## CPU Scheduler

CPUs today come with multiple cores – these multicore processors are optimized to handle simultaneous execution – also known as parallel processing.

CPU scheduler manages the incoming multiple jobs and distributes them among the available processors. process is known as *scheduling*.

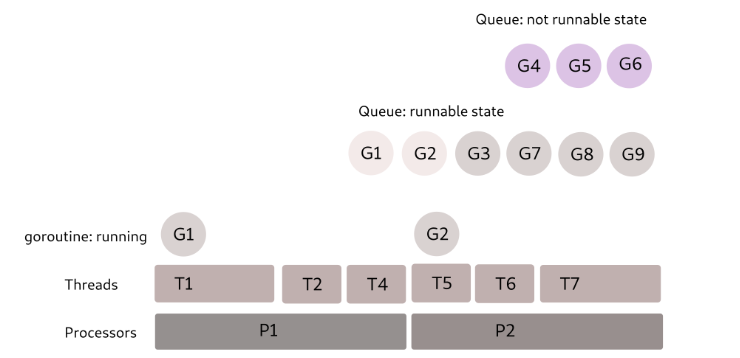
A *scheduler* schedules jobs at the software level

Go’s Runtime Scheduler

The Go runtime scheduler schedules goroutines.

A goroutine is a lightweight thread that has the ability to execute on a single OS thread.

 The OS threads run on single or multiple available processors. The runtime scheduler of Go distributes goroutines over multiple threads.

The scheduler determines the state of the goroutine. A life cycle of the goroutine can be in one of three fundamental states : *running*, *runnable*, and *not runnable* (due to IO blocked or system call):

Go works on a type of scheduler called an **m:n scheduler (M:N scheduler)**, which states that **M** number of goroutines can be distributed over **N** number of OS threads

OS threads have much more overhead than goroutines. Therefore, Go uses a limited number of threads to run a maximum number of goroutines.

Goroutines are user-space threads managed entirely by the Go runtime and the runtime scheduler schedules them.

This makes goroutines cheaper, more lightweight than kernel threads, and they run on a very small memory footprint.

|  |  |
| --- | --- |
| Type of thread | Allocated initial stack sized |
| OS thread | 8kb |
| Goroutines | 2kb |

The runnable goroutines (shown in the above figure) are picked from the queue to run over available OS threads, which, in turn, run on one or more available processors.

The goroutines that are blocked are put into a *not runnable* state queue.

Once unblocked, the goroutine is put back on the *runnable* queue and waits for its turn to run on the available OS thread.

## Fork-join Concurrency Model in Go

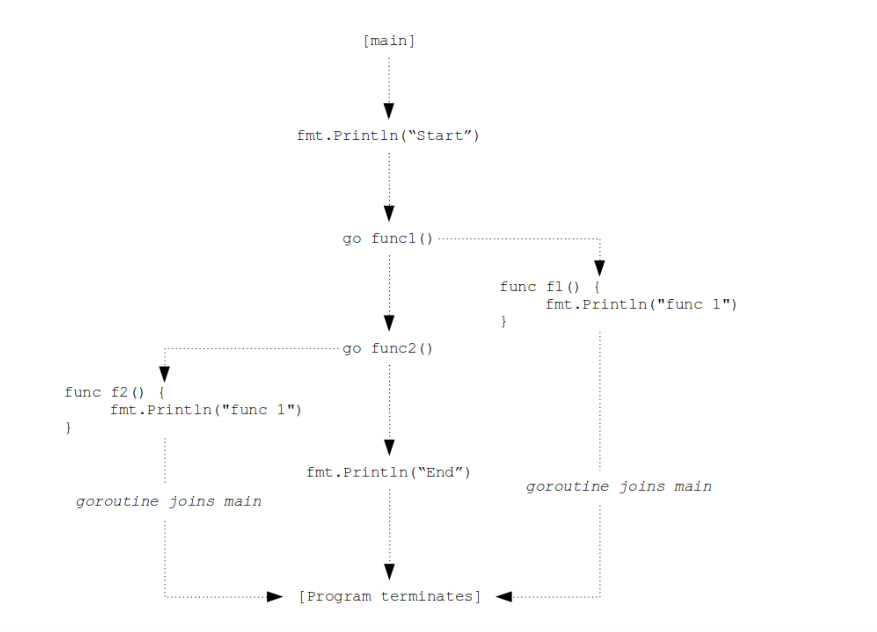
Go uses the *fork-join concurrent execution strategy* to execute programs in parallel.

This enables the Go program to branch its own execution path to be run with its main branch.

This splitting branch can coincide at some later point and run as a single execution path.

The fork part of the model states branching off of code at designated points and the join part states reuniting back to the caller after execution finishes.

 Let’s try to understand this with the help of an example. Here is a simple program showing how to perform fork-join concurrency in Go and Golang: <https://go.dev/play/p/VOz2lrLU8Ex>



The **main** method begins with a linear execution path and the designated split point is the function call with the **go** keyword (**go func1()**).

Similarly, another function call, **go func2()** splits into another execution path.

Both the functions join back to the source after finishing their execution.

The main program waits courtesy of **time.Sleep(1\*time.Second)** for a constant time so that the child branches finish their execution in the meantime.

Try executing the same code by commenting out the the line: **time.Sleep(1\*time.Second)**. The output will be as follows

The output from **func1** and **func2** will not be displayed.

 This is because the main program terminates before the child branches and rejoins.

This means the **main** goroutine must wait till the forked child is able to rejoin its parent.

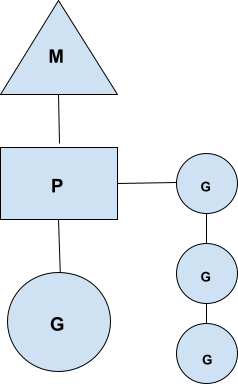
The function **time.Sleep** makes force wait possible in this case. A better way to write concurrent execution code is with the help of **WaiteGroup** from the **sync** package.

## Work Stealing Strategy in Go

The work stealing strategy the Go scheduler looks for any logical underutilized processor and steals some processing time for the runnable goroutines to execute

## ****Go Runtime Scheduler****

* **M:**M represents the OS thread, which is entirely managed by the OS, and it’s similar to POSIX thread. M stands for machine**.**
* **G:**G represents the goroutine. Now, a goroutine is a resizable stack that also includes information about scheduling, any channel it’s blocked on, etc**.**
* **P:**P is a context for scheduling. This is like a single thread that runs the Go code to multiplex M goroutines to N OS threads. This is important part, and that’s why P stands for processor.



The P processor basically holds the queue of runnable goroutines—or simply run queues.

So, anytime the goroutine (G) wants to run it on a OS thread (M), that OS thread first gets hold of P i.e., the context.

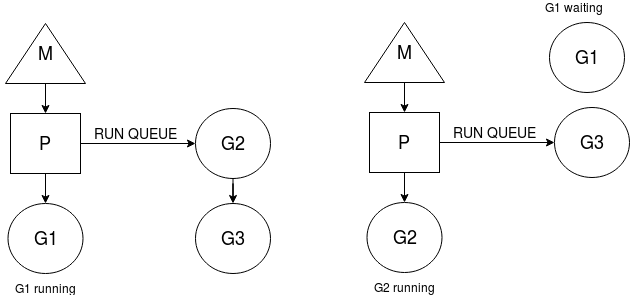
Now, this behaviour occurs when a goroutine needs to be paused and some other goroutines must run.

One such case is a buffered channel. When the buffer is full, we pause the sender goroutine and activate the receiver goroutine.

Imagine the above scenario:

G1 is a sender that tries to send a full buffered channel, and G2 is a receiver goroutine. Now, when G1 wants to send a full channel, it calls into the runtime Go scheduler and signals it as **gopark.**

 So, now scheduler, or M, changes the state of G1 from running to waiting, and it will schedule another goroutine from the run queue, say G2.



As you can see, after the **gopark** call, G1 is in a waiting state and G2 is running.

We haven’t paused the OS thread (M); instead, we’ve blocked the goroutine and scheduled another one.

So, we are using maximum throughput of an OS thread.

The context switching of goroutine is handled by the scheduler (P), and because of this, it adds complexity to the scheduler.

This is great. But how do we resume G1 now because it still wants to add the data/task on a channel, right?

So, before G1 sends the **gopark**signal, it actually sets a state of itself on a **hchan struct,**

i.e., our channel in the **sendq**field. Remember the **sendq and recvq**fields? They’re waiting senders and receivers.

Now, G1 stores the state of itself as a **sudog** struct. A sudog is simply a goroutine that is waiting on an element.

The sudog struct has these elements:

type sudog struct{

g \*g

isSelect bool

next \*sudog

prev \*sudog

elem unsafe.Pointer //data element

...

}

* g is a waiting goroutine,
* next and prev are the pointers to sudog/goroutine respectively if there’s any next or previous goroutine present,
* and elem is the actual element it’s waiting on.

So, considering our example, G1 is basically waiting to write the data so it will create a state of itself, which we’ll call sudog as below:

Cool. Now we know, before going into the waiting state, what operations G1 performs. Currently, G2 is in a running state, and it will start consuming the channel data.

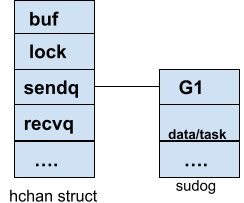
As soon as it receives the first data/task, it will check the waiting goroutine in the sendq attribute of an hchan struct, and it will find that G1 is waiting to push data or a task.

Now, here is the interesting thing: **G2 will copy that data/task to the buffer**,

and it will call the scheduler, and the scheduler will put G1 from the waiting state to runnable,

and it will add G1 to the run queue and return to G2.

This call from G2 is known as **goready**

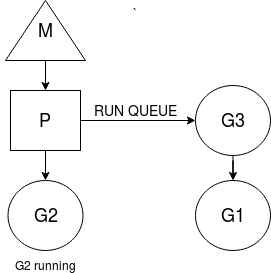


Golang behaves like this because when G1 runs, it doesn’t want to hold onto a lock and push the data/task.

That extra overhead is handled by G2.

That’s why the sudog has the data/task and the details for the waiting goroutine.

So, the state of G1 is like this:



As you can see, G1 is placed on a run queue.

Now we know what’s done by the goroutine and the go scheduler in case of buffered channels.

 In this example, the sender gorountine came first, but what if the receiver goroutine comes first?

What if there’s no data in the channel and the receiver goroutine is executed first?

The receiver goroutine (G2) will create a **sudog**in **recvq** on the **hchan struct**.

Things are a little twisted when G1 goroutine activates.

It will now see whether there are any goroutines waiting in the recvq, and if there is, it will copy the task to the waiting goroutine’s (G2) memory location,

i.e., the **elem**attribute of the sudog.

This is incredible! Instead of writing to the buffer, it will write the task/data to the waiting goroutine’s space simply to avoid G2’s overhead when it activates.

We know that each goroutine has its own resizable stack, and they never use each other’s space except in case of channels.