Quest Journals
Journal of Software Engineering and Simulation
Volume 10 ~ Issue 5 (2024) pp: 01-06
ISSN(Online) :2321-3795 ISSN (Print):2321-3809
www.questjournals.org

Research Paper

Thread Scheduling and Synchronization for Multi-Threaded Applications

Akshat Gupta
Computer Science and Engineering (CSE)
Lovely Professional University (LPU)
Jalandhar, India
gupta.akshat.m@gmail.com

Ansh Arora
Computer Science and Engineering (CSE)
Lovely Professional University (LPU)
Jalandhar, India
ansharora5971@gmail.com

Monika
Computer Science and Engineering (CSE)
Lovely Professional University (LPU)
Jalandhar, India
monikagandas20@gmail.com

Amandeep Kaur
Computer Science and Engineering (CSE)
Lovely Professional University (LPU)
Jalandhar, India
mannuaman38@gmail.com

Abstract**—***Multithreading is very important for modern computing in today's world, but it still faces challenges like race conditions. This research basically focuses on scheduling and synchronization mechanisms and the effect of AI on them. While Operating System is responsible for managing low-level kernel resources, the programming framework offers abstractions. At last, AI integration with scheduling and synchronization of multithreads shows an opportunity for advancement in this field but also requires careful inspection of overhead and the tradeoffs caused by it.*

R*eceived 26 Apr., 2024;* ***R****evised 03 May, 2024;* ***A****ccepted 05 May, 2024* © *The author(s) 2024.*
P*ublished with open access at* ***www.questjournals.org***

I. INTRODUCTION

Nowadays, information technology standards are established and prevail across sectors and industries, and comparatively, product life cycles shorten; they are built on the aspects of simultaneity. Both commercial people and individuals today have become so interested in highly interactive digital environments; thus, the demand for computers that can facilitate several processes into one single operation is so overwhelming. Multi-

threading, as a result, then becomes vital in modern computer science since it is the increasing case that determines such.

Computers are used to perform a multiplicity of tasks by a single person who may be working under the same productivity regime. Response time has considerable importance; even when there is a delay or a change from the external world, system adaptability for quick corrections still needs to be updated. Whether it is the possibility of managing networking requests on the server side or letting users navigate web applications easily and smoothly, multi-threading becomes a very important factor in the application experience.

On the other hand, there are many challenges to be tackled, due to which efficient multithreading cannot be guaranteed. Another significant issue is the numerous interwoven mechanisms for coordinating and synchronizing time frames because multithreaded systems need to run data integrity protection and conflict resolution concurrently.

Scheduling is paving the path for the assignment of CPU to threads in a manner that considers system load, resource availability, and thread priority. On one hand, synchronization is all about creating the means of joint access to shared resources to prevent data corruption and unauthorized actions, while on the other hand, it is about coordinating access to those resources to achieve high performance and prevent race conditions.

Many study efforts, among ongoing research, scale up the algorithms used in scheduling for more efficiency and synchronization methods. It has been identified that one of those areas that is being studied is preemptive planning, as it allocates CPU time throughout running systems in relation to thread priority and resource need.

On the other hand, using sync primitives, e.g., locks, semaphore, and mutexes, in the shared data structure helps to control the data access where these primitives can be used to control concurrency.

Hence, in this research paper, we will be going to see about a new way to enhance the way multi-threaded applications can be scheduled and synchronized for better efficiency and stability.

II. INTRODUCTION TO THREADS

A single, sequential flow of activities that are executed within a process in an operating system (OS) is called as a Thread in simple term. A running instance of a program is called as a process. Program's code, data, and its execution state are the parts of a process. While threads are like smaller units of the control unit in a process, which allows the execution of tasks simultaneously in the same memory space, Threads always share resources with their parent processes, which is why they are much lighter and quicker to create and manage.

A thread refers to a single sequential flow of activities being executed in a process; it is also known as the thread of execution or the thread of control. Now, thread execution is possible within any OS's process. Apart from that, a process can have several threads. A distinct program counter, a stack of activation records as well as control blocks are used by each thread of the same process. Thread is frequently described as a light technique.

