Iraqi Journal of Industrial Research, Vol. 10, No. 3 (2023)

Open Access

Iraqi Journal of Industrial Research (IJOIR)

Journal homepage: http://ijoir.gov.iq

An Improved Dynamic Slicing Algorithm to Prioritize a Concurrent Multi-threading in Operating System

Maysoon A. Mohammed*

Department of Mechanical Engineering, University of Technology – Iraq

Article information

Article history:
Received: April, 14, 2023
Accepted: June, 17, 2023
Available online: December, 14, 2023

Keywords:
Multithreading,
Prioritization,
Dynamic Slicing Algorithm

Corresponding Author:
Maysoon A. Mohammed
maysoon.a.mohammed@uotechnology.edu.iq

DOI:
https://doi.org/10.53523/ijoirVol10I3ID331

Abstract

One of the issues with multi-threading in operating systems is the concurrency of operations or threads. In a multithreaded process on a single processor, the processor can switch execution resources between threads, enabling concurrent execution. Concurrency indicates that more than one thread is making progress, but the threads are not actually running simultaneously. The switching between threads occurs rapidly enough that the threads might appear to run simultaneously. In this paper, three related strategies for prioritizing multi-threading are presented: ACE-thread, Semaphore coprocessor, and the Concurrent Priority Threads Algorithm. The aim of this work is to enhance an existing prioritization algorithm, specifically the Concurrent Priority Threads Algorithm, by extending a dynamic slicing algorithm to prioritize multi-threading concurrently. The algorithm is designed to compute correct slices in multi-threading prioritization scenarios. Threads with the same highest priority can perform in a synchronized manner without encountering deadlocks. The C++ programming language is used to implement the extended algorithms. The improved algorithm achieved results that were 3% more accurate than the existing one. The outcomes of this work would facilitate the simultaneous execution of threads with the same priority, ultimately reducing waiting and processing times.

1. Introduction

Of late, multi-threading advanced into a standard way to upgrade processor utilization and program effectiveness. The dynamic program slice can be an application component that ‘‘affects’’ the working out of a movable with respect to intrigued all through program execution over a specific framework input. Dynamic program slicing depicts a sum of program slicing methods that depend on program execution and may indeed altogether diminish the measure of an application slice essentially since run-time information, collected amid program execution, is utilized in arrange to figure out framework slices. Because data is constantly growing, it is becoming denser and more diverse by the minute across multiple channels. Consumers are creating large amounts of data on a daily basis, so big data has appeared and is rapidly evolving [1]. It has become necessary to control this big data, and multi-threading is one of the important mechanics to use in controlling big data.

Multi-tasking runs multiple programs simultaneously and, in the programming language, such a phenomenon is named multithreading. The modern computers have become very fast for a human to interpret the switching mechanism by running multiple threads in parallel on a single processor. Thus, the interpreter would switch from one thread to another. A multi-threaded program begins with a single thread of execution, and alternative threads can be created later. Typically, a multithreaded program comprises a (global) principal thread and many other threads. Other threads are started from the main thread. For instance, a multi-threaded program comprises a principal thread and other threads named X, Y and Z. Also, the principal thread considers the initiating point as well as initiating point of execution of the threads is created. Also, the local threads X, Y, and Z can be the main thread of other threads, etc., as shown in Figure (1) below:

