MULTITHREADING JOB SCHEDULING

1st A.MANEESH REDDY
department of AIML
SRM Institute of Science and Technology
tiruchirapalli, Tamilnadu, India
aramatimaneeshreddyr@gamil.com

2nd K.KEVIN REDDY
department of AIML
SRM Institute of Science and Technology
tiruchirapalli, Tamilnadu, India
b13441678@gmail.com

***Abstract*—In this paper we introduce the design, development and evaluation of a simulated work scheduler implemented with threads and a priority queue for managing task execution in multithreaded environments. This Scheduler features Priority-Queue-based preemptive scheduling combined with time-slicing (Round-robin within same priority) and overhead modelling of context switching. The simulator generates a detailed event log and calculates standard scheduling metrics - turn-around time, wait time, response time, CPU utilization. Experiments present responsiveness versus throughput trade-offs as quantum and context-swi- long overheadchange this balance. Herein the system may be used to illustrate scheduling concepts and—and to implement advanced scheduling policies.**

Kmacros: Job scheduling, priority queue, preemptrve scheduling, time slicing, context switch, multithreading, simulatorcontext-swi-long overheadchange this balance. Herein the system may be used to illustrate scheduling concepts and—and to implement advanced scheduling policies.

Kmacros: Job scheduling, priority queue, preemptrve scheduling, time slicing, context switch, multithreading, simulator

I. INTRODUCTION

TTask scheduling is an important function of operating systems because it decides how tasks use the CPU. When many tasks run together, the system must manage them in an efficient and fair way. Sequential execution is simple but slow, as tasks run one after another. Modern systems use multithreading to allow several tasks to run at the same time and improve performance.

This project demonstrates the difference between sequential and multithreaded execution using a simple scheduler. A graphical interface shows each task ' s progress and performance details like execution time and CPU usage. By comparing both modes, we can clearly see how multithreading increases speed and makes better use of system resources

II. LITERATURE REVIEW

Task scheduling is one of the key aspects in operating system design, because it directly affects system performance, CPU utilization and responsiveness. As computing environments became more sophisticated (and, notably, with the advent of multi-core processors and parallel systems), researchers turned their attention to how best manage multiple tasks or threads being executed simultaneously. A good scheduler is supposed to be fair, minimizes the waste of waiting time and keeps threads well synchronized when accessing the shared resource.The concept of semaphores and monitors was first formalized in Edsger Dijkstra's study on concurrent programming (1968) and C.A.R. Hoare (1974), respectively, which later forms the foundation for mechanism used to synchronize processes in modern systems. They prevented race conditions, deadlocks in multi-threaded environments. As higher-level programming models became available, these approaches were implemented as thread-sync-chronization primitives such as the mutex locks and condition variables found in contemporary multithreading libraries (i.e., POSIX threads).

Earlier times had simple algorithms like FCFS and SIF in the operating system. These algorithms were simple to implement, but were ineffective in timesharing systems as they could result in bad response times and process life starvation. To overcome this Round Robin scheduling was developed, which allowed all the processes to have a time slice and rotated them in circular order. By this method, equity was improved, but it failed to take into account processor urgency.This method resulted in a better fairness but process priority was not taken into account. And so, burst scheduling was introduced and allowed urgent jobs to run ahead of not-so-important ones. But it produced the starvation issues for low priority processes, which brought the concept of aging,i.e. over time waiting tasks gets preference.

With the emergence of multithreading and parallel computing, the focus of scheduling researchshifted to addressing concurrent threads within a single process. Two threads share the same memory and CPU's synchronization was now the issue. There began to be research into ways to minimise context switching and save CPU use without generating inconsistent data. Preemptive scheduling was first introduced to with allow the system to interrupt a currently running thread in order to start a rad higher priority one and leading grad.

There also has been work Michael2000 hybrid scheduling algorithms that use both a time slice based algorithm and one in which the priorities are dynamically adjusted. These techniques are capable of adapting to varying system workloads and do not degrade performance statically. The various scheduling algorithms used in the more recent operating systems, all rely on one or other form of combination of these two to achieve a balance between fairness and efficiency. Also, load balancing methods are now receiving increasing attention to achieve balanced distributions of workloads among many cores/processors for high overall throughput in parallel systems.

A. Ordinary Spatial Cloaking

Waves and the host material in which they propagate have a symbiotic relationship: both act on each other. A simple spatial cloak relies on fine tuning the properties of the propagation medium in order to direct the flow smoothly around an object, like water flowing past a rock in a stream, but without reflection, or without creating turbulence. Another analogy is that of a flow of cars passing a symmetrical traffic island - the cars are temporarily diverted, but can later reassemble themselves into a smooth flow that holds no information about whether the traffic island was small or large, or whether flowers or a large advertising billboard might have been planted on it.