The procedure can be easily broken down into numerous different threads. Things form opening multiple tabs in a web browser to taking input for a touch screen of a mobile all use threads to execute the user tasks.

III. IMPORTANCE OF THREADS

As processes provide strong isolation mechanisms, they are likely to be heavyweight due to the over headed creation and management of different memory spaces. Also, threads offer many advantages:

A. Better Responsiveness

The programs that take a long time to run can use threads to stay responsive. For example, while using a word processor, a thread can come into existence as the front user will be typing and the backend text formatting will be done.

B. Efficient Resource Utilization

Using threads, independent tasks can run parallel on multicore systems, which at last will surely improve the performance.

C. Simplified Code Structure

Using threads, one can improve code maintenance and organization. Also, complex programs can be reduced to smaller ones, which will look like more managed code.

D. Enhanced Concurrency

Occurrence of multiple activities or tasks at the same time within a single process to greatly increase the concurrency in the application, can be achieved by the use of threads, hence increasing the throughput and responsiveness of the system where user interaction or input-output operations are generally in high numbers.

E. Faster Context Switching

Shared memory space is the concept that is typically used by the threads to reduce the overhead requirement for context switching and also allow faster switching between threads, which in the end provides more efficient multitasking ability and also reduces switching latency between different jobs or tasks.

F. Support for Asynchronous Programming

Programming models which not synchronous in nature, hence know asynchronous programming models which typically use threads to handle input-output operation's or external event's waiting times, hence preventing the degradation of application's or system's responsiveness and concurrency efficiency.

IV. IMPORTANCE OF MULTITHREADS

A single CPU can be empowered using multithreading to perform multiple tasks or jobs at the same time, which results in better application efficiency. Multithreading can be achieved by dividing a bigger process into multiple smaller execution units, also known as threads within the parent application, as we discussed in the previous part of the paper. This group of threads is tuned to handle different aspects of the program's workload parallelly or concurrently, hence ensuring optimal CPU resource utilization.

Multithreads are very helpful for any application because they allow the program to be executed concurrently, which in turn increases its performance and responsiveness. Concurrent execution and segregation of lengthy operations into its smaller programs (or threads), which helps in maintaining the interface's smoothness is also due the multithreading.

V. PROBLEM FACED DURING MULTITHREADING

Multithreading in applications has various benefits, but there are also various conditions that might occur during multithreading due to its complexity, which may affect the way the application performs. Some of these conditions are listed below:

A. Race Condition

It is the most common condition that occurs when two or more operations or events try to access the same resources at the same time, which results in unpredictable results every time. This problem occurs due to the nature of the computer system.

B. Starvation

It is a condition in which the lower priority processes are sent to the block state for a very long period of time due to the execution and processing of higher priority processes, which results in the OS becoming unable to provide or allocate resources to the lower priority process as all of the resources needed for the lower priority are already being allocated to other high priority processes.

C. Deadlocks

A condition in which more than one process or thread is unable to proceed with its execution due to resource unavailability as it is allocated to another process or thread, which results in a state in which a group of processes becomes stuck in a way that makes them unable to make further progress.

D. Priority Inversion

Situations that occur when a low-priority task is holding a resource, such as a semaphore, due to which a higher-priority task has to wait. This resulted in the high-priority task effectively acquiring the priority of the low-priority thread; hence, the name priority inversion is used.

E. Thread Synchronization Overheading

Time spent waiting for another task or job by any task or job at any time is called synchronization overhead. For example, tasks may hit an explicit barrier during synchronization, such that all of them have finished their execution for a certain timestep before updating the status of shared data and computing the next timestep.

F. Thread Safety Issues

Some of the data structures and operations are not thread-safe, i.e., they can generate false results by multiple threads. Keeping this in mind, we should use synchronization tools and alternatives.

G. Data corruption

It is a condition which can happen when parallel access to shared data is given without proper synchronization, which can lead to incorrect program behavior.

All these conditions can be handled by various techniques, which are listed in the next section.

VI. SOUTION FOR MULTITHREADING RELATED PROBLEMS

Multithreading related problems or issues can be solved easy by implementing the techniques and algorithms of scheduling and synchronization which are being explained below:

A. Race Condition

Strategical allocation of CPU time and resources to threads to increase the system's performance and resource utilization within a multithreaded environment, which is managed by various techniques, which we collectively call as scheduling.