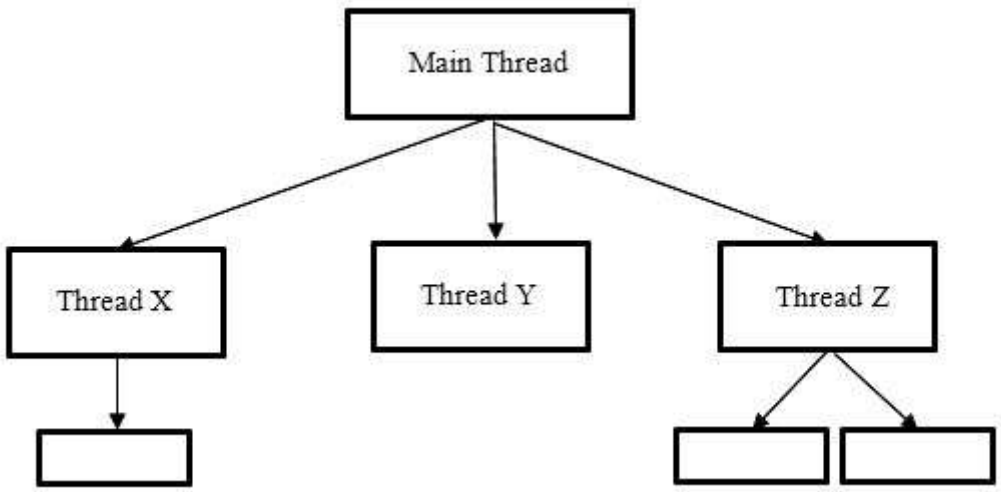

Figure (1). Thread Initialization.

Most of the applications that used by a user through a computer or even a mobile device is multi-threaded, which means, that several tasks are executed at the same time. For example, when using the Internet, the web browser is implemented by more than one thread, one thread that displays images and another thread that retrieves data from the network. As another example, a Microsoft Word application might have multiple threads to display graphics, respond to user keystrokes, and run spelling and grammar checks in the background. And, for instance, if a user needs printing and reading a report from files simultaneously, he can achieve this by running multiple threads running in parallel. So, a single thread would correspond to print the report, as well as another thread would correspond to the procedure of reading the report from the file. One solution to concurrent operations is to break a complex program into simpler segments, or tasks, and run them in parallel. This would speed up the execution of a complex program to reduce the execution speed significantly. The processor processes the execution threads that have priority in sequential order. Depending on the type of thread processing, there are three priority levels: highest, normal, and lowest. For example, background tasks like display customization should have the lowest priority, while error detection or file updating tasks should have the highest priority. Normally, the thread with the highest priority runs first, followed by the thread with the lower priority. If multiple threads have the highest priority, the CPU time cannot be shared among threads with the same priority, resulting in a deadlock situation. A dynamic allocation algorithm called Concurrent Priority Threads Algorithm (CPTA) [2] has been extended to ensure three things: multiple threads run smoothly, increase CPU speed, and prevent CPU damage from incoming multi-threads with the same priority. This work focuses on making multiple threads run smoothly to reduce the processing time by improving the Concurrent Priority Threads Algorithm (CPTA) to the Improved Concurrent Priority Threads Algorithm (ICPTA). Because different dynamic slice times will be defined based on the priorities of the threads, the shortest threads will be able to complete their task before the longest threads. This reduces waiting time compared to the existing approach, which primarily focuses on CPU time and increases the risk that the shortest threads may starve to death. Additionally, context shifts are reduced when each thread's dynamic selection time is divided by its count by using a suggested formula. Therefore, utilizing the provided method to combine different priorities with dynamic slice time helps to maximize CPU performance.

The classification of this article is as follows: section 2 is the theoretical part, where in this section, the proposed algorithm and the proposed equation have been explained. Section 3 is related work, because the number of pieces of research and methods used in this field are few, three methods have been discussed and compared in this section. In section 4 which is the proposed system, the details of the proposed algorithm, the steps of the algorithm and the flowchart of the algorithm have been discussed. Section 5 is the experimental procedure and results. Finally, section 6 is the conclusion.

2. Theoretical Part

Traditionally, these reactive software systems were designed as multiprocessor or multi-threaded shared memory programs [3]. One of the options available to the user, depending on the scale of the experiment, is for creating a governing system from a scratch, employing a chosen programming language as well as a working regime, using helpful guides and various libraries [4]. The program P contains all the statements belonging to the program section, which assign the value of the variable v to the corresponding place p [5].

Dynamic Slicing brings further advantages in debugging, testing, program understanding, software maintenance and presence on Android systems (e.g., [6], [7], [8]) and [9].

This article focuses on modifying an effective dynamic slicing algorithm for the multi-threaded programs with equivalent priority threads. And, the algorithm would effectively help more than two threads being processed with the uppermost primacy. Also, the multi-threaded programs requiring locking as well as unlocking of crucial section resources operate synchronously. Additionally, the algorithm would calculate exact slices by getting the execution dynamic traces in a multi-threaded program using a new equation. The sections correspond to the program instructions influenced by the dynamic criterion of partitioning. Dynamic slicing is an advantageous method that would aid synchronizes multi-threaded programs and accelerates execution speed with absolute priority for threads.

