Notebook - The Golang Scheduler

kindle

Kevin Kelche

Page 4 | Highlight

> a scheduler is a program that decides which process should be executed next.

Page 4 | Highlight

> The Golang scheduler is responsible for scheduling the goroutines to the kernel threads.

Page 7 | Highlight

> scheduler is responsible for adding goroutines to the run queue and removing them from the run queue. The run queue follows the FIFO

Page 7 | Highlight

> A mutex is a lock that is used to synchronize access to a shared resource. In our case, the shared resource is the run queue.

Page 7 | Highlight

> mutex to make sure that only one kernel thread can access the run queue at a time.

Page 10 | Highlight

kindle

> starvation

Note:

Look into work Starvation!

Page 10 | Highlight

> Preemption is the ability of the scheduler to preempt (stop) a goroutine that is currently running and execute another goroutine.

Page 10 | Highlight

> if a goroutine is running for more than 10ms, the scheduler will preempt it and execute another goroutine.

Page 10 | Highlight

> preempted goroutine will be added to the end of the global run queue which is a FIFO queue. This means that the goroutine will be executed after all the other goroutines in the global run queue.

Page 11 | Highlight

kindle

> local run queue is a run queue that is local to a kernel thread. This means that each kernel thread has its run queue.

Page 11 | Highlight

> each kernel thread will have its run queue and will not be affected by the other kernel threads in terms of resource sharing.

Page 11 | Highlight

> If the local run queue is full, the goroutine will be added to the global run queue. The global run queue is a queue that is shared between all the kernel threads.

Page 12 | Highlight

> if a thread is blocked in a system call, the thread does not need to maintain its local run queue. Subsequently, those goroutines will be run elsewhere.

Page 12 | Highlight

> The P is in charge of managing the interaction between the local run queues, the global run queue, and the kernel threads.

Page 13 | Highlight

> Work stealing is a technique that is used to balance the load between the processor's kernel threads.

Page 13 | Highlight

> a processor is idle, it will steal work from another processor or the global run queue, or the network poller.

> taking half of the work from the other processor's run queue and adding it to its run queue.

Page 14 | Highlight

> Before a processor starts stealing work, it will first check its run queue. If the run queue is not empty, it will pull a goroutine from the run queue and execute it. If the run queue is empty, it will then check the global run queue. It only checks the run queue 1/61 of the time. This is done to avoid the overhead of checking the run queue all the time. There is a tick that counts the number of times we have checked the local run queue, once it reaches 61 or a multiple of 61, we check the global run queue. This is important to avoid the starvation of goroutines in the global run queue.

Note:

A processor will check the global run queue 1/61 of the time to make sure global run queue does not starve.

Page 20 | Highlight

kindle

> time slice inheritance.

Note:

What is this?

Page 22 | Highlight

> When this happens, the go runtime will call releasep to release the processor. This will disassociate the processor from the kernel thread. The processor will then be assigned a new kernel thread that is already available or a new kernel thread will be created.

Note:

If a kernel thread gets blcked on a process, the processor will release from a thread and assign itself to an available kernel thread or a kernel thread is created.

> go scheduler will do immediate handoff if it knows that the syscall will be blocking for a long time. For instance, doing a read on a socket will block the kernel thread for a long time.

> other cases, it will let the processor be in a block. It will then set the status to reflect that it is in a syscall. Using Sysmon the go runtime will periodically check if the processor is still in a syscall.