We can use these scheduling techniques to prevent common problems that we have discussed in the above section, like process starvation, priority inversion, and race conditions. Such problems can be solved by using pre-emptive scheduling algorithms like round-robin, shortest job next, shortest remaining time, etc. to have a better distribution of CPU time and resources, hence avoiding the entirety of the resources being used by a certain thread. This also helps higher-priority threads access resources.

Implementation of effective scheduling algorithms for multithreaded systems can result in better stability and efficiency by preventing resource collisions.

Types of scheduling used in today's world:

1) Round Robin Scheduling: Arrangement of threads in a round manner to prevent resource domination and also to increase the time and resource distribution.

2) Shortest Job Next Scheduling: In this scheduling, the thread with the shortest execution time is executed first, reducing the waiting time, which results in enhanced system throughput.

3) Shortest Remaining Time Scheduling: It is a similar algorithm to Shortest Job Next Scheduling with a small change in the CPU time allocation, in which the thread with the shortest remaining execution time will be executed first.

4) Priority Scheduling: It is a priority-based scheduling method in which the thread with the highest priority is executed first, followed by the next.

B. Synchronization

Coordinated access to the shared resources among multiple threads of a multithreaded application can be used to prevent various problems that we have already discussed in the previous section, like race conditions, deadlocks, and data corruption.

This coordinated access to shared resources can be handle easily by using various techniques or algorithms, which are collectively known as Synchronization. Algorithms like locks, mutexes and semaphores which are explained further in the paper, are types of primitive synchronization algorithms, that can be used to ensure exclusive access to the critical code section, which is a section of a program where a shared resource can be accessed.

The use of this synchronization algorithms can help to ensure data integrity, prevent conflicts and thread safety, hence ensuring multithreaded application's correctness, reliability and scalability.

Types of synchronization used in today's world:

1) Lock: Is one of the simplest synchronization mechanism that uses a lock variable to hold or lock the states of any program at any given time. This prevents the program from being modified or accessed by multiple threads that execute at the same time.

2) Mutexes (Mutual Exclusion): It is a binary variable that can provide locking by allowing only one thread to enter and execute a critical section of code at a time. Hence, the program form changes and is accessed by multiple threads. This also prevents concurrent access to shared resources.

3) Semaphores: It is a technique that uses a resource counter to regulate shared resources, which allows a certain number of threads to access the resource more efficiently.

4) Monitors: This is an abstract data type that can be used to embrace shared resources in order to make changes to them, hence providing access to multiple threads in a synchronized way.

VII. SCHEDULING AND SYNCHRONIZATION MANAGER

Operating System kernel and the Application/Runtime environment are the two, which are equally responsible for managing of scheduling and synchronization for a multithreaded application.

A. Operating System kernel

It is responsible for handling CPU resource allocation for low-level processes and scheduling of processes or threads. OS kernel uses many algorithms, such as round-robin, priority-based scheduling, and real-time scheduling to allocate CPU time and needed resources to threads or processes. It also includes system calls for processes and threads to manage synchronization primitives such as semaphore and lock.

B. Application/Runtime Environment

Applications like libraries, frameworks and programming languages provide high-level abstraction and APIs for scheduling and synchronization. It makes it easier for the developers to manage concurrency within their programs. For example, Python, C, C++ provide threading libraries for creating and managing threads with synchronized mechanisms. In Java, keywords like synchronized and classes like "java.util.concurrent" are few examples of built-in support for synchronization.

VIII. USE OF AI FOR MULTITHREADING

In today's world, artificial intelligence (AI) is spreading like wildfire. It has also become important that if AI can be dependent on a computer system's OS, then why not have our computer system's OS be dependent on AI for enhancement of multithreaded application's scheduling and synchronization? By looking at that side of AI, we have come up with some insights through our research on how the use of AI can benefit our multithreaded application scheduling and synchronization algorithms and mechanisms, which are as follows:

A. Resource Allocation Efficiency

According to our research, we found out that AI can be used to analyse the requirement of resources for a particular thread of a multithreading application more accurately by analysing various scheduling and synchronization techniques and algorithms beforehand so that it can prevent the problems that may arise in traditional methods that can lead to underutilization of resources. In some cases, this technique of using AI can reduce the idle resource count by up to 30%, hence increasing CPU utilization by 25%.