In this article, a new dynamic formula has been added to the existing algorithm (CPTA) where enhanced to (ICPTA), the formula is according to Equation 1:

$$\mathbf{Pi = v \times (E \ \% \ S)} \qquad \mathbf{1}$$

Where: pi is the priority of thread I, S is the place S that's the declaration into the program, v represents the object or variable utilized in the targeted declarations, and E represents the implementation hint with the enter i furnished dynamically at the working period.

Each thread is given a different priority by the formula above, and this article tends to benefit from the priorities by mixing them with the other thread-related components, S, v, and E. This formula first performs a modulation operation between the location S and the reference time E, after which the result of the modulation is multiplied by the object v to produce a new slice time for each thread. This formula makes sure that each thread is given a reasonable amount of time. Because the processing time of the thread will be shortened using the rational time, this guarantees that the lowest priority thread won't be starved by the highest priority threads.

3. Related Work

The first method is ACE-thread which depends on three resources: the application, the control device, and the external data. The idea of this method is, each thread has its own priority, and the sending unit will serve the threads according to their priority. And, the priority p of thread i is calculated as follows: pi = min(ai, ei) + ri where ri = min(mi, bi + si). Where, bi represents the lowermost or the base priority of thread i; mi represents the highest priority, and si represents the offset for adjusting the present priority according to the use, set by the user or the governing device. ei is an external data bound by ai to the priority taken via the use as well as governing unit, or even to a fresh but a higher ultimate limit value. When the operating system wants to quickly execute external interrupt routines or release important resources as quickly as possible, the function of increasing the priority for a short period of time will be useful in this case [10]. Figure (2) evinces the range of single and multi-threaded priorities that can be used at runtime.

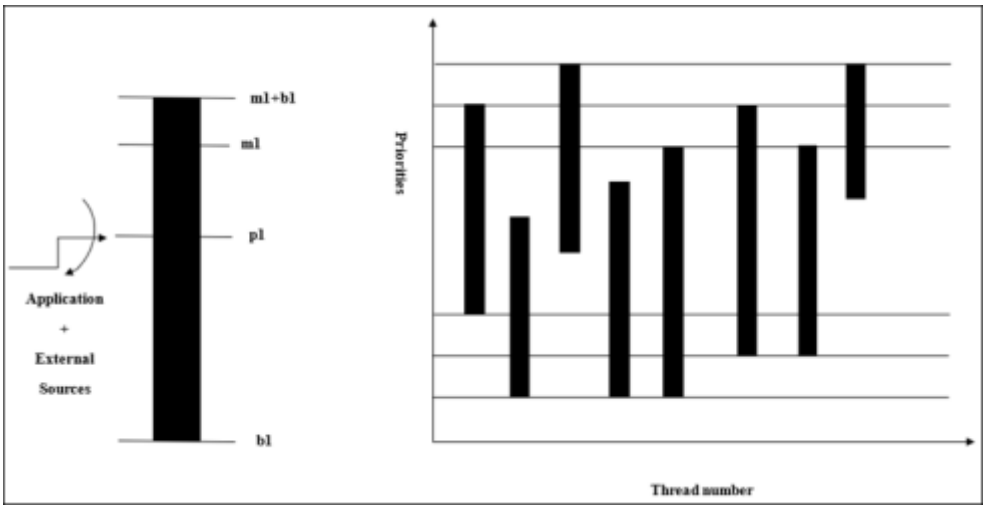

Figure (2). Range of Priority for single and multiple threads [10].

The second method is the Semaphore coprocessor, it is an extension of the ACE technique by placing some kind of semaphore coprocessor seeing that every other has an impact on in which threads have an effect on every different and additionally which normally makes use of the exterior enter pertaining to coping with the thread priority. Selecting the semaphore coprocessor covers the factors, thread stalls due to the synchronization between the software program threads and the additional synchronization that has a hardware unit. Equally being generally associated with the several components of embedded structures as well as the additional servers [11]. And, the semaphore coprocessor, as elucidated in Figure (3), handles a configurable range of resources. Each aid has a queue plus a token Si, i ∈ 0..k. Also, the lock and additionally unencumber guidelines generally are put into the processor's preparation set, thus the unit of problem passes the whole directions toward the coprocessor. A positive token is requested by means of a lock instruction. When the training arrives and if the token is accessible then the coming education will be served at once. Else, it's buffered, and the thread is paused till the occupying thread frees the token. When a thread asks for a token and this token is allotted by using every other thread, then to keep away from the lengthy pause for the ready thread the precedence of the allocation thread will be raised. To perform Semaphore coprocessor, two instructions to regulate the priority must be followed which are:

