CS60203: Design Optimization of Computing Systems *October 10, 2024*

Concurrency and Lightweight Threading in Modern Computing Systems

Instructor: Mainack Mondal *Scribe-By: P V Mokshith*

1 Introduction

This lecture explores the concepts of concurrency and lightweight threading in modern computing systems, emphasizing the distinction between parallelism and concurrency, the challenges posed by OS threads, and the implementation of concurrency through event loops. Additionally, it delves into Python's asyncio event loop, extending concurrency to multi-core systems, and the internal mechanics of the Go runtime.

2 Parallelism vs. Concurrency

2.1 Parallelism

- Parallelism occurs when multiple tasks are executed simultaneously across multiple CPU cores. According to Sun, it exists when at least two threads execute concurrently.
- Enhances throughput by efficiently utilizing available CPU cores.
- Inefficient for tasks that are idle most of the time (e.g., waiting for I/O), as parallelizing these may not yield significant benefits.
- Examples include Instruction-Level Parallelism (ILP) and SIMD (Single Instruction, Multiple Data), where hardware executes multiple instructions in parallel.

2.2 Concurrency

- Concurrency involves tasks that can run simultaneously or overlap in execution but do not necessarily run at the same time. Sun defines it as existing when at least two threads are making progress, even if they are not executing simultaneously.
- Ideal for I/O-bound tasks (e.g., database or network requests) that spend significant time waiting for external resources.
- While parallelism improves throughput for CPU-bound tasks, concurrency enhances efficiency in I/O-bound tasks, thereby improving responsiveness and resource utilization.

3 Challenges in Achieving Concurrency with OS Threads

3.1 Overhead in OS Threads

- **Thread Creation Costs:** Starting or waking up an OS thread incurs significant overhead, often measured in thousands of clock cycles. For example, creating a simple thread (e.g., std::thread) on a Core i9 processor averages 50,202 nanoseconds.
- **Context Switching:** Switching between threads incurs a performance penalty, taking up to 20,000 instructions to regain normal performance post-system call.

3.2 Inefficiencies of OS Threads for Concurrency

- **Small Compute Tasks:** OS threads are inefficient for tasks where the computation time is minimal compared to the time required to start or switch threads, common in I/O-bound programs.
- **Control Flow:** OS threads are unsuitable for tasks involving control flow, as waiting for a thread to finish introduces latency.
- **Memory Overhead:** OS threads require substantial memory for their stacks (in MBs), and once allocated, memory cannot be decommitted, leading to inefficiency in scenarios with frequent task creation and destruction.

4 Implementing Concurrency Using Event Loops

4.1 Overview of Event Loops

- Event loops provide an efficient alternative to OS threads for managing concurrency, particularly in I/O-bound systems.
- **Operation:**
  1. A task is initiated and runs until it blocks (e.g., waiting for I/O).
  2. The task's state is saved upon blocking.
  3. The event loop checks if the task is unblocked and ready to resume.
  4. A task from the queue of ready tasks is selected and resumed.
  5. This cycle repeats, ensuring tasks progress without the need for new thread creation.

4.2 Achieving Concurrency Through Event Loops

- In an event-driven model, tasks wait for events (e.g., I/O completion), and the event loop dispatches tasks when ready to run.
- **Callback Mechanism:** Callbacks register tasks to run when events (e.g., unblocking) occur. For instance, if Task A runs and blocks, a callback is registered to execute when Task A is ready.

4.3 Real-World Example: Python's asyncio Event Loop

- Python's standard library for asynchronous I/O, structured around the event loop concept, enabling efficient execution of I/O-bound tasks.
- **Internal Structure:**
  - Tasks are queued in the self._ready queue for execution.
  - A selector manages tasks waiting on I/O (e.g., using epoll on Linux).
  - Tasks execute based on readiness, minimizing overhead from thread creation and context switching.
- **Event Loop Iteration (_run_once):**
  - Handles bookkeeping for delayed or canceled tasks.
  - Checks the self._ready queue for runnable tasks.
  - Executes tasks and schedules callbacks for future execution as needed.

