Group Team: Himadri Saha, Ashwin Srinivasan, Yaritza Sanchez

EECE 4811

Q1 – HW1

**Difference between Goroutines and Threads**

Goroutines and operating system threads both allow programs to perform multiple tasks simultaneously, but they operate at different levels. In Article 2 by Mattoo, threads are described as the units that the operating system schedules on the CPU. Each thread has its own stack and registers, and the kernel determines which thread runs at any given time. Threads can execute in parallel on multiple cores, sharing the memory of their process, which facilitates fast communication but can also introduce risks if synchronization is not handled carefully. Mattoo's article also discusses common threading models and issues such as blocking system calls, context switch costs, race conditions, and deadlocks, which aligns with our class discussions on concurrency.

Goroutines, on the other hand, function above the kernel, managed by the Go runtime. In Article 1 by Boyle, it is explained that you initiate a goroutine by using the ‘go’ keyword before a function call. There is always a main goroutine, and the program exits when this main goroutine completes. Multiple goroutines can be multiplexed onto a smaller number of operating system threads. The article emphasizes two key considerations for design. First, the order in which goroutines are executed cannot be relied upon, and per-goroutine priorities cannot be set, the anticipatory scheduler makes these decisions for you. Second, while you can adjust ‘GOMAXPROCS’ (how Go decides how many OS threads can run Go code at the same time) to limit the number of operating system threads that Go uses simultaneously, increasing this value does not always lead to better performance. Extra context switching at the thread level can actually slow down the program. The article also suggests that switching between goroutines is much faster compared to switching between operating system threads.

Article 3 by Bansal aligns with the perspectives of the other two articles. It characterizes operating system threads as kernel-managed and heavier, whereas goroutines are lightweight tasks managed by the Go runtime. Bansal explains that the Go scheduler operates in user space, mapping many goroutines to fewer operating system threads. This design enables Go to efficiently manage a large number of concurrent tasks. The article also highlights Go channels as the preferred method for coordinating goroutines, which often feels simpler than directly managing locks on threads.

In summary, the design differences are clear. Threads are scheduled by the operating system and come with significant overhead, whereas goroutines are scheduled by the Go runtime and utilize a pool of threads. This design makes goroutines inexpensive to create, quick to switch, and scalable to thousands of concurrent tasks. However, there are trade-offs: you lose low-level control over order and priority, must ensure that the main goroutine does not terminate before the others finish, and changing ‘GOMAXPROCS’ without careful measurement can reduce performance. The potential for common concurrency issues still exists if data is shared irresponsibly, so using the patterns Go provides, such as channels, is essential for maintaining safety.

These sources are credible for this comparison as each addresses a different aspect while agreeing on core facts. Article 1 offers a focused overview of how goroutines are created and scheduled in real Go programs, including the role of ‘GOMAXPROCS’ and common pitfalls when the main goroutine exits prematurely. Article 2 by Mattoo provides a structured explanation of threads, how the kernel schedules them, and the problems and models associated with them, which aligns with standard operating systems courses. Article 3 by Bansal presents a direct comparison, reiterating the same conclusions drawn by Mattoo and explaining why goroutines are perceived as lighter and more scalable in Go. Because all three independent sources converge on the same design differences and each comes from its own area of expertise, their consensus enhances their credibility.

**References**

1. Boyle, M. (2024, October 11). *What are goroutines and how are they scheduled?* DEV Community. https://dev.to/gophers/what-are-goroutines-and-how-are-they-scheduled-2nj3

2. Mattoo, S. (2025, September 20). *Threads in operating system*. Intellipaat. <https://intellipaat.com/blog/threads-in-operating-system/>

3. Bansal, D. (2024, November 13). OS Threads vs Goroutines: Understanding the Concurrency model in GO. *Medium*. https://daminibansal.medium.com/os-threads-vs-goroutines-understanding-the-concurrency-model-in-go-bad187372c89