1. A static price ei would increase the precedence so that the precedence of a thread is continually will be the easiest if there is at least one thread requiring the identical token, or
2. It is improved with the aid of a dynamic fee ei. In this case, ei relies upon the range of threads ready for the token. So, ei = c ∗ #queue the place c is steady and #queue is the modern-day queue size [12].

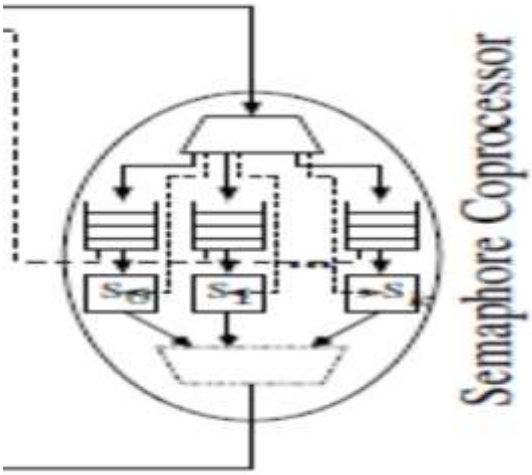

Figure (3) Semaphore coprocessor [12].

The programming languages which have an object-oriented base are greater appropriate than the different languages for the dynamic reducing technique which is the third method called Dynamic Slicing Algorithm. The dynamic slicing algorithm identifies the input and considers the distinctive incidences of an announcement in an implementation hint [13]. The entered and the software announcement are belonging to the run hint which in flip belongs to the dynamic slicing algorithm. The approach works with the aid of computing slices on an exceptional kind of design known as System Dependence Graph based totally on a unique reducing criterion, the place is the application announcement, and V represents the applied variable for which the slice is calculated [14].

The execution of many threads simultaneously beginning with a world thread can be achieved with the aid of abstraction of two sorts of dependence named Dynamic thread dependence design (DTDG) and Priority multithreaded dependence sketch (PMDG).

A comparison of the three above methods to four categories (time, aspect, precision, and system type) is manifested in Table (1).

Table (1). Comparison of multithreading Techniques.

Methods / Findings	ACE	Semaphore coprocessor	Dynamic Slicing-threads
Time	Reduce time with very low cost	Reduce time with resource increasing	Reduces Time Processing
Aspect	S.W., M.W. and H.W.	H.W.	S.W.
Accuracy	YES	YES, with more needs for processor resources as for Semaphore	YES
Kind of Systems	Independent	Independent	Independent

In this research we focused on the third method which is dynamic slicing threads.

4. Proposed System

The slicing algorithm for the multi-threaded packages founded totally upon the precedence gets the implementation hint of the application at the working period. And, the algorithm calculates the whole threads having the most precedence for the CPU planning function. Also, the threads having minimal precedence are carried out ultimately beyond the excessive precedence threads. In addition, the algorithm is referred to as Improved Concurrent Priority Threads Algorithm (ICPTA) as well as would be calculating the whole threads in the multithreaded application having the identical allocated primacy using a new equation. Furthermore, the threads would be successively synchronized for the treatment via CPU scheduling. Moreover, the dynamic slicing criterion got for the such algorithm being the place S that's the declaration into the program, v represents the object or variable utilized in the targeted declarations, E represents the implementation hint with the enter i furnished dynamically at the working period [15]. Such an algorithm creates certain that the threads with high primacies are similarly treated via using the CPU via the simultaneous hanging as well as restarting of threads. And, the threads can additionally be despatched to the condition of sleep for a distinctive duration of the period in milliseconds.

Similar to other studies conducted by other researchers like [16], the altering of slice time has a positive impact on the CPU's performance. Slice time is crucial issue in CPUs since short time slices lead to huge context flips and long-time slices lengthen waiting times [17]. To determine which thread should control the CPU first, Dynamic slice time and priorities are more than adequate. Using dynamic slice time prevents this from happening by ensuring that each thread receives a reasonable amount of CPU time based on its assigned priority. The lowest priority thread could be starved from the highest priority thread otherwise. Depending on the threads' slice times, the thread priority is determined.