Although both analogies given above have an implied direction (that of the water flow, or of the road orientation), cloaks are often designed so as to be isotropic, i.e. to work equally well for all orientations. However, they do not need to be so general, and might only work in two dimensions, as in the original electromagnetic demonstration, or only from one side, as for the so-called carpet cloak.

Spatial cloaks have other characteristics: whatever they contain can (in principle) be kept invisible forever, since an object inside the cloak may simply remain there. Signals emitted by the objects inside the cloak that are not absorbed can likewise be trapped forever by its internal structure. If a spatial cloak could be turned off and on again at will, the objects

FIG 2: (a) Schematic figure of London (UK) in 2010, and published in the Journal of Optics. An experimental demonstration of the basic concept using nonlinear optical technology has been presented in a preprint on the Cornell physics arXiv. This uses time lenses to slow down and speed up the light, and thereby improves on the original proposal from McCall which instead relied on the nonlinear refractive index of optical fibres. The experiment claims a cloaked time interval of about 10 picoseconds, but that extension into the nanosecond and microsecond regimes should be possible.

Studies also focus on evaluation of scheduler performance and use measures such as turnaround time, waiting time, response time and CPU utilization. Such measurements contribute to the assessment of how a scheduler works under workloads and conditions variety. The primary objective of scheduling studies is to achieve efficient system resource utilization and at the same time provide responsive behavior for users and stable response times for jobs.

The Multithreaded Task Scheduler assignment builds upon these research ideas, and uses them to replicate in simplified form how conventional operating systems perform task scheduling. It is a preemptively priorities based multitasking including synchronization and time slicing. This makes it possible to observe how the various jobs compete for CPU

III.

SYSTEM AND IMPLEMENTATION

A. system architecture

Task priorities are sorted in the Scheduler Core. Tasks marked as high receive enhanced priority in the queue, executing before low-priority tasks. The scheduler utilize Priority Scheduling with Time Slicing to waiting threads, balancing justice among the waiters preserving system responsiveness for the high priority tasks. Thread manager initiates and handles threads. Every task is run in a new thread with Java Runnable and Thread. I mean thread safe and serialized to ensure tasks don't interfere.

The Thread Manager (Concurrency Handler) will handle to spawn and maintain threads. Each task is running in a separate thread by utilizing Java's Runnable abstraction and Thread. Thread safety and synchronization are implemented to avoid the collisions between tasks simulated context switching delay models actual CPU handling in a multitasking mode. Execution Monitor The status of each launched, running and completed task is monitored by the Execution Monitor as well as statistics one execution time taken, waiting times and CPU used(span).

B. system implementation

The system was developed in Java due to its robust support for multithreading and object-oriented design. Tasks in the sequential set signifies that the tasks will be performing one after another, and having a high waiting time, with minimal CPU utilization. However, the multithreaded version executes tasks asynchronously. Threads are almost started at the same time and each threads simulates running with sleep method as that amount of processing time The join() method is responsible for synchronizing across all threads, so that every thread has finished and a performance summary is produced as the final output. We aim at building a model of scheduling that is true to the operating system's operation, where we demonstrate how running real-time applications as multi-threaded processes increase computer utilization and responsiveness, overall system performance and finally contribute to effective, safe and predictable task execution on resource-constrained platforms.

IV. RESULTS AND IMPACT ANALYSIS

A. Objective of the Analysis

Result and Impact Analysis have to show us whether the Willow scheduler is able to achieve better performance in multithreading model versus single thread execution. The experiment has been particularly considered making use of metrics, including execution time, CPU utilization responsiveness and fairness among tasks, which gives perception about the efficiency of scheduling.

B. Experimental Setup

Also administered were two versions of the program:

Sequential Scheduler : Tasks that are processed one at a time using a single thread.

Multithreaded Scheduler: Tasks that are run concurrently with multiple threads.

Task ID	Priority	Burst Time (ms)
T1	3	2000
T2	1	4000
T3	4	1500
T4	2	3000
T5	5	1000

Fig. 1. cpu utilisation

4. Sample Output

(a) Sequential Execution Output

```yaml
Starting Sequential Execution...
T1 started
T1 completed
T2 started
T2 completed
T3 started
T3 completed
T4 started
T4 completed
T5 started
T5 completed
Total Execution Time: 11500 ms
Average Waiting Time: 4600 ms
CPU Utilization: 52%
```

Fig. 2. Caption

5 sample tasks were tested in each version with diverse priorities and burst times. The system was executed on a dual-coreprocessor for acting as there al-time parallelism emulator.

C.

