Use a sync.Mutex or a channel?

One of Go's mottos is "*Share memory by communicating, don't communicate by sharing memory.*"

That said, Go does provide traditional locking mechanisms in the sync package. Most locking issues can be solved using either channels or traditional locks.

So which should you use?

Use whichever is most expressive and/or most simple.

A common Go newbie mistake is to over-use channels and goroutines just because it's possible, and/or because it's fun. Don't be afraid to use a `sync.Mutex` if that fits your problem best. Go is pragmatic in letting you use the tools that solve your problem best and not forcing you into one style of code.

As a general guide, though:

Channel	Mutex
passing ownership of data, distributing units of work, communicating async results	caches, state

If you ever find your sync.Mutex locking rules are getting too complex, ask yourself whether using channel(s) might be simpler.

Wait Group

Another important synchronisation primitive is sync.WaitGroup. These allow co-operating goroutines to collectively wait for a threshold event before proceeding independently again. This is useful typically in two cases.

Firstly, when 'cleaning up', a sync.WaitGroup can be used to ensure that all goroutines - including the main one - wait before all terminating cleanly.

The second more general case is of a cyclic algorithm that involves a set of goroutines that all work independently for a while, then all wait on a barrier, before proceeding independently again. This pattern might be repeated many times. Data might be exchanged at the barrier event. This strategy is the basis of Bulk Synchronous Parallelism (BSP).

Channel communication, mutexes and wait-groups are complementary and can be combined.

More Info

- Channels in Effective Go: https://go.dev/doc/effective_go#channels
- The sync package: https://pkg.go.dev/sync/