5 Extending Concurrency to Multiple Cores

To extend concurrency to multiple cores, multiple threads must be created, generalizing to running m virtual threads on n OS threads.

5.1 Attempt 1: Event Loop with a Thread Pool

- Multiple threads run their own event loops, pulling tasks from a shared central ready queue.
  - **Central Ready Queue:** Holds tasks ready for execution, allowing threads to pull tasks as they become idle.
  - **Thread Pool:** Consists of multiple threads waiting to execute tasks from the shared queue.
  - **Task Execution:** Threads run tasks until they block (e.g., waiting for I/O), allowing the event loop to switch to other tasks.

5.1.1 Example: Old Go Runtime

- **Major Structs:**
  - **G:** Represent lightweight tasks for executing code.
  - **M:** Represents OS threads, each running one goroutine at a time.
  - **Sched (Global Scheduler):** Manages runnable queues and assigns tasks to threads under a global lock (Sched.Lock).
- **Execution:**
  - Idle threads (M) pick up runnable goroutines (G) from the global queue.
  - If a goroutine blocks, another thread is woken up to run the next task.
  - A single mutex (Sched.Lock) manages access to the task queue.

5.2 Shortcomings of Attempt 1 and the Old Go Runtime

- **Bottleneck in Central Queue:** Contention arises as all threads access the same ready queue, with a global lock limiting throughput.
- **Memory Inefficiency:** Threads hold memory resources even when idle, leading to waste.
- **Blocking/Unblocking Overhead:** High management overhead from frequent blocking/unblocking during I/O operations.

5.3 Attempt 2: Work-Stealing Scheduler

This attempt improves concurrency by replacing the central queue with local task queues for each P, mitigating contention issues. This attempt also introduces indirect ownership of resources to tackle other bottlenecks.

5.3.1 Example: Work-Stealing Scheduler in Go

- **Major Structs:**
  - **G:** Represent lightweight tasks for executing code.
  - **M:** Represents OS threads, each running one goroutine at a time.
  - **P (Processors):** Essential for M to execute code, managing the free goroutines list, managing memory using mcache shifted into P from M.
- **Execution:**
  - When an OS thread (M) is ready to execute Go code, it pops a processor (P) from a free list.
  - New or runnable goroutines (G) are added to the local runnable list of the current processor (P).
  - If the local list is empty, the processor attempts to steal half of another processor's runnable goroutines.

5.4 Improvements from Attempt 1

1. **Elimination of Central Ready Queue:** Local queues reduce contention and enable simultaneous execution.
2. **Dynamic Work Stealing:** Idle threads can efficiently steal tasks from local queues, optimizing task processing.
3. **Reduced Lock Contention:** Lower contention leads to improved performance and better CPU utilization.
4. **Improved Load Balancing:** Work stealing dynamically balances loads, keeping threads active.
5. **Efficient Memory Usage:** As mcache is shifted from M to P, memory is only held in P instead of each M which reduces memory wastage.

6 Comparing Goroutines and OS Threads

6.1 Goroutines

- **Lightweight:** Require minimal memory and stack space (KBs vs. MBs).
- **Cooperative Scheduling:** More efficient context switching (only 3 fields are saved/restored).
- **I/O-Bound Tasks:** Capable of efficiently handling up to 100k concurrent operations without blocking.

6.2 OS Threads

- **Resource-Intensive:** Larger memory footprints and slower context switches.
- **Preemptive Scheduling:** Less efficient for managing high numbers of concurrent tasks.

7 Conclusion

Concurrency is a crucial aspect of modern computing systems, providing efficient task management and improved resource utilization. Through event loops and lightweight threading, systems can achieve high levels of concurrency and performance, particularly in I/O-bound applications. The exploration of work-stealing schedulers exemplifies the evolution of concurrency mechanisms, ensuring scalability and efficiency across multi-core architectures.

8 Resources

- To know more about asyncio: asyncio documentation
- To know more about stacks in Go: How stacks are handled in Go
- To know more about m virtual threads mapping to n os threads: Lightweight Threads Paper