Execution Time Comparison, Average Waiting Time Comparison, and CPU Utilization Comparison. The charts clearly show that the multithreaded version significantly reduces total runtime and waiting time. Sequential execution forced tasks to wait for others to finish, causing CPU idle periods. Multithreading allowed simultaneous execution of independent tasks, increasing CPU activity and responsiveness. The performance improvement becomes more visible as the number of tasks increases

(b) Multithreaded Execution Output

```yaml
Starting Multithreaded Execution...
T1 started
T2 started
T3 started
T4 started
T5 started
T3 completed
T5 completed
T1 completed
T4 completed
T2 completed
Total Execution Time: 4300 ms
Average Waiting Time: 1200 ms
CPU Utilization: 88%
```

Fig. 3. Caption

Parameter	Impact of Multirhreading
Speed	Tasks complete faster due conccruent execution
Responviisess	All tasks begin almost imedatiely
CPU Efficiency	High utiization with fewer idle gaps
Fairness	Balanced scheduling with time s
Scalability	Can handle large worklods woıklads efficiently

Fig. 4. caption

SEQUENTIAL	MULTITHREADED	DIFFERENCE
FINAL EXECUTION TIME:	11500 MS	62.6% FASTER
AVERAGE WAIT:	4600 MS 1200 MS	73.9% DOWN
CPU UTILIZATION:	52% 88%	

Fig. 5. Caption

D.

The results support the multithreaded technique. This is a well-paced utilization of processing to get staff off the line and reduces wait time more quickly. This results in a more efficient and dynamic system overall, particularly when performing multiple tasks at the same time or doing PC tasks with powerful hardware.

V. Challenges and Future Direction

Problems Encountered During the development of Multithreaded Job Scheduler, several problems were faced. A thread synchronization was a foremost problem as multiple threads working on a common resource resulted in race conditions and incongruous results. It was vital to provide adequate use of synchronization techniques such as locks and join() to maintain the consistency in the data.

Another problem was load balancing: some of the high-priority threads would finish while others got starved; so fairness was decreased. Content switch delays were also a difficult matter to handle programs were more difficult to debug than sequential ones, because of unpredictable order of thread execution. For future upgrade, the project can be improved by adding dynamic load balancing which sends the

whole or parts of tasks to another thread based on real CPU burden. One possible path is to develop adaptive scheduling algorithms which dynamically change priorities and time-slices according to the state of the system. It would also be possible to generalize the scheduler for distributed systems.

VI. CONCLUSION

The multithreaded job scheduler is a neat example of how modern operating systems can nimbly handle multiple tasks concurrently using techniques such as threading, assigning priorities, and synchronization. The project, by virtue of implementing both sequential and multithreaded versions, makes it very clear how parallelism brings in these huge benefits of drastically cutting down the execution time, efficient use of the CPU, and better system responsiveness. Priority queues and time slicing are the mechanisms through which the scheduler guarantees that no task is neglected and, at the same time, the system operates at maximum efficiency. It is absolutely clear from the comparison that multithreading drastically reduces waiting time and at the same time optimizes resource usage, hence, this is the best approach for systems that are designed to handle multiple jobs at the same time. the project serves as a valuable resource for understanding the issues surrounding thread creation, synchronization, and context switching, which, in turn, are very essential in operating system design. In general, the research demonstrates that an efficiently designed scheduler not only leads to up performance levels but also long-term computing stability and capacity get enhance techniques such as threading, assigning priorities, and synchronization. The project, by virtue of implementing both s versions, makes it very clear how parallelism brings in these huge benefits of Priority queues and time slicing are the mechanisms through which the scheduler guarantees that no task is neglected and, at the same time, the system operates at maximum efficiency. It is absolutely clear from the comparison that multithreading drastically reduces waiting time and at the same time optimizes resource usage, hence, this is the best approach for systems that are designed to handle multiple jobs at the same time. Additionally, the project serves as a valuable resource for understanding the issues surrounding thread creation, synchronization, and context switching, which, in turn, are very essential in operating system design.

VII. REFERENCE

A., Galvin, P. B., and Gagne, G. (2018). Operating System Concepts (10th ed.). John Wiley and Sons. Stallings, W. (2018). Operating Systems: Internals and Design Principles (9th ed.). Pearson Education. Dijkstra, E. W. (1968). Cooperating Sequential Processes. Technological University Eindhoven. Hoare, C. A. R. (1974). Monitors: An Operating System Structuring Concept. Communications of the ACM, 17(10), 549–557.Tanenbaum, A. S., and Bos, H. (2015). Modern Operating Systems (4th ed.). Pearson. Oracle. (2023). Java Platform, Standard Edition Documentation.URL: https://docs.oracle.com/javase/8/docs/api/

William, J., and Andrew, T. (2020). Multithreading and Concurrency in Java: Concepts and Best Practices. Journal of Computer Applications, 45(2), 89–96. IEEE Computer Society. (2021). Proceedings of the IEEE Symposium on Parallel and Distributed Processing. IEEE Press.Rahman, M., and Sharma, D. (2019). Comparative Analysis of CPU Scheduling Algorithms for Multi-Core Processors. International Journal of Computer Science and Engineering, 7(5), 112–118. GeeksforGeeks. (2023). Thread Synchronization in Java.URL: https://www.geeksforgeeks.org/thread-synchronization-java/