Go Programming Language

Memory synchronization

Department of Computer Science and Engineering

Go Programming Language

Memory synchronization

- In a modern computer there may be dozens of processors, each with its own local cache of the main memory.
- For efficiency, writes to memory are buffered within each processor and flushed out to main memory only when necessary.
- They may even be committed to main memory in a different order than they were written by the writing goroutine.
- Synchronization primitives like channel communications and mutex operations cause the processor to flush out and commit all its accumulated writes so that the effects of goroutine execution up to that point are guaranteed to be visible to goroutines running on other processors.

Go Programming Language

Memory synchronization

- Consider the possible outputs of the following snippet of code:

```
var x, y int
go func() {
    x = 1                   // A1
    fmt.Print("y:", y, " ") // A2
}()
go func() {
    y = 1                   // B1
    fmt.Print("x:", x, " ") // B2
}()
```

- Since the two statements in the functions are concurrent and access shared variables without mutual exclusion, there is a data race, so we should not be surprised that the program is not deterministic.
- We might expect it to print any one of these four results, which correspond to intuitive interleavings of the labeled statements of the program:

```
y:0 x:1
x:0 y:1
x:1 y:1
y:1 x:1
```

Go Programming Language

Memory synchronization

- Within a single goroutine, the effects of each statement are guaranteed to occur in the order of execution;
- goroutines are sequentially consistent.
- But in the absence of explicit synchronization using a channel or mutex, there is no guarantee that events are seen in the same order by all goroutines.
- Although goroutine A must observe the effect of the write x = 1 before it reads the value of y, it does not necessarily observe the write to y done by goroutine B, so A may print a stale value of y.
- If the two goroutines execute on different CPUs, each with its own cache, writes by one goroutine are not visible to the other goroutine's Print until the caches are synchronized with main memory.

Go Programming Language

Semaphores

- **Mutex** constrains access to a single thread, to guard a critical section of code.
- **Semaphore** constrains access to at most **N threads**, to control/limit concurrent access to a shared resource.
- Implementation using a semphore
    1. Instantiate a semaphore with some defined limit (N).
    2. Call Acquire() every time access into a shared resource is requested
    3. Call Release() every time access into a shared resource is finished

https://blog.stackademic.com/go-concurrency-visually-explained-semaphore-3ffe23f11388

Threads

- Each OS thread has a fixed-size block of memory (often as large as 2MB) for its stack, the work area where it saves the local variables of function calls that are in progress or temporarily suspended while another function is called.
- This fixed-size stack is simultaneously too much and too little. A 2MB stack would be a huge waste of memory for a little goroutine, such as one that merely waits for a WaitGroup then closes a channel.
- It’s not uncommon for a Go prog ram to create hundreds of thousands of goroutines at one time, which would be impossible with stacks this large.
- Yet despite their size, fixed-size stacks are not always big enough for the most complex and deeply recursive of functions.
- Changing the fixed size can improve space efficiency and allow more threads to be created, or it can enable more deeply rec ursive functions, but it cannot do both.

Go Programming Language

Growable stacks

- A goroutine starts life with a small stack, typically 2KB.
- A goroutine's stack, like the stack of an OS thread, holds the local variables of active and suspended function calls, but unlike an OS thread, a goroutine's stack is not fixed; it grows and shrinks as needed.
- The size limit for a goroutine stack may be as much as 1GB, orders of magnitude larger than a typical fixed-size thread stack, though of course few goroutines use that much.

Goroutine scheduling

- OS threads are scheduled by the OS kernel. Every few milliseconds, a hardware timer interrupts the processor, which causes a kernel function called the scheduler to be invoked.
- This function suspends the currently executing thread and saves its registers in memory, looks over the list of threads and decides which one should run next, restores that thread's registers from memory, then resumes the execution of that thread.
- Because OS threads are scheduled by the kernel, passing control from one thread to another requires a full context switch, that is, saving the state of one user thread to memory, restoring the state of another, and updating the scheduler's data structures.
- This operation is slow, due to its poor locality and the number of memory accesses required, and has historically only gotten worse as the number of CPU cycles required to access memory has increased.

Go Programming Language

Goroutine scheduling

- The Go runtime contains its own scheduler that uses a technique known as m:n scheduling, because it multiplexes (or schedules) m goroutines on n OS threads.
- The job of the Go scheduler is analogous to that of the kernel scheduler, but it is concerned only with the goroutines of a single Go program.
- Unlike the operating system's thread scheduler, the Go scheduler is not invoked periodically by a hardware timer, but implicitly by certain Go language constructs.
- For example, when a goroutine calls time.Sleep or blocks in a channel or mutex operation, the scheduler puts it to sleep and runs another goroutine until it is time to wake the first one up.
- Because it doesn't need a switch to kernel context, rescheduling a goroutine is much cheaper than rescheduling a thread.
- The Go scheduler uses a parameter calle d GOMAXPROCS to deter mine how many OS threads may be actively executing Go code simultaneously

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering