Multithreading on Different Operating Systems

Fabricio Junqueira, Eric Shen, Ari Torczon

Abstract—**A foundational concept in modern operating systems is multithreading, it allows software to perform multiple tasks at the same time by dividing the workload across multiple CPU cores. It is essential to efficiently allocate resources, have a responsive user interface(s), and handle complex simultaneous operations. Every operating system implements multithreading differently, using its fundamental architecture to optimize performance and experience for developers. This paper analyzes multithreading usage in Linux, Windows, and macOS, highlighting their respective thread management techniques, scheduling algorithms, APIs, and use cases. The macOS portion, presents frameworks like Grand Central Dispatch, NSTread, and POSIX Threads, focusing on their purpose in balancing control, simplicity, and efficiency. The Windows section covers the powerful, but complex WindowsAPI, showing how Windows thread management makes the most of its custom API to give the best user experience in application responsiveness and performance. The Linux portion will present an overview of the lightweight, efficient implementation of POSIX Threads, how threads are integrated with the kernel using tasks, and the Completely Fair Scheduler (CFS), which ensures fair CPU distribution and optimized performance for high-demand applications. Through this analysis, the paper identifies challenges and strengths or multithreading on all platforms, giving insights for better practices for developers.**

I. Introduction

MacOS multithreading is built on the Darwin Kernel, which is a hybrid of BSD and Mach components. This allows macOS to provide powerful, friendly tools for simultaneous programming. It utilizes preemptive multitasking, guaranteeing that the kernel dynamically schedules threads based on priorities and system date. This is achieved through three different APIs: Grand Central Dispatch (GCD), NSThread, and POSIX Threads. All of these have an individual yet unique level of abstraction and control, opening possibilities to developers to freely choose the best fit for their application's requirements.

II. MACOS MULTITHREADING

We start by analyzing macOS design with multithreading support through three essential frameworks: grand Central Dispatch (GCD), NSThread, and POSIX Threads (pthreads). These frameworks provide multiple levels of abstraction, allowing developers to have flexibility on the requirements and complexity of their applications.

Grand Central Dispatch (GDC) is a high-level concurrency framework that Apple introduced with the purpose of simplifying multithreading. As an alternative to direct thread management, developers would define tasks that are executed on dispatch queues. This method supports two different types of queues: serial queues, where the execution takes place sequentially, and concurrent queues, allowing tasks to run simultaneously, giving CPU a way to optimize usage. One of the features of GCD is the Quality of Service (QoS) classes, which allows tasks to be executed based on their priority of importance. Although this framework provides efficiency and ease of use, the thread-level control on GCD is majorly abstracted, creating limitations for sustainability on highly-complex or real-time applications. NSThread lies on the mid-level approach, giving developers direct control over the initialization of threads and lifecycle management. In comparison with the previous framework, NSThread provides the option to start, pause, or stop threads. This is useful for precise management of thread behavior on applications. And although it provides more flexibility than GCD, developers still need to be careful with synchronization to avoid race conditions or deadlocks. Finally, we have POSIX Threads (pthreads) on the lowest-level API available on macOS, providing full control on thread behavior and synchronization. This framework is highly portable across different platforms, making it ideal for applications that are performance critical with required fine-grained control, just like a UNIX-compliant library. Everything is manually manipulated, from thread creation to managing their lifecycle explicitly. Because the level of control is powerful, it poses significant complexity, leading to race conditions, deadlocks, or priority inversions if there is any sort of thread mismanagement. This framework is recommended to be used on specialized or high-performance applications. MacOS manages thread scheduling with the Darwin Kernel utilizing preemptive multitasking, meaning that it dynamically allocates CPU time to threads based on their priority, having critical tasks such as UI interactions prioritized over background tasks to ensure responsiveness for instance. Additionally, preemption is implemented to interrupt low priority tasks when higher-priority ones need execution, giving maintenance system performance under various workloads.

All in all, macOS offers a varied range of frameworks with different advantages. GCD being simple and efficient by abstracting thread management while still offering resource use optimization. NSThread gives explicit thread control, increasing complexity but making it ideal for applications requiring manual management. And finally, POSIX Threads, offering maximum control and performance to developers with the cost of complexity and error-prone environment; suitable for critical systems. Overall, macOS permits flexibility for developers to choose the framework that best suits their requirements and needs.

