cat /dev/head

Sunday, August 14, 2016

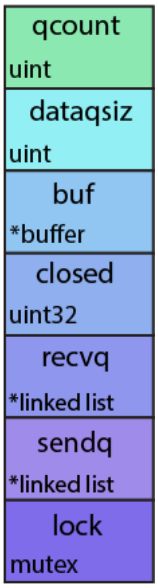
Golang: channels implementation

Introduction

Go is getting more and more popular nowadays, one of the reasons for that is great support of concurrency. Channels and goroutines simplify programming of concurrent applications. There are several articles that give an insight into how Go data structures are implemented, for example: slices, maps, interfaces, however there is little information about the implementation of Go channels. In this article you will explore how channels work and how they are implemented. (If you have never used channels in Go I'd recommend to read this.)

Channel structure

Let's start with the definition of channel structure:



- qcount - number of elements in the buffer
- dataqsiz - buffer capacity
- buf - pointer to a buffer for channel elements
- closed - flag that shows whether the channel closed or not
- recvq - pointer to a linked list of goroutines waiting to receive elements from the channel
- sendq - pointer to a linked list of goroutines waiting to send elements to the channel
- lock - mutex for concurrent access to the channel

In general goroutine acquires the lock when it performs an action on the channel, except in cases where it does lock free checks in non-blocking function call (I'll explain more about that later). Closed - is a flag that is set to 1 when channel is closed and 0 when it's not. These fields will be omitted from the pictures for clarity.

A channel can be synchronous (unbuffered) or asynchronous (buffered). Let's first see how synchronous channels work.

Blog Archive
▼ 2016 (1)
▼ August (1)
Golang: channels implementation

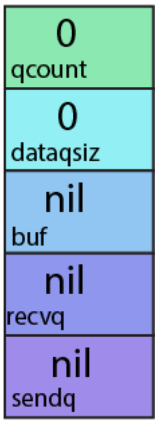
Synchronous channels

Given the following sample of code:

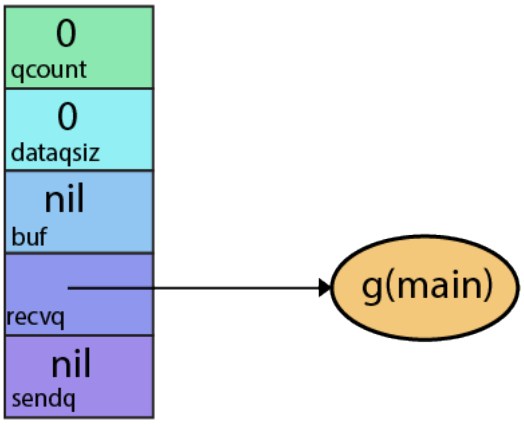
```
package main


func main() {
  ch := make(chan bool)
  go func() {
    ch <- true
  }()
  <-ch
}
```

New channel is created, and it looks like this:



Go doesn't allocate a buffer for synchronous channels, so the pointer to buffer is nil and `dataqsiz` is zero. In the code above Go doesn't guarantee what happens first - receive from the channel or send to the channel. Let's assume that the first action would be receiving from the channel (Reverse order of operations is covered in the paragraph about buffered channels). First of all, the running goroutine does some checks, such as: checking if channel is closed, whether it's buffered or not, if it contains goroutines in the send queue. In this situation the channel has neither a buffer nor queued senders, so the goroutine adds itself to the `recvq` and blocks. After this step the channel looks like this:



Now there is only one goroutine that is ready to run and that goroutine tries to send data to the channel. All the checks are repeated again and when goroutine checks `recvq`, it finds there the waiting goroutine, removes it from the receive queue and writes the data to the waiting goroutine's stack and awakes it. This is the only place in Go runtime where one goroutine writes to the stack of another goroutine. After this step the channel looks like it was after initialization. Both goroutines return and the program terminates.

That is all about synchronous channels. Now let's have a look at buffered channels.

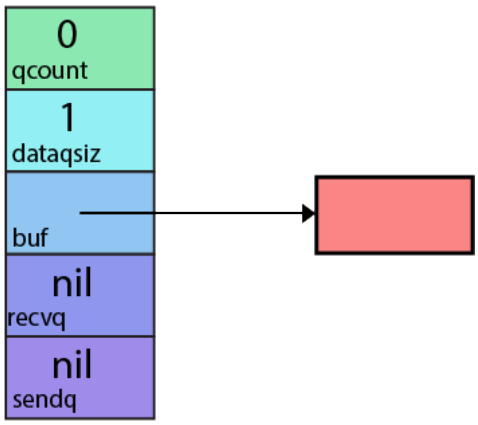
Buffered channels

Consider the following example:

