Using goroutines for Performance

This lesson focuses on handling multiple clients at the same time through goroutines.

WE'LL COVER THE FOLLOWING

- Flags for user's interaction

There is still a performance problem with the **2nd version** if too many clients attempt to add URLs simultaneously. Our map is safely updated for concurrent access thanks to the locking mechanism, but the immediate writing of each new record to disk is a bottleneck. The disk writes may happen simultaneously, and depending on the characteristics of your OS, this may cause corruption. Even if the writes do not collide, each client must wait for their data to be written to disk before their `Put` function will return. Therefore, on a heavily I/O-loaded system, clients will wait longer than necessary for their `Add` requests to go through. To remedy these issues, we must decouple the `Put` and save processes; we do this by using Go's concurrency mechanism. Instead of saving records directly to disk, we send them to a channel, which is a kind of buffer, so the sending function doesn't have to wait for it.

The save process, writes to disk reads from that channel and is started on a separate thread by launching it as a goroutine called **saveloop**. The main program and **saveloop** are now executed concurrently, so there is no more blocking. We replace the file field in `URLStore` by a channel of records: `save chan record`.

```
type URLStore struct {
  urls map[string]string
  mu sync.RWMutex
  save chan record
}
```

A channel, just like a map, must be made with `make`; we will do this in our changed factory `NewURLStore` and give it a buffer length of 1000, like:

```
save := make(chan record, saveQueueLength)
```

To remedy our performance situation instead of making a function call to save each record to disk, `Put` can send a record to our buffered channel save:

```
func (s *URLStore) Put(url string) string {
  for {
    key := genKey(s.Count())
    if s.Set(key, url) {
      s.save <- record{key, url}
      return key
    }
  }
  panic("shouldn't get here")
}
```

At the other end of the `save` channel, we must have a receive. Our new method `saveLoop` will run in a separate goroutine; it receives record values and writes them to a file. The `saveLoop` is also started in the `NewURLStore()` function with the keyword `go`, and we can remove the now-unnecessary file opening code. Here is the modified `NewURLStore()`:

```
const saveQueueLength = 1000
func NewURLStore(filename string) *URLStore {
  s := &URLStore{
  urls: make(map[string]string),
  save: make(chan record, saveQueueLength),
  }
  if err := s.load(filename); err != nil {
    log.Println("Error loading URLStore:", err)
  }
  go s.saveLoop(filename)
  return s
}
```

Here is the code for the method `saveloop`:

```
func (s *URLStore) saveLoop(filename string) {
  f, err := os.Open(filename, os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0644)
  if err != nil {
    log.Fatal("URLStore:", err)
```

```
    log.Fatal("URLStore: ", err)
  }
  defer f.Close()
  e := gob.NewEncoder(f)
  for {
    r := <-s.save // taking a record from the channel and encoding it
    if err := e.Encode(r); err != nil {
      log.Println("URLStore:", err)
    }
  }
}
```

Records are read from the `save` channel in an infinite loop and encoded to the file. In chapter 12, we studied goroutines and channels in-depth, but here we have seen a useful example for better managing the different parts of a program. Notice also that now we only make our `Encoder` object once.

Flags for user's interaction

Another improvement can be introduced to make `goto` more flexible. Instead of coding the filename, the listener address and the hostname hard-coded, or as constants in the program, we can define them as flags. That way, they can be given new values if they are typed in on the command line when starting the program. If this is not done, the default value from the flag will be taken. This functionality comes from a different package, so we have to: `import "flag"`.

We first create some global variables to hold the flag values:

```
var (
  listenAddr = flag.String("http", ":3000", "http listen address")
  dataFile = flag.String("file", "store.gob", "data store file name")
  hostname = flag.String("host", "1dkne4jl5mmmm.educative.run", "host nam
e and port")
)
```

In your case, the url will be different which is next to the **Your app can be found at**.

For processing command-line parameters, we must add `flag.Parse()` to the `main` function, and instantiate the `URLStore` after the flags have been parsed, once we know the value of `dataFile` (in the code `*dataFile` is used. This is because a flag is a pointer and must be dereferenced to get the value):

because a flag is a pointer and must be dereferenced to get the value).

```
var store *URLStore
func main() {
  flag.Parse()
  store = NewURLStore(*dataFile)
  http.HandleFunc("/", Redirect)
  http.HandleFunc("/add", Add)
  http.ListenAndServe(*listenAddr, nil)
}
```

Compile and test this **Version-3** like in the previous versions.

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main

import (
        "flag"
        "fmt"
        "net/http"
)

var (
        listenAddr = flag.String("http", ":3000", "http listen address")
        dataFile   = flag.String("file", "store.gob", "data store file name")
        hostname   = flag.String("host","1dkne4jl5mmmm.educative.run", "http host name")
)

var store *URLStore

func main() {
        flag.Parse()
        store = NewURLStore(*dataFile)
        http.HandleFunc("/", Redirect)
        http.HandleFunc("/add", Add)
        http.ListenAndServe(*listenAddr, nil)
}

func Redirect(w http.ResponseWriter, r *http.Request) {
        key := r.URL.Path[1:]
        url := store.Get(key)
        if url == "" {
                http.NotFound(w, r)
                return
        }
        http.Redirect(w, r, url, http.StatusFound)
```

```
}

func Add(w http.ResponseWriter, r *http.Request) {

        w.Header().Set("Content-Type", "text/html")
        url := r.FormValue("url")
        if url == "" {
                fmt.Fprint(w, addForm)
                return
        }
        key := store.Put(url)
        fmt.Fprintf(w, "%s", key)
}

const addForm = `
<html><body>
<form method="POST" action="/add">
URL: <input type="text" name="url">
<input type="submit" value="Add">
</form>
</html></body>
`
```

Click the **RUN** button and type `go run *.go`.

In case you want to run multiple clients, open console-windows. In each terminal, a client process is started performing the following steps for each separate terminal:

- Type `usr/local/go/src` and press ENTER.
- Type `go run *.go` and press ENTER.

Click the URL next to **Your app can be found at:** .

> **Remark:** To run it locally, change the port in **line 10** to `8080`. Change **line 12** as `hostname = flag.String("host","localhost:8080", "http host name")`. To test this program open browser at http://localhost:8080.

That's how performance can be made efficient using gob. The next lesson focuses on JSON to achieve the same.