Understanding Channels in Go

Normally, when we talk about channels, we think of the ones in applications like RabbitMQ, Redis, AWS SQS, and so on. Anyone with no or only a small amount of Golang knowledge would think like this. But **Channels in Golang are different from a work queue system**

In the work queue system like above, there are TCP connections to the channels, but in Go, the channel is a data structure or  even a design pattern, which we’ll explain later.

Channels are the medium through which goroutines can communicate with each other. In simple terms, a channel is a pipe that allows a goroutine to either put or read the data.

A channel is a communication medium for goroutines

## ****Types of channels:****

* **Unbuffered channel:** This is what we have seen above. A channel that can hold a single piece of data, which has to be consumed before pushing other data. That’s why our main goroutine got blocked when we added data into the channel.
* **Buffered channel:**In a buffered channel, we specify the data capacity of a channel. The syntax is very simple. c := make(chan int,10)  the second argument in the make function is the capacity of a channel. So, we can put up to ten elements in a channel. When the capacity is full, then that channel would get blocked so that the receiver goroutine can start consuming it.

**Properties of a channel:**

A channel does lot of things internally, and it holds some of the properties below:

* Channels are goroutine-safe.
* Channels can store and pass values between goroutines.
* Channels provide FIFO semantics.
* Channels cause goroutines to block and unblock

## ****Channel Structure:****

The channel internally behaves in that fashion. It has a circular queue, a lock, and some other fields.

When we do this c := make(chan int,10) Go creates a channel using **hchan struct,**which has the following fields:

type hchan struct {

qcount uint // total data in the queue

dataqsiz uint // size of the circular queue

buf unsafe.Pointer // points to an array of dataqsiz elements

elemsize uint16

closed uint32

elemtype \*\_type // element type

sendx uint // send index

recvx uint // receive index

recvq waitq // list of recv waiters

sendq waitq // list of send waiters

// lock protects all fields in hchan, as well as several

// fields in sudogs blocked on this channel.

//

// Do not change another G's status while holding this lock

// (in particular, do not ready a G), as this can deadlock

// with stack shrinking.

lock mutex

}

This is what a channel is internally. Let’s see one-by-one what these fields are.

**qcount** holds the count of items/data in the queue.

**dataqsize**is the size of a circular queue. This is used in case of buffered channels and is the second parameter used in the make function.

**elemsize**is the size of a channel with respect to a single element.

**buf**is the actual circular queue where the data is stored when we use buffered channels.

**closed** indicates whether the channel is closed. The syntax to close the channel is close(<channel\_name>). The default value of this field is 0, which is set when the channel gets created, and it’s set to 1 when the channel is closed.

**sendx and recvx** indicates the current index of a buffer or circular queue. As we add the data into the buffered channel, **sendx**increases, and as we start receiving, **recvx**increases.

**recvq and sendq**are the waiting queue for the blocked goroutines that are trying to either read data from or write data to the channel.

**lock**is basically a mutex to lock the channel for each read or write operation as we don’t want goroutines to go into deadlock state.

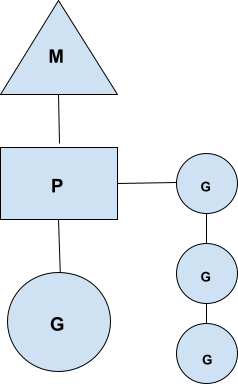
These are the important fields of a hchan struct, which comes into the picture when we create a channel.

This **hchan struct**basically resides on a **heap** and the make function gives us a pointer to that location.

**Read and write operations on a channel:**

We are considering buffered channels in this. When one goroutine, let’s say G1, wants to write the data onto a channel, it does following:

* **Acquire the lock:**As we saw before, if we want to modify the channel, or hchan struct, we must acquire a lock. So, G1 in this case, will acquire a lock before writing the data.
* **Perform enqueue operation:**We now know that buf is actually a circular queue that holds the data. But before enqueuing the data, goroutine does a memory copy operation on the data and puts the copy into the buffer slot. We will see an example of this.
* **Release the lock:**After performing an enqueue operation, it just releases the lock and goes on performing further executions.



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.

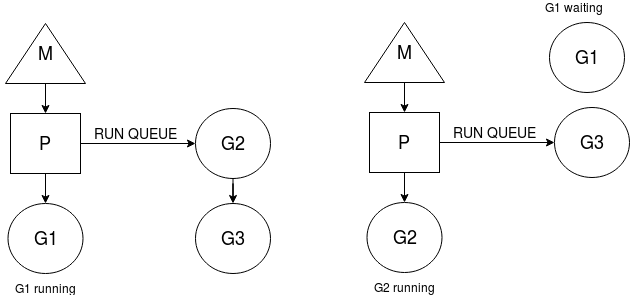
Now, this behaviour occurs when a goroutine needs to be paused and some other goroutines must run.

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.

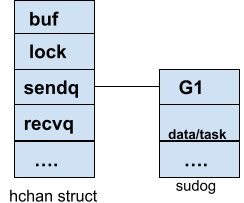
As soon as it receives the first data/task, it will check the waiting goroutine in the sendq attribute of an hchan struct, and it will find that G1 is waiting to push data or a task.

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**

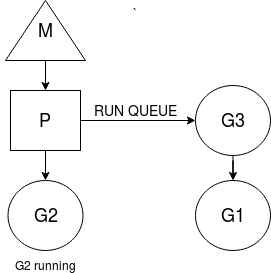


Golang behaves like this because when G1 runs, it doesn’t want to hold onto a lock and push the data/task.

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.