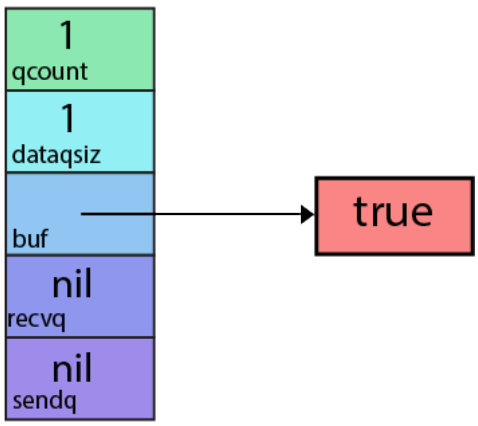
```
package main


func main() {
  ch := make(chan bool, 1)
  ch <- true
  go func() {
    <-ch
  }()
  ch <- true
}
```

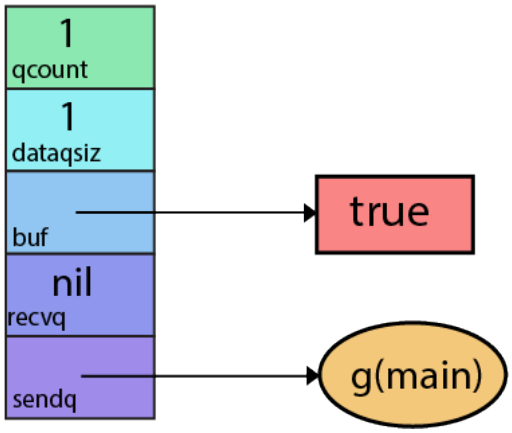
Again the order of goroutine execution is unknown, the example with receiving from the channel as the first operation was covered in the previous paragraph, so here let's assume that two elements are sent to the channel and after that one of the elements is received. The first step is to create the channel, it looks like this:



The difference in comparison with synchronous channels is that Go allocates a buffer and sets the `dataqsiz` field to one.
Next step is to send the first value to the channel. To perform this action, the goroutine does several checks like: checking whether `recvq` is empty, if the buffer is empty, whether there is free space in the buffer.
In this case there is free space in the buffer and there are no goroutines receiving from the channel, so it just writes the element to the buffer, increments `qcount` and continues execution. The channel looks like this:



In the next step, the main goroutine sends the second value to the channel. When the buffer is full, a buffered channel behaves like a synchronous (unbuffered) channel, meaning that the goroutine adds itself to the waiting list and blocks, as a result the channel structure looks like this:



Now the main goroutine is blocked and Go runs the anonymous goroutine that receives a value from the channel. Here comes the tricky part. Go guarantees that a channel works as FIFO queue (specification), but the goroutine cannot just get the value from the buffer and continue execution. In that case the main goroutine would block forever. To handle this, the running goroutine reads a value from the buffer, then appends the value from the first waiting goroutine to the buffer and unlocks the waiting goroutine and removes it from the queue. (In the case of no queued senders it will simply read the first entry from the buffer).

Select

But wait, Go supports the select statement with a default case, and if a channel blocks how would it execute the default case?
Good question. Let's have a brief look at private channels' API. When you run the following piece of code:

```
select:
case <-chan:
    foo()
default:
    bar()
```

Go executes a function with the following signature:

```
func chanrecv(t *chantype, c *hchan, ep unsafe.Pointer, block bool)
```

`chantype` is the type of the channel (e.g. bool for make(chan bool)), `hchan` is a pointer to the channel structure, `ep` is a pointer to a segment of memory where data from the channel should be written, and the last but most interesting for us parameter is the `block` flag. If it's set to `false` the function is running in non-blocking mode. In non-blocking mode a goroutine checks the buffer, the waiting queue and returns `true` and writes the data to `ep` or returns `false` if there is no data in the buffer or senders in the queue. Buffer and queue checks are done with atomic operations instead of locking the mutex.
There is also a function to send data to a queue with the similar signature.

Receive and send operations have been covered, now let's have a look at what happens when a channel is closed.

Closing a channel

Closing a channel is a simple operation, Go iterates through all the senders and receivers in the queues and then unlock them. All receivers get default values, for the data type, from the channel and all the senders panic.

Summary

In this article you have explored how channels are implemented and work. I tried to describe this as simple as possible, so omitted some details. The goal of the article is to provide a basic understanding of Go channels and to encourage you to read the source code if you want to get a more complete understanding. Just check channels' source code. I have found it simple, well-documented and quite short, just around 700 lines of code.

Resources:

Source code
Channels in Go specification
Go channels on steroids

Posted by Dmitry Vorobev at 11:23 AM

Labels: go, golang

No comments:

Post a Comment

Home

Subscribe to: Post Comments (Atom)