The ICPTA steps:

Input: The threads in a priority founded multi-threaded program, the dynamic slicing factors <S, v, E, i>.

Output: The processed threads having the uppermost equal priorities.

Step 1: Construct the PMDG of the provided priority founded multithreaded program

Step 2: Take the slicing factors <S, v, E, i>, where the input (i) represents the ultimate priority thread got dynamically at the working period as well as repeat the as follows:

a- Repeat while thread X is stopped; and thread Y is stopped;

b- thread X is started; and thread Y is started;

c-for X=1 to n and for Y=1 to n;

d- If (thread X==MAX_PRIORITY),

e- then, Print thread X processing

f- If (thread Y==MAX_PRIORITY),

g- then Print thread Y processing

h- If (thread X and thread Y==MAX_PRIORITY),

i- then calculating a new time.

j- Using the formula i=v*(E % S),

k- print thread X and thread Y processing.

l- else, print thread X is pre-empted; thread X is suspended

m- else, print thread Y is pre-empted; thread Y is suspended

Step 3: Restart the hanging threads

(a) thread X is resumed; Print thread X is processed

(b) thread Y is resumed; Print thread Y is processed

Step 4: End

The thread flow of the proposed algorithm (ICPTA) and existing algorithm (CPTA) is shown in Figures (4 & 5).

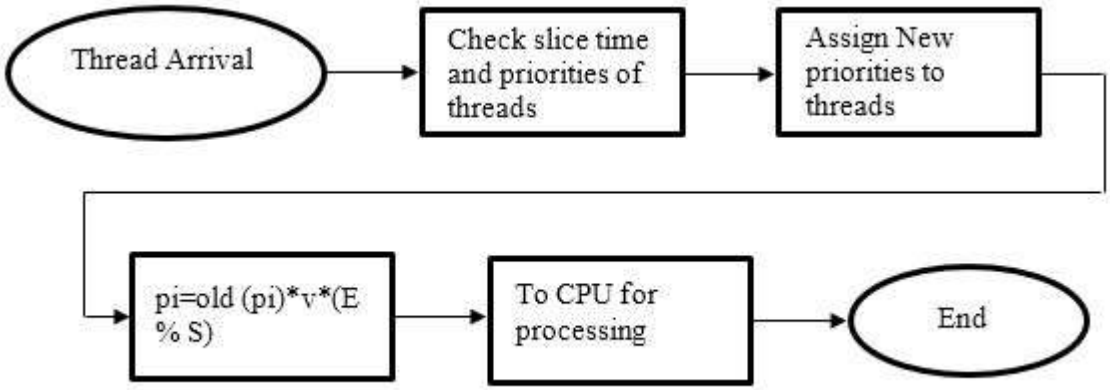

Figure (4). Thread Flow of ICPTA.

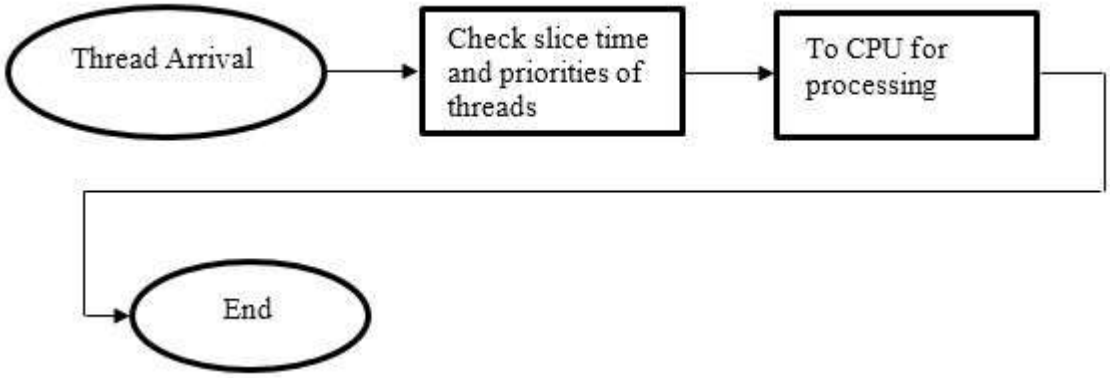

Figure (5). Thread Flow of CPTA.