B. Workload Balancing

AI can be used to work on real-time data workload distribution, which can be more helpful than the traditional load balance techniques, which are statically configured, which lead to an unequal distribution of CPU time, which can lead to many problems like starvation and deadlocks. Hence, using AI can help the system dynamically distribute the CPU time to each thread for a multithreaded application based on the current workload distribution. In this way, there is a chance to reduce the response time of the CPU by approximately. 40% during heavy traffic.

C. Power Efficiency

Even with day to day advancements in multithreaded applications and systems, one thing that is still haunting the current system is the system's power consumption. Even when the workload on the system is less, around 50% watt power is wasted on average due to issues with the scheduling and synchronizations of the threads, which lead to unwanted starvation and deadlocks, which lead to more power consumption. So by using AI, we can efficiently manage the scheduling and synchronizations of the threads to reduce the unwanted problems. This method of using AI for this can reduce the power consumption by approximately. by 25%.

IX. CONCLUSION

In the era of modern computing, multithreading is an inseparable part for achieving higher responsiveness and efficiency. But it still faces complex problems such as race conditions and deadlocks, which is affecting its performance, due to which the necessity of scheduling and synchronization techniques is increasing at a very fast pace. Multithread scheduling, which always gives fair resource allocation can be ensured using various effective algorithms, while data corruption can be prevented by using various synchronization mechanisms.

The responsibility for managing the multithreading is shared between both the operating system kernel and the programming framework, where low-level resources are handled by the kernel and high-level abstract resources are managed by the framework.

AI-driven methodologies always show dynamic adjustments to the workload changes and the system environment, while at the same time optimizing resource utilization across various landscapes. However, consideration of facts like overhead and trade-offs is essential for successful implementation is also needed.

ACKNOWLEDGMENTS

We are very grateful to our supervisor, Amandeep Kaur ma'am for her invaluable insights, motivation and guidance throughout this research.

We would also like to show our gratitude to our research team for their input in discussion and research and also for their invaluable ideas and insights.

We are very grateful to all of the peoples and organization for their work in this field of technological study.

A heartfelt thank you to our university, "Lovely Professional University, Punjab" for their support in our research, as we wouldn't be able to complete our research without them backing us through the process.

Lastly, we would like to thank our family, friends and colleagues for their motivation and support.

REFERENCES

[1]. F.J. Cazorla, P.M.W. Knijnenburg, R. Sakellariou,E. Fernandez, A. Ramirez, and M. Valero, "Predictableperformance in SMT processors: synergy between theOS and SMTs," Computers, IEEE Transactions on, vol 55, no. 7, pp. 785 – 799, july 2006. F.N. Sibai, "Performance effect of localized thread

[2]. schedules in heterogeneous multi-core processors," in Innovations in Information Technology, 2007. IIT '07.4th International Conference on, nov. 2007, pp. 292 –296.

[3]. O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers," Micro, IEEE, vol. 29, no. 1, pp.22 –32, jan. feb. 2009.

[4]. Holland, D., Lim, A., and Seltzer, M., A New Instructional Operating System, ACM SIGCSE Bulletin, 34, 1 (Mar. 2002), 146-148.

[5]. Hovemeyer, D., Hollingsworth, J., and Bhattacharjee, B., Running on the Bare Metal with GeekOS, ACM SIGCSE Bulletin, 36, 1 (Mar. 2004), 315-319.

[6]. Atkin, B., and Sirer, E. G., PortOS: An Educational Operating System for the Post-PC Environment, ACM SIGCSE Bulletin, 34, 1 (Mar. 2002), 116-120.

[7]. S. Midkiff and D. Padua. Compiler algorithms for synchronization. IEEE Transactions on Computers, 36(12):1485–1495, Dec. 1987

[8]. A. Birrell. Systems Programming with Modula-3, chapter An Introduction to Programming with Threads. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[9]. P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. Journal of Parallel and Distributed Computing, 49(2):2218–244, Mar. 1998. 32.

[10]. A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, May 1991.