III. WINDOWS MULTITHREADING

A. Windows API

Windows is unique in that it has its own API called Win32 API, offering powerful yet complex capabilities, which allows for a more manual management of threads. Essentially, Windows multithreading caters towards application responsiveness and system integration because of its robust and versatile framework; however, this comes at the cost of resource overhead and extensive manual complexity. Looking at the Windows multithreading architecture, its thorough framework integrates thread creation, scheduling, and synchronization directly in the operating system itself. This allows for developers to have powerful tools that are comprehensively detailed, putting the power of managing concurrent tasks in their own hands. The Windows API, also known as Win32 API, is responsible for the creation and management of threads. Now the reason why this is so powerful yet requires a lot of manual work is because of the code editor Microsoft Visual Studio, more on this later. To break down Windows multithreading, it begins with the creation of threads with the C-based function "createThread." This function contains many parameters such as lpThreadId, dwStackSize, and lpThreadAttributes. Having so many parameters is why Windows thread management is a lot more complex and detailed than other systems. This also means Windows threads have a larger overhead, and this is not including the stack size. Even though threads share the resources and memory of their parent processes, each thread has its own stack size. These threads can have manually set stack sizes (default is typically 1 mb), whereas other operating systems have dynamically allocated stack sizes (starting with 8 kb). In any case, all of these parameters for Windows threads make up the metadata in threads, which is stored in the Thread Environment Block (TEB). Diving deeper into the Windows API, developers do have leadway in automation of thread management, which ironically, is manually incorporating the Windows Thread Pool API. This API can automatically manage a "pool" of reusable threads, which involves dynamically adjusting their numbers depending on the system load and requirements of different applications. This abstraction of manual thread creation simplifies development and lifecycle management.

B. Scheduling and Synchronization

Now as far as scheduling for Windows threads, Windows uses a priority-based preemptive scheduler. This works by having each thread assigned one of the 32 priority levels, including real-time and variable classes, where the higher one always preempts the lower. This ensures that essential tasks are completed quickly and that lower-priority requests wait until resources free up. As for the kernel, it provides context switching, where the CPU switches between threads, which allows it to maintain responsiveness; However, context switching does cost in performance whenever the kernel needs to save and restore thread state. This is why developers need to carefully set thread priorities to avoid the case of priority inversion, which is where higher-priority threads are blocked by lower-priority threads that are holding the resources. Now for to face synchronization challenges, Windows uses the following primitives:

1. Critical sections for threads within the same process
2. Mutexes to allow access to the resources for one thread at a time
3. Semaphores to have permissions for multiple threads when competing for limited resources
4. Event objects which indicate state changes for threads, which signal when operations are complete

As with any challenge of coming up with solutions to synchronization problems, these primitives are flexible yet must be utilized carefully to prevent deadlocks and race conditions while still being able to maximize concurrency in multithreading.

C. Advantages

Multithreading in Windows is a very flexible technology to work with when developing applications that require high performance, responsiveness, and deep integration with the operating system. From ensuring optimal user experience using end to end tooling all the way down to strong system level support, the threading model under Windows has its unique strengths. One of the most impressive features of multithreading in Windows is the extensive development tool set provided by Microsoft. Tools such as Visual Studio, as mentioned earlier, Concurrency Visualizer, and WinDbg allow developers to create, debug, and optimize multithreaded applications easily. The debugging capabilities of Visual Studio show runtime views of thread states, resource contention, and performance bottlenecks in an application. Concurrency Visualizer provides a visual of thread execution to help developers quickly identify inefficiencies like excessive context switching or synchronization delays. Now, in terms of operating system integration, Windows threads are closely integrated, which makes them very suitable for applications that require responsiveness from GUI and asynchronous operations. Take GUI's for example, threads under Windows can work well with the message loop system to perform background processing without hanging the interface. This is great in applications such as Microsoft Word and multimedia editors. Another example is asynchronous I/O operations, where Windows, by default, has the feature of IOCP (I/O completion ports) that efficiently supports handling thousands and thousands of asynchronous I/O operations. It marks a special difference for making Windows threads an excellent choice in server-side applications such as web servers and database systems. Additionally, thread pools under Windows ease the job of management by themselves, dynamically adjusting the number of active threads according to workload. This dynamic optimization would bring on better resource utilization without extra manual effort. In terms of flexibility and scalability, Windows threading itself is designed to scale well on modern hardware architectures, which include systems containing multiple CPU's, hyperthreading capabilities, and Non-Uniform Memory Access. The Windows threading model provides fine control with thread affinity, thus supporting the ability to bind threads to processors or cores for optimal execution of applications that are computationally expensive. For more advanced scenarios, Windows also supports fiber threads, which are user-managed execution contexts. These allow developers to build lightweight, cooperative multitasking systems within a single thread. This kind of flexibility is quite useful in situations such as game engines, where control over the execution order and timing is crucial to the user experience. Overall, by utilizing the thread managing abilities of Windows, it can work with bigger workloads while being responsive in nature. Whether it is live data processing in financial applications or rendering in a Game Engine; Windows multithreading thus lets applications balance all the computational tasks along with various I/O operations.