From Figure (4), the threads join to the end of the ready queue. The scheduler then assigns a new priority for each thread based on its slice time. After assigning priorities, a new dynamic slice time is generated for each thread based on their priorities using with the proposed formula (Equation 1). The scheduler picks the thread from the ready queue and allocates the CPU to the current thread, completes the job, and leaves the system.

In comparison to the current algorithm, which primarily concentrates on CPU time more than the possibility of starving the shortest threads, which leads to increase the waiting time, in the proposed algorithm, the shortest threads will be able to finish their work before the longest threads because various dynamic slice times will be set based on the priorities of the threads. Additionally, dividing each thread's dynamic selection time by its number (using the suggested Equation 1) results in fewer context shifts. Therefore, combining various priority with dynamic slice time using the suggested formula aids in maximizing CPU performance.

5. Experimental Procedure and Results

The goal of the Improved Concurrent Priority Threads Algorithm experiment is to reduce the CPU processing time by using only the instructions that affect the thread segment. Even during thread execution time, the algorithm increases the speed of the highest priority threads. Figure 6 is a bar chart showing the threads with their respective priorities; We see threads starting with the highest for thread Q and decreasing by a total of threads until the lowest priority is reached for thread X (i.e. execution is sequential). The instructions for this graph in the program take the code:

```
class threadPriority
{
    void main()
    {
        X threadX=new X();
        Y threadX=new Y();
        Z threadX=new Z();
        M threadX=new M();
        N threadX=new N();
        O threadX=new O();
        P threadX=new P();
        Q threadX=new Q();
        threadQ.setPriority(thread.MAX_PRIORITY);
        threadP.setPriority(threadO.getPriority()+1);
        threadO.setPriority(threadN.getPriority()+1);
        threadN.setPriority(threadM.getPriority()+1);
        threadM.setPriority(threadZ.getPriority()+1);
        threadZ.setPriority(threadY.getPriority()+1);
        threadY.setPriority(threadX.getPriority()+1);
        threadX.setPriority(thread.MIN_PRIORITY);
```

