# A new sync primitive in golang

a new synchronization primitive for golang...

I've been working on lots of new stuff in mgmt (https://github.com/purpleidea/mgmt/) and I had a synchronization problem that needed solving... Long story short, I built it into a piece of re-usable functionality, exactly like you might find in the *sync (https://golang.org/pkg/sync/)* package. For details and examples, please continue reading...

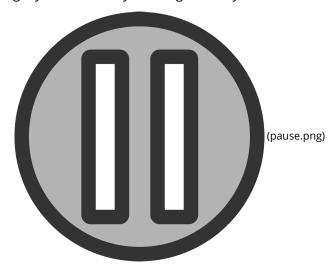
### **The Problem:**

I want to multicast a signal to an arbitrary number of goroutines. As you might already know, this can already be done with a chan struct{} . You simply close the channel to send the signal, and anyone running a select on that channel will return when it closes.

I'd like to do all of that, however I'd also like for the initiating close signal to wait until everyone has acknowledged the multicast signal before it continues.

Lastly, I'd like for this to be re-usable in that I'd like this to work for another cycle after the first synack iteration finishes.

Feel free to pause reading if you'd like to try building this on your own first.



## Code:

Here's what the code looks like. It's surprisingly short.

1 of 7 2018-05-11, 5:30 p.m.

```
type SubscribedSignal struct {
          *sync.WaitGroup
    exit chan struct{}
    mutex *sync.RWMutex
}
func NewSubscribedSignal() *SubscribedSignal {
    return &SubscribedSignal{
        wq:
               &sync.WaitGroup{},
        exit: make(chan struct{}),
        mutex: &sync.RWMutex{},
   }
}
func (obj *SubscribedSignal) Subscribe() (<-chan struct{}, func()) {</pre>
    obj.mutex.Lock()
    defer obj.mutex.Unlock()
    obj.wg.Add(1)
    return obj.exit, func() { // cancel/ack function
        obj.wg.Done()
        // wait for the reset signal before proceeding
        obj.mutex.RLock()
        defer obj.mutex.RUnlock()
    }
}
func (obj *SubscribedSignal) Send() {
    obj.mutex.Lock()
    defer obj.mutex.Unlock()
    close(obj.exit) // send the close signal
    obj.wg.Wait() // wait for everyone to ack
    obj.exit = make(chan struct{}) // reset
   // release (re-use the above mutex)
}
```

#### **Explanation**:

You start by creating a single \*SubscribedSignal with NewSubscribedSignal(). You can then Subscribe to it any number of times. When Subscribe returns, you can guarantee that you are now successfully subscribed. It will return two values:

The first is the multicast channel which closes when the signal is sent. You can wait on this channel in a select statement.

The second is an acknowledge ( ack ) function which you *must* run after you have received the signal or if you are no longer interested in waiting for the signal and you wish to cancel.

Once the signal is sent via Send , it will only unblock and terminate once all the subscribed signals have replied by running their individual ack functions. If you do not do so, you will block indefinitely.

At this point it is safe to run Subscribe again and repeat the process.

#### **Sneaky Races**:

Keep in mind that these types of problems have some sneaky races. For example, if the cancel/ack function didn't have the read lock, then the system wouldn't wait for all the individual subscribed workers to ack before continuing. It's preferable that it is a read lock so that they're all released simultaneously, instead of sequentially which depending on your containing code could cause a deadlock. This made it tricky to come up with, but elegant now that it's done. I was particularly happy that I was able to reuse the mutex, so that only one was necessary.

#### Example:

Here's a full usage example in the golang testable example (https://blog.golang.org/examples) format:

3 of 7 2018-05-11, 5:30 p.m.

```
func ExampleSubscribeSync() {
    fmt.Println("hello")
    x := NewSubscribedSignal()
    wg := &sync.WaitGroup{}
    ready := &sync.WaitGroup{}
    // unit1
    wg.Add(1)
    ready.Add(1)
    go func() {
        defer wg.Done()
        ch, ack := x.Subscribe()
        ready.Done()
        select {
        case <-ch:
            fmt.Println("got signal")
        time.Sleep(1 * time.Second) // wait a bit for fun
        fmt.Println("(1) sending ack...")
        ack() // must call ack
        fmt.Println("done sending ack")
    }()
    // unit2
    wg.Add(1)
    ready.Add(1)
    go func() {
        defer wg.Done()
        ch, ack := x.Subscribe()
        ready.Done()
        select {
        case <-ch:
            fmt.Println("got signal")
        }
        time.Sleep(2 * time.Second) // wait a bit for fun
        fmt.Println("(2) sending ack...")
        ack() // must call ack
        fmt.Println("done sending ack")
    }()
    // unit3
    wg.Add(1)
    ready.Add(1)
    go func() {
        defer wg.Done()
        ch, ack := x.Subscribe()
        ready.Done()
        select {
```

```
case <-ch:
            fmt.Println("got signal")
        time.Sleep(3 * time.Second) // wait a bit for fun
        fmt.Println("(3) sending ack...")
        ack() // must call ack
        fmt.Println("done sending ack")
    }()
    ready.Wait() // wait for all subscribes
    fmt.Println("sending signal...")
    x.Send() // trigger!
    fmt.Println("done sending signal")
    wg.Wait() // wait for everyone to exit
    fmt.Println("exiting...")
   // Output: hello
   // sending signal...
   // got signal
   // got signal
   // got signal
   // (1) sending ack...
   // (2) sending ack...
   // (3) sending ack...
   // done sending signal
   // done sending ack
   // done sending ack
   // done sending ack
   // exiting...
}
```

#### **Conclusion:**

Hope you enjoyed this. Leave me a comment if you build your own synchronization primitive.

Happy Hacking,

James

You can follow James on Twitter (https://twitter.com/intent/follow?screen\_name=purpleidea) for more frequent updates and other random thoughts.

You can support James on Patreon (https://www.patreon.com/purpleidea) if you'd like to help sustain this kind of content.

May 11, 2018

5 of 7 2018-05-11, 5:30 p.m.