo López Monteagudo, Francisco Javier Martinez Ruiz, and Leticia del Carmen Ríos Rodríguez. "Detection of traffic accidents using artificial intelligence." ITEGAM-JETIA 10, no. 46 (2024): 15-21. DOI: https://doi.org/10.5935/jetia.v10i46.1109.

[28] Zhao, Shuai, Xiaotian Dai, and Iain Bate. "DAG scheduling and analysis on multi-core systems by modelling parallelism and dependency." IEEE transactions on parallel and distributed systems 33, no. 12 (2022): 4019-4038. DOI: 10.1109/TPDS.2022.3177046

[29] Ahmed WS, Abdul amir A. Karim. "The impact of filter size and number of filters on classification accuracy in CNN". In2020 International conference on computer science and software engineering (CSASE) 2020 Apr 16 (pp. 88-93). IEEE. DOI: 10.1109/CSASE48920.2020.9142089.

[30] Xu, Y., Liu, S., & Wang, Z. (2022). "Complexity Analysis of a Parallel Algorithm for the All-Pairs Shortest Paths Problem on Road Networks." IEEE Transactions on Parallel and Distributed Systems 33(9), 2205-2218.

[31] Abdulateef, Isra H., and Dhia A. Alzubaydi. "An Evolutionary Algorithm with Gene Ontology-Aware Crossover Operator for Protein Complex Detection." Iraqi Journal of Science (2023): 1975-1987. https://doi.org/10.24996/ijs.2023.64.4.34

[32] Zhou, Houji, Yi Li, and Xiangshui Miao. "Low-time-complexity document clustering using memristive dot product engine." Science China Information Sciences 65, no. 2 (2022): 122410. https://doi.org/10.1007/s11432-021-3316-x

[33] Ayad, Hayder, Nidaa Flaih Hassan, and Suhad Mallallah. "A modified segmentation approach for real world images based on edge density associated with image contrast stretching." Iraqi Journal of Science (2017): 163-174]. https://ijs.uobaghdad.edu.iq/index.php/eijs/article/view/6237

[34] Sohrabi, Somayeh, Koorush Ziarati, and Morteza Keshtkaran. "Revised eight-step feasibility checking procedure with linear time complexity for the Dial-a-Ride Problem (DARP)." Computers & Operations Research 164 (2024): 106530. https://doi.org/10.1016/j.cor.2024.106530

[35] Shi, Feng, Frank Neumann, and Jianxin Wang. "Time complexity analysis of evolutionary algorithms for 2-hop (1, 2)-minimum spanning tree problem." Theoretical Computer Science 893 (2021): 159-175. https://doi.org/10.1016/j.tcs.2021.09.003

[36] BH, Krishna Mohan, Padmaja Pulicherla, M. Purnachandrarao, and P. Nagamalleswararao. "Quantum machine learning: bridging the GAP between classical and quantum computing." ITEGAM-JETIA 10, no. 48 (2024): 122-128. DOI: https://doi.org/10.5935/jetia.v10i48.943

[37] Menghani, Gaurav. "Efficient deep learning: A survey on making deep learning models smaller, faster, and better." ACM Computing Surveys 55, no. 12 (2023): 1-37. https://doi.org/10.1145/3578938

[38] Ghosh, Sourav Kumar, Sumon Hossain, Hafijur Rahman, Naurin Zoha, and Mohammad Arif-Ul Islam. "Developing a linear programming model to maximize profit with minimized lead time of a composite textile mill." ITEGAM-JETIA 6, no. 22 (2020): 18-21. DOI:https://dx.doi.org/10.5935/2447-0228.20200012