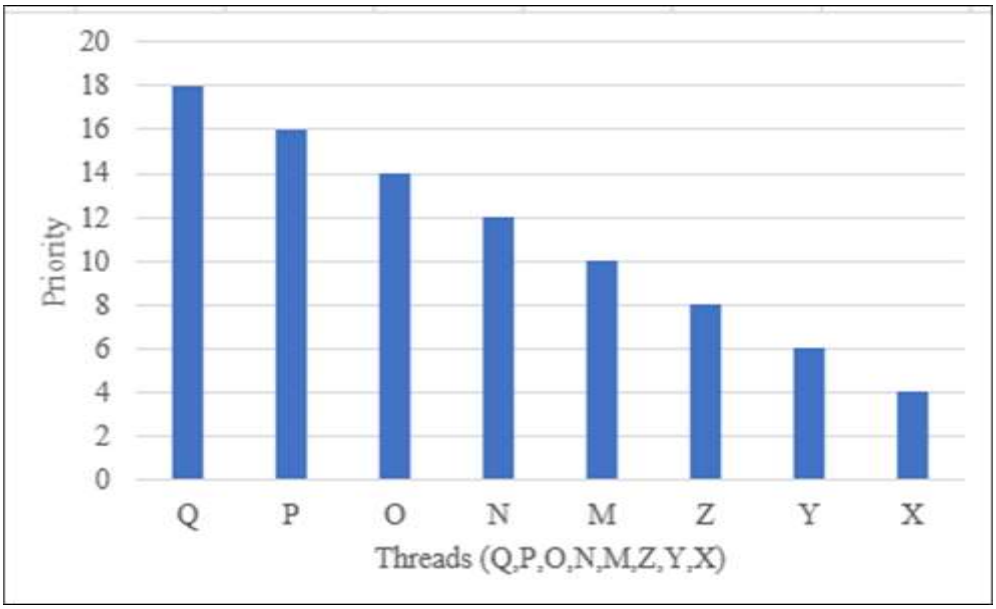

Figure (6). Priorities of the threads.

```
class threadPriority
{
    void main()
    {
        X threadX=new X();
        Y threadX=new Y();
        Z threadX=new Z();
        M threadX=new M();
        N threadX=new N();
        O threadX=new O();
        P threadX=new P();
        Q threadX=new Q();
        threadQ.setPriority(thread.MAX_PRIORITY);
        threadP.setPriority(threadO.getPriority()+1);
        threadO.setPriority(threadN.getPriority()+1);
        threadN.setPriority(threadM.getPriority()+1);
        threadM.setPriority(threadZ.MAX_PRIORITY);
        threadZ.setPriority(threadY.getPriority()+1);
        threadY.setPriority(threadX.MAX_PRIORITY);
        threadX.setPriority(thread.MIN_PRIORITY);
        i=v*(E % S);
```

In the code above, we prioritize threads first, so threads Q, M and Y have the highest priorities and other threads have lower priorities. Secondly, we could get the result that the three highest priority threads Q, M and Y run concurrently, followed by other lower priority threads after running to the second time as shown in Figure (7).

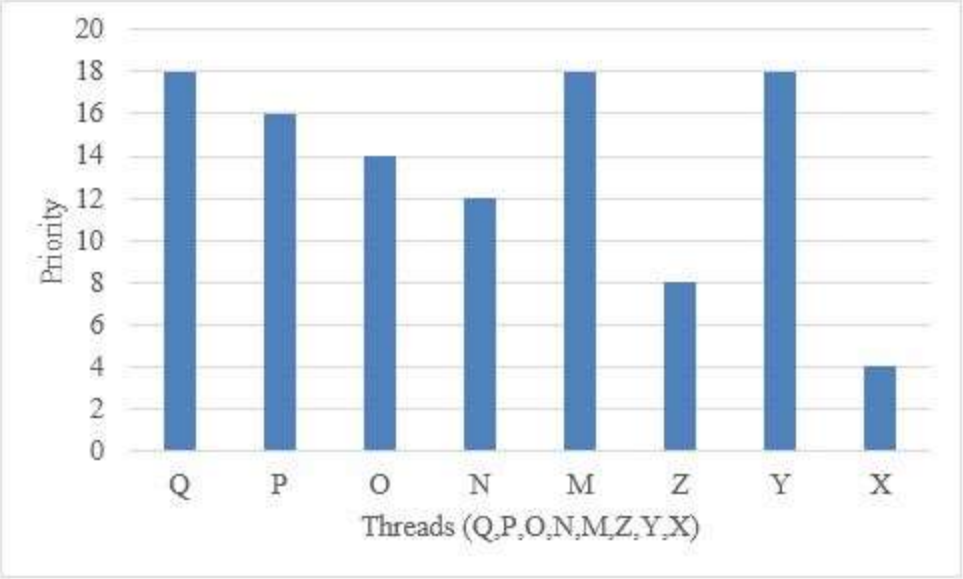

Figure (7). Priorities concurrently for threads Q, M and Y.

Table (2). Experiment Summary.

Algorithm	Number of Threads	Number of Replication	Performance Factors	
			Waiting Times	Context Switches
CPTA	X, Y, Z, M, N, O, P and Q 8 Threads	1	Conducted	Conducted
ICPTA	X, Y, Z, M, N, O, P and Q 8 Threads	1	Reduced to about 3%	Reduced to about 3%

The percentages from the experiment that was already carried out in this section are shown in Table (2). Context Switches and Waiting Time percentages have been developed and compared for the existing and proposed methods (CPTA and ICPTA) after data collection and calculations. The outcome indicates an increase in the ICPTA above CPTA of roughly 3%.

6. Conclusions

In situations where there is multi-threading prioritizing, the technique would generate accurate slices. The performance of the threads with the same highest priority may be synchronized without thread deadlock. Since the C++ programming language is the finest language for managing threads concurrently, it is used to apply the expanded algorithms. The new algorithm produces findings that are 3% more accurate than the old one. The results of this work will enable threads with the same priority to run concurrently, reducing waiting and processing times. In future work we will consider adding more dual processor threads and making threads dependent on each other and because the issues that brought by the intricacy of communication between various applications and underlying Quality of service architectures (Qos) [18], which reduces the usefulness of QoS provisioning, so, idea for future work trying to increase the benefit of Qos by incentivizing threads in real time systems.