D. Disadvantages

While Windows multithreading offers some very powerful tools and integration, it does have some major complications with regards to development and performance because of resource intensive design, manual management, and portability issues. Threads in Windows are just heavier by nature compared to their equivalents under Linux and macOS. By default, each thread is given a 1 MB stack, whereas on Linux, the default stack size is 8 KB with pthreads. Windows threads also keep a lot of kernel-level metadata, such as Thread Environment Blocks (TEBs) and security descriptors. This makes them more memory intensive, especially for applications that use a large number of threads. Context switching is a common operation in multithreaded applications and also adds to the resource overhead. Given that Windows utilizes a priority-based preemptive scheduler, the kernel needs to save and restore the thread state at every switch, which introduces latency and makes the CPU less efficient. In terms of manual complexity, multithreading in Windows requires an awful lot of manual management by a developer, thus increasing the chances of errors. The main sources of complexity come from things like thread lifecycle management, where threads must be created and destroyed explicitly by the programmer. Poor thread termination can give rise to resource leaks. Also, from critical sections to mutex and semaphores, every possible flavor of synchronization tool is available under Windows; however each one could be dangerous if used improperly, which leads to race conditions, deadlock, and other problems. Now, for priority tuning, scheduling based on priority in Windows requires that developers are solely responsible for explicitly setting thread priorities. Inadequate priority can result in priority inversion, which is where higher-priority threads become blocked waiting for lower-priority threads that are holding a resource. These manual requirements not only increase development time but also demand a deeper understanding of system level threading concepts, making Windows less beginner friendly compared to higher level abstractions like macOS's Grand Central Dispatch (GCD). The portability challenges that Windows faces also should be mentioned, where the threading APIs of Windows, such as CreateThread and WaitForSingleObject, are proprietary and not portable across platforms. Applications written to exploit a specific threading functionality of Windows will more often than not require substantial reworking before they can run on Linux or macOS. On the other hand, Linux's POSIX-compliant threading libraries are implemented and supported on macOS, which makes them cross platform and more portable. This lack of standardization makes Windows multithreading less appealing for developers building cross-platform applications, especially when portability is a high priority. For debugging and performance optimization, even with good tools, debugging and performance optimization of multithreaded applications under Windows can be cumbersome. For example, finding and fixing deadlocks or race conditions may involve detailed analysis, which can take a lot of time. Powerful profiling tools like Visual Studio's Concurrency Visualizer can sometimes overwhelm a developer with the volume of thread-level data, making diagnosis of specific performance issues harder to come up with. Looking at scalability, because threads in Windows have been designed to be scalable, very high numbers of threads have proportionally low marginal benefit, considering the rather high cost for each thread. This is a situation where alternative models, such as asynchronous programming or lightweight cooperative multitasking, like fibers, would probably be much more efficient. However, they also introduce additional complexity to be handled by the developers.

IV. LINUX MULTITHREADING

Linux multithreading revolves around POSIX-compliant threads, commonly known as Pthreads, which provide developers with a standardized and portable API for managing concurrency. P-threads are lightweight and efficient, making them ideal for applications that require high performance without the overhead of heavier thread models like those in Windows. Unlike Windows threads, Pthreads offer dynamic stack allocation by default, starting at 8 KB, which reduces memory usage and is very useful for applications that use many threads. Developers working with P-threads can use functions such as pthread_create to spawn threads and pthread_join to wait for thread completion. Synchronization mechanisms, including mutexes, condition variables, and semaphores, offer a balance of simplicity and control, allowing developers to handle complex multithreaded workloads effectively.

Linux uses the "Completely Fair Scheduler (CFS)" for thread scheduling, ensuring fairness by allocating CPU time based on thread priority and usage history. Threads can be assigned real-time or non-real-time priorities, giving developers the flexibility to optimize performance according to their needs. Synchronization in Linux relies on robust but also streamlined primitives such as mutexes for mutual exclusion, semaphores for managing access to shared resources, spinlocks for minimizing context-switching overhead in short-duration locks, and condition variables that enable threads to wait for specific conditions to be met.

