Tasks and Worker Processes

This lesson lists the possible ways of communication and their pros and cons in different situations.

WE'LL COVER THE FOLLOWING

- 1st paradigm: use shared memory to synchronize
- 2nd paradigm: channels
  - Use an in-channel and out-channel instead of locking
  - What to use: a `sync.Mutex` or a channel?
    - Using locking (mutexes)
    - Using channels

Suppose we have to perform several tasks; a task is performed by a worker (process). A `Task` can be defined as a struct (the concrete details are not important here):

```go
type Task struct {
  // some state
}
```

1st paradigm: use shared memory to synchronize

The pool of tasks is shared memory. To synchronize the work and to avoid race conditions, we have to guard the pool with a `Mutex` lock:

```go
type Pool struct {
  Mu sync.Mutex
  Tasks []Task
}
```

A `sync.Mutex` (as mentioned earlier in the course) is a *mutual exclusion* lock. It serves to guard the entrance to a critical section in code. Only one goroutine (thread) can enter that section at one time. If more than one goroutine is

allowed, a race-condition can exist, which means the `Pool` struct can no longer be updated correctly.

In the traditional model (applied in most classic OO-languages like C++, Java, and C#) the `Worker` process could be coded as:

```
func Worker(pool *Pool) {
  for {
    pool.Mu.Lock()
    // begin critical section:
    task := pool.Tasks[0] // take the first task
    pool.Tasks = pool.Tasks[1:] // update the pool of tasks
    // end critical section
    pool.Mu.Unlock()
    process(task)
  }
}
```

Many of these worker processes can run concurrently. They can certainly be started as goroutines. A worker locks the pool, takes the first task from the pool, unlocks the pool, and then processes the task. The lock guarantees that only one worker process at a time can access the pool. A task is assigned to one and only one process. If the lock is not there, the processing of the worker-routine can be interrupted in the lines `task := pool.Tasks[0]` and `pool.Tasks = pool.Tasks[1:]` with abnormal results. Some workers do not get a task, and several workers obtain some tasks. This locking synchronization works well for a few worker processes. Still, if the `Pool` is very big and we assign a large number of processes to work on it, the efficiency of the processing will be diminished by the overhead of the lock-unlock mechanism. This is the bottle-neck that causes performance to certainly decrease when the number of workers increases drastically at a certain threshold.

2nd paradigm: channels

In this case, channels of `Tasks` are used to synchronize. A pending channel receives the requested tasks, and a done channel receives the performed tasks (with their results). The worker processes are started as goroutines; their number **N** should be adjusted to the number of tasks. The main routine, which performs the function of `Master`, can be programmed as:

performs the function of `Master`, can be programmed as:

```go
func main() {
  pending, done := make(chan *Task), make(chan *Task)
  go sendWork(pending) // put tasks to do on the channel
  for i := 0; i < N; i++ { // start N goroutines to do work
    go Worker(pending, done)
  }
  consumeWork(done) // continue with the processed tasks
}
```

The worker process is very simple as it takes a task from the pending channel, processes it, and puts the finished task on the done channel:

```go
func Worker(in, out chan *Task) {
  for {
    t := <-in
    process(t)
    out <- t
  }
}
```

There is no locking; the process of getting a new task involves no contention. If the amount of tasks increases, the number of workers can be increased accordingly, and the performance will not degrade nearly as badly as in the 1st solution. From the pending channel, there is, of course, only one copy in memory. Still, there is no contention because, for the first `Worker` to finish the 1st pending task, it will have to process it completely since reading from and sending to a channel are atomic operations. It is impossible to predict which task will be performed by which process and vice versa. With an increasing number of workers, there is also an increasing communication overhead, which has a slight impact on performance.

In this simple example, it is perhaps difficult to see the advantage of the 2nd model, but applications with complex lock-situations are tough to program and to get right. A great deal of this complexity in the software is not needed in a solution that applies the 2nd model.

Thus not only performance is a significant advantage, but the clearer and more elegant code is perhaps an even bigger advantage. It is undoubtedly a Go idiomatic way of working:

Use an in-channel and out-channel instead of locking

```
func Worker(in, out chan *Task) {
  for {
    t := <-in
    process(t)
    out <- t
  }
}
```

For any problem, which can be modeled as such a **Master-Worker** paradigm, an analogous solution with `Workers` as goroutines communicating through channels and the `Master` as a coordinator is a perfect fit. If the system distributes over several machines, a number of machines could execute the `Worker` goroutines, and the `Master` and `Workers` could communicate amongst themselves through `netchan` or `rpc`.

What to use: a `sync.Mutex` or a channel?

Although in this chapter, we put a strong emphasis on goroutines using channels, because this is quite new in system languages, this doesn't mean that the classic approach with locking is now taboo. Go has both and gives you a choice according to the problem being solved. Construct the solution that is the most elegant, simplest, and most readable, and in most cases, performance will follow automatically. Don't be afraid to use a `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 rule of thumb:

Using locking (mutexes)

Use mutex when:

- Caching information in a shared data structure
- Holding state information, i.e., the context or status of a running application

Using channels

Use channels when:

- Communicating asynchronous results

- Distributing units of work
- Passing ownership of data

If you find your locking rules are getting too complicated, ask yourself if using channel(s) might be simpler.

Thus, when it comes to designing a way of communication, whether it be a shared memory or a channel, a lot has to be taken into consideration to get the most out of the process. In the next lesson, you’ll learn the basics of lazy evaluation.