Conflict of Interest: The authors declare that there are no conflicts of interest associated with this research project. We have no financial or personal relationships that could potentially bias our work or influence the interpretation of the results.

References

[1] N. Saeed and L. Husamaldin, "Big data characteristics (V's) in industry," *Iraqi Journal of Industrial Research,* vol. 8, pp. 1-9, 2021.

[2] M. A. Mohammed, M. A. Majid, M. A. Ibrahim, and B. A. Mustafa, "Multithreading Prioritization Concurrently By using an effective Dynamic Slicing Algorithm," in *The 3rd International Conference on Computer Engineering and Mathematical Sciences (ICCEMS 2014)*, Maiaysia, 2014, pp. 709-713.

[3] S. W. Beyene and J.-H. Han, "Prioritized Hindsight with Dual Buffer for Meta-Reinforcement Learning," *Electronics,* vol. 11, p. 4192, 2022.

[4] A. Steed, L. Izzouzi, K. Brandstätter, S. Friston, B. Congdon, O. Olkkonen*, et al.*, "Ubiq-exp: A toolkit to build and run remote and distributed mixed reality experiments," *Frontiers in Virtual Reality,* vol. 3, 2022.

[5] N. Fuhrberg, "Isolating Cross-Domain Links with SDN based E2E Network Slices," 2022.

[6] I. Postolski, V. Braberman, D. Garbervetsky, and S. Uchitel, "Dynamic Slicing by On-demand Re-execution," *arXiv preprint arXiv:2211.04683,* 2022.

[7] S. Panda, D. Munjal, and D. P. Mohapatra, "A slice-based change impact analysis for regression test case prioritization of object-oriented programs," *Advances in Software Engineering,* vol. 2016, p. 20, 2016.

[8] V. Lenarduzzi, A. Sillitti, and D. Taibi, "Analyzing forty years of software maintenance models," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 146-148.

[9] S. Wei, "The Design and Implementation of a Mobile Learning Platform Based on Android," in *2013 International Conference on Information Science and Cloud Computing Companion*, 2013, pp. 345-350.

[10] G. K. Adam, "Co-Design of Multicore Hardware and Multithreaded Software for Thread Performance

Assessment on an FPGA," *Computers,* vol. 11, p. 76, 2022.
[11] N. Rother, T. Stuckenberg, S. Nolting, C. Uhlemann, and H. Blume, "A case study on multi-softcore aided hardware architectures for powerline MAC-layer," presented at the Konferenzschrift, ICT.OPEN, Germany, 2023.
[12] C. Albrecht, A. C. Doring, F. Penczek, T. Schneider, and H. Schulz, "Impact of coprocessors on a multithreaded processor design using prioritized threads," in *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, 2006, p. 7 pp.
[13] C. Galindo, S. Pérez, and J. Silva, "Program Slicing Techniques with Support for Unconditional Jumps," in *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings*, 2022, pp. 123-139.
[14] F. Ullah, J. Wang, S. Jabbar, F. Al-Turjman, and M. Alazab, "Source code authorship attribution using hybrid approach of program dependence graph and deep learning model," *IEEE Access,* vol. 7, pp. 141987-141999, 2019.
[15] W. Lulu, L. Bixin, and K. Xianglong, "Type slicing: An accurate object oriented slicing based on sub-statement level dependence graph," *Information and Software Technology,* vol. 127, p. 106369, 2020.
[16] M. T. Ogedengbe and M. A. Agana, "New fuzzy techniques for real-time task scheduling on multiprocessor systems," *International Journal of Computer Trends and Technology,* vol. 47, pp. 189-196, 2017.
[17] M. Iqbal, Z. Ullah, I. A. Khan, S. Aslam, H. Shaheer, M. Humayon*, et al.*, "Optimizing Task Execution: The Impact of Dynamic Time Quantum and Priorities on Round Robin Scheduling," *Future Internet,* vol. 15, p. 104, 2023.
[18] H. W. Oleiwi, H. L. Al-Taie, N. Saeed, and D. N. Mhawi, "A Comparative Investigation on Different QoS Mechanisms in Multi-Homed Networks," *Iraqi Journal of Industrial Research,* vol. 9, pp. 1-11, 2022.