Linux multithreading offers several advantages that make it a powerful tool for developers. Its lightweight design with dynamically allocated stack sizes ensures efficiency, particularly for applications requiring many threads. Portability is another strength; POSIX compliance enables threading libraries to work seamlessly across platforms, including macOS, with little or no modification. The threading model emphasizes simplicity, which provides an accessible API that developers can easily learn and implement into their own work. Scalability is another advantage, as Linux threads perform very well on multi-core processors and systems with hyperthreading. The kernel offers fine-grained control over CPU affinity, allowing developers to bind threads to specific cores, optimizing performance for computationally intensive tasks. Linux also provides a wide variety of tools for debugging and profiling, such as gdb, perf, and valgrind, which deliver detailed insights into thread behavior, performance bottlenecks, and synchronization issues. Additionally, the open-source nature of Linux gives developers the ability to explore and customize the kernel's threading and scheduling mechanisms, further improving their applications.

Despite these advantages, Linux multithreading still comes with some challenges. The learning curve can be steep, as developers have to grasp concurrency concepts in order to handle race conditions and deadlocks effectively. Manual resource management requires developers to manage thread lifecycles and resource deallocation, increasing the risk of resource leaks if not handled correctly. Unlike macOS's Grand Central Dispatch or Windows Thread Pools, Linux lacks built-in high-level abstractions for thread management, which means developers must either implement their own thread pooling mechanisms or rely on third-party libraries. Additionally, while Linux threads are lightweight, extremely high thread counts can still lead to context-switching overhead and competition for shared resources. Ensuring consistent performance for real-time applications can be difficult when under heavy system load in a general-purpose operating system. For applications requiring scalability, the traditional threading model may run into issues, requiring the use of asynchronous programming models like libuv or frameworks such as Boost.Asio.

Overall, Linux multithreading provides a powerful, efficient, and portable solution for developers creating high-performance applications. With its lightweight design, scalability, and POSIX compliance, it is particularly well-suited for cross-platform development and applications with demanding concurrency requirements. However, it requires careful management of resources and a solid understanding of concurrency principles to maximize its potential. By using all the available tools and best practices, developers can have very good results in performance and scalability, whether building web servers, scientific computing applications, or other complex systems.

V. MULTITHREADING ALGORITHMS

A. Methodology

In order to directly test the operating systems against each other fairly, there were various different factors we had to consider. Uniformity, as in, if we are comparing the operating systems directly to each other, having each operating system test the algorithms under the same conditions. This would involve running them in a virtual environment and allocating the same amount of resources to each operating system. Then we also had to consider a universal language/code editor that would run the program on each operating system without any compatibility issues. Although we covered many different advantages and disadvantages of each operating system throughout this report, it was best not to include them when running these tests, as that would result in having an extreme amount of results to process, which is why the previous considerations were made solely for the sake of this test.

B. Implementation

The multithreading algorithm we chose to test on each operating system was based on the pseudocode for the Fibonacci Sequence from Jesus Gonzalez, where "spawn" and "sync" indicate concurrency.

```
P-FIB (n)
    if n ≤ 1
        return n
    else x = spawn P-FIB (n - 1)
        y = P-FIB (n-2)
        sync
        return x + y
```

Now, the reason why this is considered a multithreading algorithm is because each input number can be assigned to a thread which can be called in any order, meaning they can run parallel to each other as the computation of one doesn't affect the other (Gonzalez 2019). In our case and in general, the reason why we choose this as a multithreading algorithm to use is because for very large numbers of n, the output grows exponentially, making this a computationally expensive algorithm. Our Python implementation included having a thread for each number compute the fibonacci sequence result for each number 30 through 33 (4 threads total).

C. Results

Using Python libraries, we were able to compute the execution time, CPU usage, and memory usage. The results can be split into the following:

s = seconds, mb, = megabytes

Windows:
- Test 1:
  - Execution Time: 1.51 seconds
  - CPU usage: 12.1%
  - Memory Usage: 14.87 mb
- Test 2:
  - Execution Time: 1.58 seconds
  - CPU usage: 13.6%
  - Memory Usage: 14.89 mb
- Test 3:
  - Execution Time: 1.54 seconds
  - CPU usage: 12.5%
  - Memory Usage: 14.89 mb

MacOS:
- Test 1:
  - Execution Time: 1.26 seconds
  - CPU usage: 22.9%
  - Memory Usage: 14.05 mb
- Test 2:
  - Execution Time: 1.21 seconds
  - CPU usage: 20.5%
  - Memory Usage: 12.67 mb
- Test 3:
  - Execution Time: 1.20 seconds
  - CPU usage: 19.7%
  - Memory Usage: 12.12 mb

Linux:
- Test 1:
  - Execution Time: 1.08 seconds
  - CPU usage: 51.5%
  - Memory Usage: 12.88 mb
- Test 2:
  - Execution Time: 1.07 seconds
  - CPU usage: 45.6%
  - Memory Usage: 12.88 mb
- Test 3:
  - Execution Time: 1.06 seconds
  - CPU usage: 46.5%
  - Memory Usage: 13.00 mb

Analyzing the execution times, Linux seemed to be the fastest, followed by macOS, and then Windows. This might indicate that Linux and macOS handle multithreading more efficiently than Windows. This could also be due to the differences in OS kernel design, thread scheduling, and Python's threading behavior on each platform. For CPU usage, Linux and macOS show higher CPU usage, suggesting that they have better thread utilization and less idle time. Windows has the lowest CPU usage, which indicates suboptimal parallelism or more thread overhead. For memory usage, MacOS and Linux use less memory than Windows, which highlights their efficient thread management systems. The higher memory usage in Windows could come from having more overhead in its threading library or possibly context switching.

VI. Conclusion

Multithreading is essential for modern operating systems, enabling efficient resource and hardware utilization that will enhance application performance, with each operating system implementing their own frameworks and design philosophies tailored to their architectures. MacOS utilizes a combination for Grand Central Dispatch (GCD) with a high-level abstraction that simplifies task management, NSThread offering control for performance critical use cases, and POSIX Threads allowing full control of complex use cases, allowing developers to use a versatile toolkit to execute concurrency. Windows has a very powerful and flexible multithreading model deeply embedded in its operating system. It also provides developers with a wide range of tools and APIs. Its priority-based, preemptive scheduler, extensive synchronization primitives, and thread pooling enable some powerful ways to control concurrency and ensure responsiveness for applications. These features come at the cost of higher resource overhead and manual thread management, which introduces complexity and potential for errors. Linux offers a model of multithreading, which puts great emphasis on lightweight processes, efficiency, and portability. Its pthreads are highly flexible, standardized, and easy to use while creating or managing threads, which makes Linux very attractive for cross-platform development. Contrary to Windows, threads in Linux have a much more uniform relationship with processes because they are mostly referred to as "tasks" within the kernel, which is very efficiently handled in terms of resource sharing and management. However, Linux does not provide that high-level abstraction as in macOS, nor the strong tooling Windows provides. That means, a developer will more often have to deal with all concurrency issues manually, which may require knowledge of both the threading model and the kernel. However, simplicity and the desire for efficiency make it anyhow a very powerful platform while developing multithreaded applications, especially when performance and resource optimization is required. Overall, Linux is the best for multithreading performance in terms of having the fastest execution, highest CPU utilization, and low memory usage. This reflects Linux's strength in managing threads for high-performance computing tasks. Now, MacOS has a good balance of speed and resource usage in terms of efficient execution and thread scheduling. It may benefit from macOS's Grand Central Dispatch system, which optimizes concurrency. On the other hand, Windows is slower and less efficient compared to macOS and Linux. Lower CPU usage suggests Python's multithreading may not work as effectively on Windows, possibly due to differences in the OS's thread scheduler or higher thread management overhead.

REFERENCES

[1] Apple Developer Documentation. “Concurrency Programming Guide: Migrating Away from Threads.” developer.apple.com/library/archive/documentation/General/Conceptual/ConcurrencyProgrammingGuide. Accessed 23 November 2024.

[2] “Introduction to Grand Central Dispatch (GCD) and Multithreading in macOS.” Swift.org Documentation, Swift.org, swift.org/documentation/concurrency-guide. Accessed 23 November 2024.

[3] “Multithreaded Algorithms.” Gonzalez, Jesus. Crystal.uta.edu https://crystal.uta.edu/~gonzalez/alg/CSE-5311-15.html

[4] “Sysadmin.” *Home*, www.linuxjournal.com/article/1363. Accessed 6 Dec. 2024.

[5] “Software.” *Home*, www.linuxjournal.com/article/3138. Accessed 6 Dec. 2024.

[6] “Informit.” *InformIT*, www.informit.com/articles/article.aspx?p=370047&seqNum=3. Accessed 6 Dec. 2024.